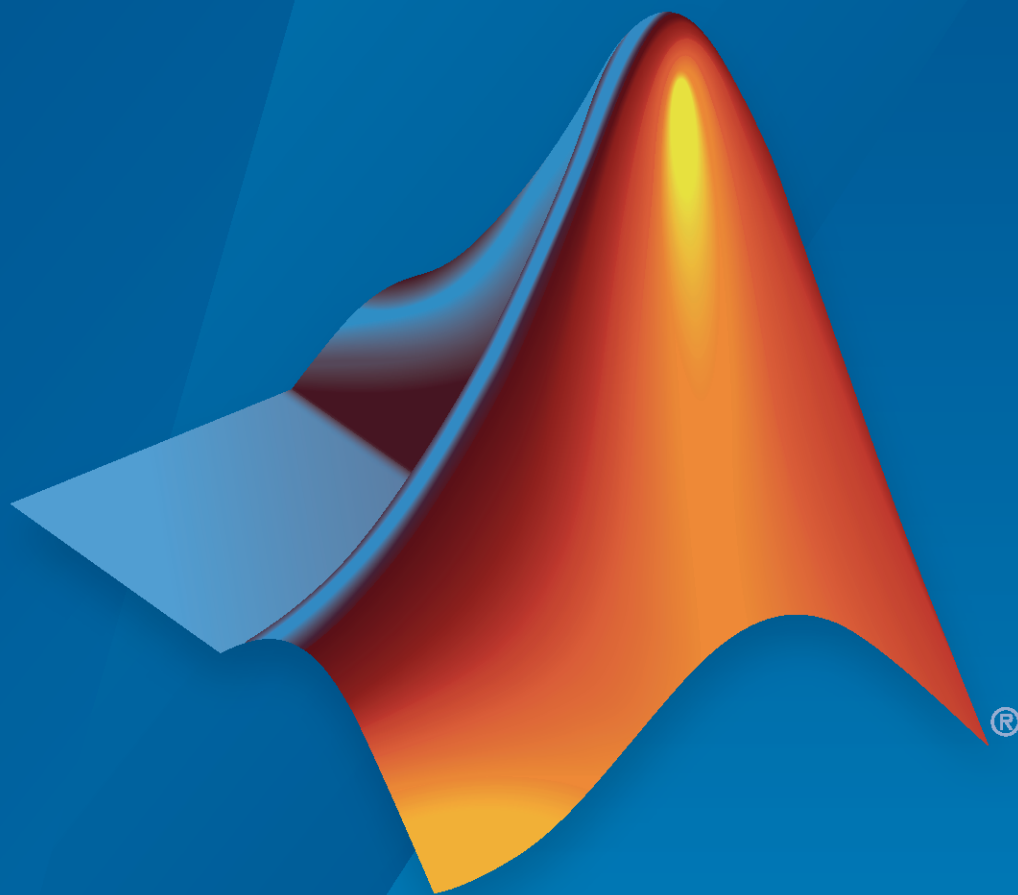


Polyspace[®] Code Prover[™] Server[™]

User's Guide



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.
1 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Code Prover™ Server™ User's Guide

© COPYRIGHT 2019-2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2019	Online only	New for Version 10.0 (R2019a)
September 2019	Online Only	Revised for Version 10.1 (Release 2019b)
March 2020	Online Only	Revised for Version 10.2 (Release 2020a)
September 2020	Online Only	Revised for Version 10.3 (Release 2020b)

1

Polyspace Analysis on Server After Code Submission

Prepare Scripts for Polyspace Analysis	1-2
Options Related to Source Code and Target	1-2
Options Related to Polyspace Analysis	1-4
Offload Polyspace Analysis from Continuous Integration Server to Another Server	1-6
Install Products	1-6
Configure and Start Job Scheduler Services on Head Node and Worker Node	1-8
Offload Analysis from Client Node	1-9
Configure Polyspace Analysis Options in User Interface and Generate Scripts	1-11
Prerequisites	1-12
Generate Scripts from Configuration	1-12
Run Analysis with Generated Scripts	1-13
Sample Scripts for Polyspace Analysis with Jenkins	1-15
Extending Sample Scripts to Your Development Process	1-15
Prerequisites	1-16
Set Up Polyspace Plugin in Jenkins	1-17
Script to Run Bug Finder, Upload Results and Send Common Notification	1-21
Script to Run Bug Finder, Upload Results and Send Personalized Notification	1-22
Sample Jenkins Pipeline Scripts for Polyspace Analysis	1-29
Prerequisites	1-29
Run Polyspace Analysis in Stages in a Pipeline Script	1-29
Run Polyspace Analysis on Generated Code by Using Packaged Options Files	1-31
Generate and Package Polyspace Options Files	1-31
Run Polyspace Analysis by Using the Packaged Options Files	1-32
Analyze Code Generated as Standalone Code in a Distributed Workflow	1-33

Use Existing Software Development Specifications for Polyspace Analysis

2

Create Polyspace Analysis Configuration from Build Command	2-2
polyspace-configure Source Files Selection Syntax	2-4
Modularize Polyspace Analysis by Using Build Command	2-6
Build Source Code	2-6
Create One Polyspace Options File for Full Build	2-8
Create Options File for Specific Binary in Build Command	2-9
Create One Options File Per Binary Created in Build Command	2-9
Create Polyspace Analysis Configuration from AUTOSAR Specifications	2-12
Benefits of Polyspace for AUTOSAR	2-12
Run Polyspace on AUTOSAR Code	2-12
Upload Results to Polyspace Access Web Interface	2-13
Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis	2-16
Adapt Linux find Command to Select Files	2-16
File Selection Options	2-16
File Selection Examples	2-17
Root Folder Specification	2-18
Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code	2-19
Overview of File Structure	2-19
See File Selections	2-19
Run Analysis	2-20
Interpret Warnings	2-20

Offload Polyspace Analysis to Remote Servers from Desktop

3

Send Polyspace Analysis from Desktop to Remote Servers	3-2
Client-Server Workflow for Running Analysis	3-2
Prerequisites	3-3
Offload Analysis in Polyspace User Interface	3-3
Send Polyspace Analysis from Desktop to Remote Servers Using Scripts	3-6
Client-Server Workflow for Running Analysis	3-6
Prerequisites	3-7
Run Remote Analysis	3-7
Manage Remote Analysis	3-8
Sample Scripts for Remote Analysis	3-10

Run Polyspace Analysis on Server with MATLAB Scripts

4

Integrate Polyspace Server Products with MATLAB and Simulink	4-2
Integrate Polyspace with MATLAB Installation from Same Release	4-2
Integrate Polyspace with MATLAB Installation from Different Release	4-2
Check Integration Between MATLAB and Polyspace	4-3
Run Polyspace Server Products with MATLAB Scripts	4-3

Configure Target and Compiler Options

5

Specify Target Environment and Compiler Behavior	5-2
Extract Options from Build Command	5-2
Specify Options Explicitly	5-3
C/C++ Language Standard Used in Polyspace Analysis	5-5
Supported Language Standards	5-5
Default Language Standard	5-5
C11 Language Elements Supported in Polyspace	5-7
C++11 Language Elements Supported in Polyspace	5-9
C++14 Language Elements Supported in Polyspace	5-12
C++17 Language Elements Supported in Polyspace	5-15
Provide Standard Library Headers for Polyspace Analysis	5-19
Requirements for Project Creation from Build Systems	5-20
Compiler Requirements	5-20
Build Command Requirements	5-21
Supported Keil or IAR Language Extensions	5-23
Special Function Register Data Type	5-23
Keywords Removed During Preprocessing	5-24
Remove or Replace Keywords Before Compilation	5-25
Remove Unrecognized Keywords	5-25
Remove Unrecognized Function Attributes	5-27
Gather Compilation Options Efficiently	5-28

Configure Inputs and Stubbing Options

6

Specify External Constraints	6-2
Create Constraint Template	6-2
Create Constraint Template from Code Prover Analysis Results	6-4
Update Existing Template	6-4
Specify Constraints in Code	6-5
External Constraints for Polyspace Analysis	6-7
Constraint Specification Limitations	6-11
Constrain Global Variable Range	6-13
User Interface (Desktop Products Only)	6-13
Command Line	6-14
Constrain Function Inputs	6-16
User Interface (Desktop Products Only)	6-16
Command Line	6-17
XML File Format for Constraints	6-19
Syntax Description — XML Elements	6-19
Valid Modes and Default Values	6-23

Configure Multitasking Analysis

7

Analyze Multitasking Programs in Polyspace	7-2
Configure Analysis	7-2
Review Analysis Results	7-3
Auto-Detection of Thread Creation and Critical Section in Polyspace ...	7-5
Multitasking Routines that Polyspace Can Detect	7-5
Example of Automatic Thread Detection	7-7
Naming Convention for Automatically Detected Threads	7-10
Limitations of Automatic Thread Detection	7-11
Configuring Polyspace Multitasking Analysis Manually	7-16
Specify Options for Multitasking Analysis	7-16
Adapt Code for Code Prover Multitasking Analysis	7-16
Protections for Shared Variables in Multitasking Code	7-20
Detect Unprotected Access	7-20
Protect Using Critical Sections	7-21
Protect Using Temporally Exclusive Tasks	7-22
Protect Using Priorities	7-22
Protect By Disabling Interrupts	7-23
Define Atomic Operations in Multitasking Code	7-24
Nonatomic Operations	7-24
What Polyspace Considers as Nonatomic	7-24

Define Specific Operations as Atomic	7-25
Define Preemptable Interrupts and Nonpreemptable Tasks	7-27
Emulating Task Priorities	7-27
Examples of Task Priorities	7-27
Further Explorations	7-28
Define Critical Sections with Functions That Take Arguments	7-30
Polyspace Assumption on Functions Defining Critical Sections	7-30
Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments	7-30

Configure Coding Rules Checking and Code Metrics Computation

8

Check for Coding Standard Violations	8-2
Configure Coding Rules Checking	8-2
Review Coding Rule Violations	8-6
Generate Reports	8-7
Avoid Violations of MISRA C:2012 Rules 8.x	8-8
Software Quality Objective Subsets (C:2004)	8-11
Rules in SQO-Subset1	8-11
Rules in SQO-Subset2	8-12
Software Quality Objective Subsets (AC AGC)	8-15
Rules in SQO-Subset1	8-15
Rules in SQO-Subset2	8-15
Software Quality Objective Subsets (C:2012)	8-18
Guidelines in SQO-Subset1	8-18
Guidelines in SQO-Subset2	8-19
Software Quality Objective Subsets (C++)	8-21
SQO Subset 1 - Direct Impact on Selectivity	8-21
SQO Subset 2 - Indirect Impact on Selectivity	8-22
Coding Rule Subsets Checked Early in Analysis	8-27
MISRA C:2004 and MISRA AC AGC Rules	8-27
MISRA C:2012 Rules	8-34
Create Custom Coding Rules	8-42
User Interface (Desktop Products Only)	8-42
Command Line	8-43
Compute Code Complexity Metrics	8-44
Impose Limits on Metrics (Desktop Products Only)	8-44
Impose Limits on Metrics (Server and Access products)	8-46

HIS Code Complexity Metrics	8-47
Project	8-47
File	8-47
Function	8-47

Configure Verification of Modules or Libraries

9

Provide Context for C Code Verification	9-2
Control Variable Range	9-2
Control Function Call Sequence	9-2
Control Stubbing Behavior	9-2
Provide Context for C++ Code Verification	9-4
Control Variable Range	9-4
Control Function Call Sequence	9-4
Verify C Application Without main Function	9-6
Generate main Function	9-6
Manually Write main Function	9-6
Verify C++ Classes	9-9
Verification of Classes	9-9
Methods and Class Specifics	9-11

Configure Code Prover Run-Time Checks

10

Modify or Disable Code Prover Run-Time Checks	10-2
Integer Overflow	10-2
Floating Point Overflow	10-3
Initialization	10-3
Library Functions	10-3
Pointers	10-4
Unreachable Code or Dead Code	10-4

Configure Comment Import from Previous Results

11

Import Review Information from Previous Polyspace Analysis	11-2
Automatic Import from Last Analysis	11-2
Import from Another Analysis Result	11-2
Import Algorithm	11-3
View Imported Review Information That Does Not Apply	11-4

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results	11-6
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result	11-7

12

Troubleshooting in Polyspace Code Prover Server

Read Error Information When Polyspace Analysis Stops	12-3
Troubleshoot Compilation and Linking Errors	12-4
Issue	12-4
Possible Cause: Deviations from ANSI C99 Standard	12-4
Possible Cause: Linking Errors	12-5
Possible Cause: Conflicts with Polyspace Function Stubs	12-5
Reduce Memory Usage and Time Taken by Polyspace Analysis	12-7
Issue	12-7
Possible Cause: Anti-Virus Software	12-7
Possible Cause: Large and Complex Application	12-7
Possible Cause: Too Many Entry Points for Multitasking Applications	12-9
Contact Technical Support About Issues with Running Polyspace	12-11
Provide System Information	12-11
Provide Information About the Issue	12-11
Polyspace Cannot Find the Server	12-14
Message	12-14
Possible Cause	12-14
Solution	12-14
Job Manager Cannot Write to Database	12-15
Message	12-15
Possible Cause	12-15
Workaround	12-15
Compiler Not Supported for Project Creation from Build Systems	12-16
Issue	12-16
Cause	12-16
Solution	12-16
Slow Build Process When Polyspace Traces the Build	12-22
Issue	12-22
Cause	12-22
Solution	12-22
Check if Polyspace Supports Build Scripts	12-23
Issue	12-23
Possible Cause	12-23
Solution	12-23

Troubleshooting Project Creation from MinGW Build	12-25
Issue	12-25
Cause	12-25
Solution	12-25
Troubleshooting Project Creation from Visual Studio Build	12-26
Error Processing Macro with Semicolon in Build System	12-27
Issue	12-27
Cause	12-27
Solution	12-27
Resolve polspace-autosar Error: Could Not Find Include File	12-28
Issue	12-28
Possible Solutions	12-28
Resolve polspace-autosar Error: Conflicting Universal Unique Identifiers (UUIDs)	12-30
Issue	12-30
Possible Solutions	12-30
Resolve polspace-autosar Error: Data Type Not Recognized	12-31
Issue	12-31
Possible Solutions	12-31
Undefined Identifier Error	12-33
Issue	12-33
Possible Cause: Missing Files	12-33
Possible Cause: Unrecognized Keyword	12-33
Possible Cause: Declaration Embedded in #ifdef Statements	12-34
Possible Cause: Project Created from Non-Debug Build	12-34
Unknown Function Prototype Error	12-36
Issue	12-36
Cause	12-36
Solution	12-36
Error Related to #error Directive	12-37
Issue	12-37
Cause	12-37
Solution	12-37
Large Object Error	12-38
Issue	12-38
Cause	12-38
Solution	12-38
Errors Related to Generic Compiler	12-40
Issue	12-40
Cause	12-40
Solution	12-40
Errors Related to Keil or IAR Compiler	12-41
Missing Identifiers	12-41

Errors Related to Diab Compiler	12-42
Issue	12-42
Cause	12-42
Solution	12-42
Errors Related to Green Hills Compiler	12-44
Issue	12-44
Cause	12-44
Solution	12-44
Errors Related to TASKING Compiler	12-46
Issue	12-46
Cause	12-46
Solution	12-46
Errors from In-Class Initialization (C++)	12-48
Errors from Double Declarations of Standard Template Library Functions (C++)	12-49
Errors Related to GNU Compiler	12-50
Issue	12-50
Cause	12-50
Solution	12-50
Errors Related to Visual Compilers	12-51
Import Folder	12-51
pragma Pack	12-51
C++/CLI	12-52
Conflicting Declarations in Different Translation Units	12-53
Issue	12-53
Possible Cause: Variable Declaration and Definition Mismatch	12-54
Possible Cause: Function Declaration and Definition Mismatch	12-54
Possible Cause: Conflicts from Unrelated Declarations	12-55
Possible Cause: Macro-dependent Definitions	12-56
Possible Cause: Keyword Redefined as Macro	12-56
Possible Cause: Differences in Structure Packing	12-57
Errors from Conflicts with Polyspace Header Files	12-58
Issue	12-58
Cause	12-58
Solution	12-58
C++ Standard Template Library Stubbing Errors	12-59
Issue	12-59
Cause	12-59
Solution	12-59
Lib C Stubbing Errors	12-60
Extern C Functions	12-60
Functional Limitations on Some Stubbed Standard ANSI Functions ...	12-60
Errors from Using Namespace std Without Prefix	12-62
Issue	12-62

Cause	12-62
Solution	12-62
Errors from Assertion or Memory Allocation Functions	12-63
Issue	12-63
Cause	12-63
Solution	12-63
Error or Slow Runs from Disk Defragmentation and Anti-virus Software	
.....	12-64
Issue	12-64
Possible Cause	12-64
Solution	12-64
SQLite I/O Error	12-66
Issue	12-66
Cause	12-66
Solution	12-66
License Error -4,0	12-67
Issue	12-67
Possible Cause: Another Polyspace Instance Running	12-67
Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder ...	12-67

Polyspace Analysis on Server After Code Submission

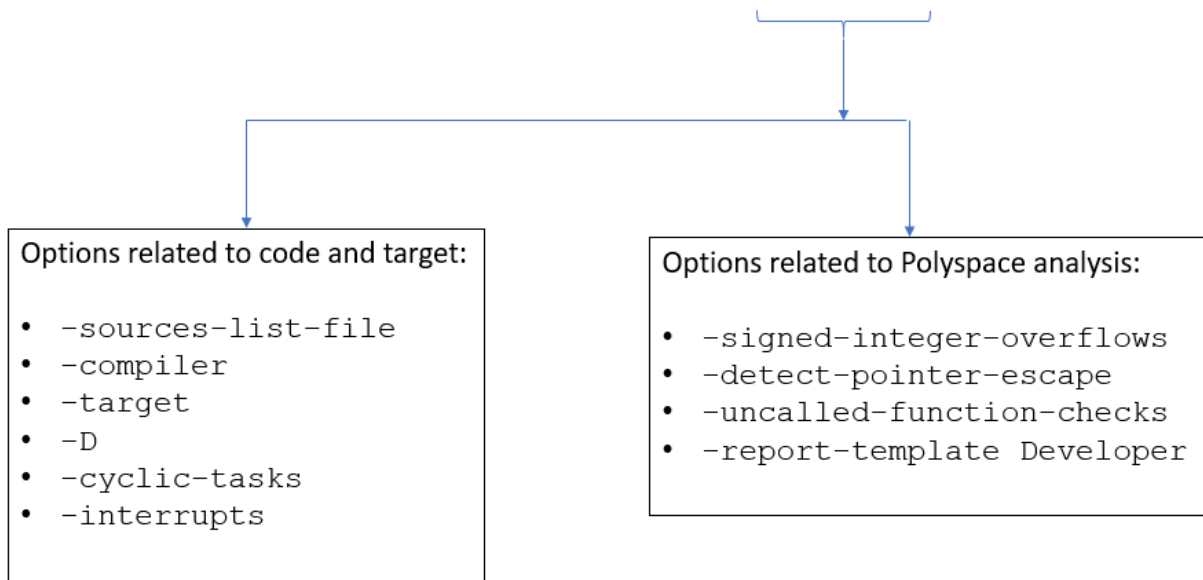
Prepare Scripts for Polyspace Analysis

When you run Polyspace as part of your software development processes, your analysis scripts must be preconfigured for new code submissions. For instance, new source files must be automatically included in the Polyspace analysis. To keep the analysis configuration updated with new submissions, you can leverage existing artifacts such as your build command (makefiles) and create your analysis configuration on the fly when new submissions occur.

The analysis configuration consists of two parts:

- Options related to the source code and target, such as data type sizes, macro definitions, cyclic tasks and interrupts, and so on.
- Options related to the analysis, such as checkers, code verification assumptions, and so on.

```
polyspace-code-prover-server -options-file file.opts
```



Options Related to Source Code and Target

The most common options related to the source code and target are:

- `-sources-list-file`: Specify a text file containing one source file per line.
- `-I`: Specify the folders containing included header files.
- **Compiler** (`-compiler`): Specify the compiler used for building your source code.
- **Target processor type** (`-target`): Specify sizes of data types and endianness by selecting a predefined target processor.

- **Preprocessor definitions (-D):** Replace unrecognized code for the purposes of Polyspace analysis. You typically use this option if the analysis shows compilation errors from compiler-specific keywords and macros.
- **Constraint setup (-data-range-specifications):** Define external constraints on global variables and function interfaces. The option is typically useful for a more precise Code Prover analysis.

For the full list of options, see:

- “Analysis Options”
- “Analysis Options”

Extract Options from Build Command

In a continuous integration workflow, you typically do not specify the option arguments explicitly. Your build command contains the specifications for sources, compiler, macro definitions and so on. Run the `polyspace-configure` command to extract these specifications from your build command and create an options file. For instance, if you use `make` to build your source code, run the analysis as follows:

```
polyspace-configure -output-options-file polyspace_opts make
polyspace-bug-finder-server -options-file polyspace_opts
polyspace-code-prover-server -options-file polyspace_opts
```

The first command extracts source and target specifications by executing the instructions in the makefile and creates an analysis options file. The second and third commands runs a Bug Finder and Code Prover analysis with the options file. See “Create Polyspace Analysis Configuration from Build Command” on page 2-2.

Specify Options Explicitly in Options File

If you cannot extract the options from your build command, specify the options explicitly. You can create some of the option arguments on the fly from new submissions. For instance, the argument for the option `-sources-list-file` is a text file that lists the sources. You can update this text file based on any new source file added to the source code repository.

If you have to specify the target and compiler options explicitly, you might not get all the options right in the first run. To find the right combination of options:

- 1 Specify the options `Compiler (-compiler)` and `Target processor type (-target)` in your options file.
- 2 Compile the code with your compiler and fix all compilation errors. Then, run only the compilation part of the Polyspace analysis.
 - In Bug Finder, disable all checkers. Specify `-checkers none` in the options file. See `Find defects (-checkers)`.
 - In Code Prover, stop the analysis after compilation. Specify `-to compile` in the options file. See `Verification level (-to)`.

If you run into compilation errors, you might have to work around the errors with Polyspace options. For instance, if you see a compilation error because the macro `_WIN32` is defined with a

compiler option but Polyspace considers the macro as undefined by default, emulate your compiler option with the Polyspace option `-D _WIN32`. See “Target and Compiler”, “Macros” and “Environment Settings” for the target and compiler options.

Once you fix all compilation errors with Polyspace analysis options, your options file is prepared with the right set of Polyspace options for the analysis.

If you have an installation of the desktop products, Polyspace Bug Finder™ and/or Polyspace Code Prover, you can perform the trial runs in the user interface of the desktop products. You can then generate an options file from the configuration defined in the user interface. The user interface provides various features such as:

- Compilation assistant that suggests workarounds for some compilation errors,
- Auto-generation of XML file for constraint specification,
- Context-sensitive help for options,

See “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 1-11.

Options Related to Polyspace Analysis

Some options related to the Polyspace analysis are:

Bug Finder

- `Find defects (-checkers)`: Specify checkers to enable for the Bug Finder analysis.
- `Check MISRA C:2012 (-misra3)` and other options related to external standards: Specify an external standard and a predefined subset of that standard.
- `Set checkers by file (-checkers-selection-file)`: Specify a custom subset of rules from external standards.
- `Bug Finder and Code Prover report (-report-template)`: Specify that a PDF, Word or HTML report must be generated along with the analysis results and specify a template for the report.
- `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`: Offload the analysis to another server. See “Offload Polyspace Analysis from Continuous Integration Server to Another Server”.

Code Prover

- `Overflow mode for signed integer (-signed-integer-overflows)`: Specify the behavior following an overflow: stop analysis or continue with wrap-around.
- `Detect stack pointer dereference outside scope (-detect-pointer-escape)`: Specify if the analysis must find cases where a function returns a pointer to one of its local variables.
- `Detect uncalled functions (-uncalled-function-checks)`: Specify if the analysis must flag functions that are not called directly or indirectly from main or another entry point function.
- `Bug Finder and Code Prover report (-report-template)`: Specify that a PDF, Word or HTML report must be generated along with the analysis results and specify a template for the report.

- Run Bug Finder or Code Prover analysis on a remote cluster (-batch): Offload the analysis to another server. See “Offload Polyspace Analysis from Continuous Integration Server to Another Server” on page 1-6.

The checkers and other options related to the Polyspace analysis can be applicable to more than one project. To maintain uniform standards across projects, you can reuse this subset of analysis options. When running the analysis, specify two options files, one containing the options specific to the current project and the other containing the reusable options. You can extract the first options file from your build command but explicitly create the second options file.

For instance, in this example, the `polyspace-bug-finder-server` command uses two options files: `compile_opts` generated from a makefile and `runbf_opts` created manually. All reusable options can be specified in `runbf_opts`.

```
polyspace-configure -output-options-file compile_opts make
polyspace-bug-finder-server -options-file compile_opts -options-file runbf_opts
polyspace-code-prover-server -options-file compile_opts -options-file runcp_opts
```

If the same option appears in two options files, the last instance of the option is considered. In the preceding example, if an option occurs in both `compile_opts` and `runbf_opts`, the occurrence in `runbf_opts` is considered. If you want to override previous occurrences of an option, use an additional options file with your overrides. Append this options file to the end of the analysis command.

See Also

`polyspace-code-prover-server` | `polyspace-configure`

More About

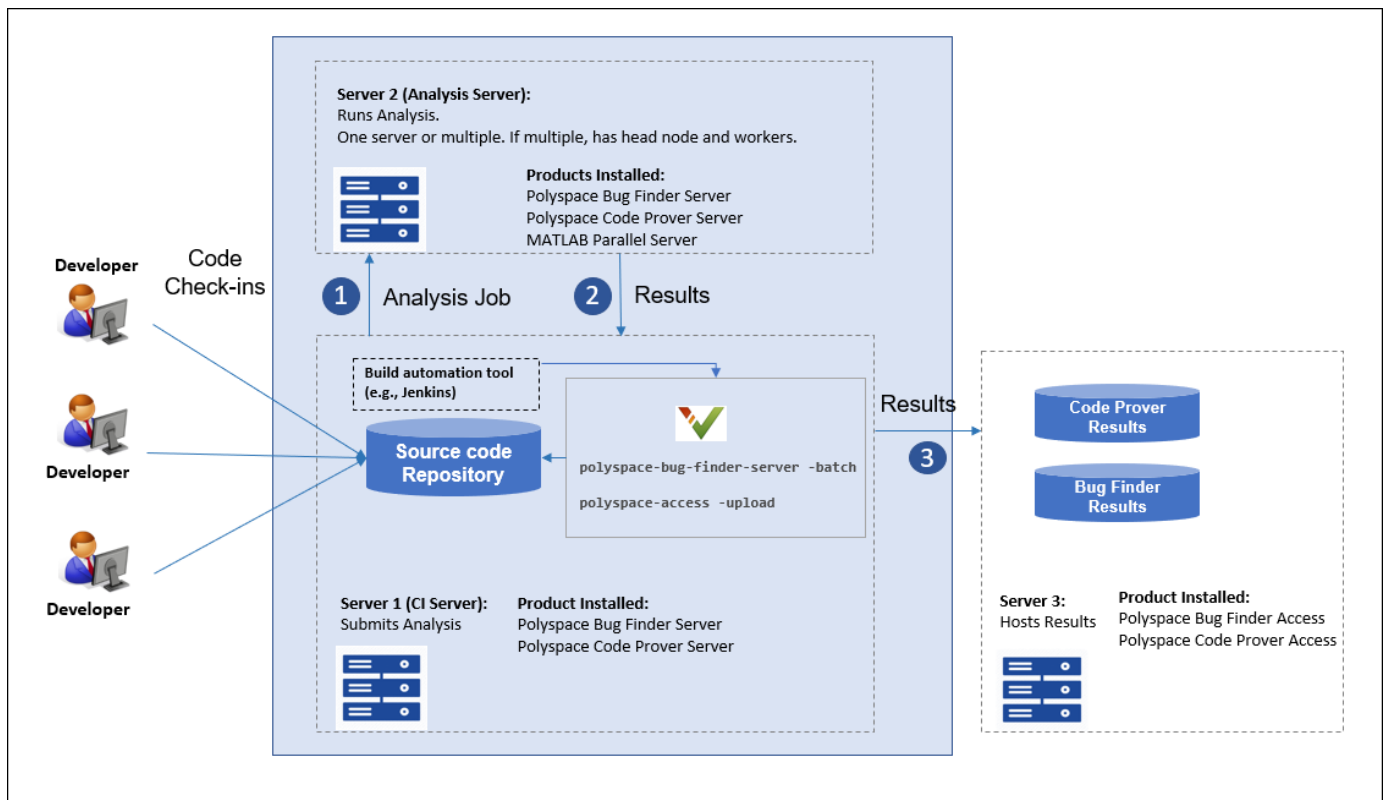
- “Run Polyspace Code Prover on Server and Upload Results to Web Interface”
- “Create Polyspace Analysis Configuration from Build Command” on page 2-2
- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 1-11

Offload Polyspace Analysis from Continuous Integration Server to Another Server

When running static code analysis with Polyspace as part of continuous integration, you might want the analysis to run on a server that is different from the server running your continuous integration (CI) scripts. For instance:

- You might want to perform the analysis on a server that has more processing power. You can offload the analysis from your CI server to the other server.
- You might want to submit analysis jobs from several CI servers to a dedicated analysis server, hold the jobs in queue, and execute them as Polyspace Server instances become available.

When you offload an analysis, the compilation phase of the analysis runs on the CI server. After compilation, the analysis job is submitted to the other server and continues on this server. On completion, the analysis results are downloaded back to the CI server. You can then upload the results to Polyspace Access for review, or report the results in some other format.



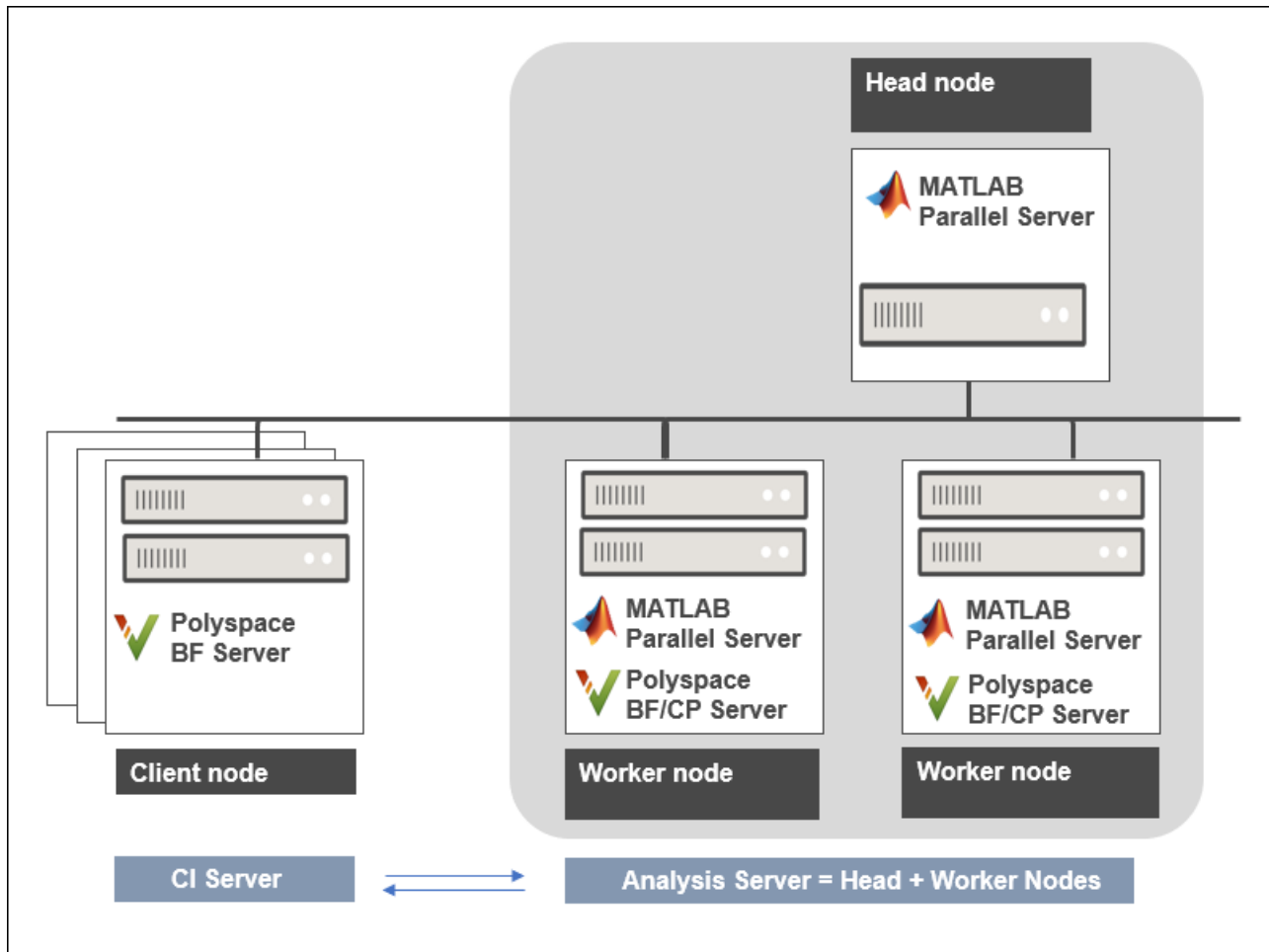
Install Products

A typical distributed network for offloading an analysis consists of these parts:

- **Client node(s):** Each CI server acts as a client node that submits Polyspace analysis jobs to a cluster.

The cluster consists of a head node and one or more worker nodes. In this example, we use the same computer as the head node and one worker node.

- **Head node:** The head node distributes the submitted jobs to worker nodes.
- **Worker node(s):** Each worker node executes one Polyspace analysis at a time.



Install these products:

- **Client nodes:** Polyspace Bug Finder Server or Polyspace Code Prover Server to submit jobs from the Continuous Integration server.
- **Head node:** MATLAB® Parallel Server™ to manage submissions from multiple clients. An analysis job is created for each submission and placed in a queue. As soon as a worker node is available, the next analysis job from the queue is run on the worker.
- **Worker node(s):** MATLAB Parallel Server and Polyspace Bug Finder Server or Polyspace Code Prover Server on the worker nodes to run a Bug Finder or Code Prover analysis.

In the simplest configuration, where the same computer serves as the head node and one worker node, you install MATLAB Parallel Server and one or both Polyspace Bug Finder Server and Polyspace Code Prover Server on this computer. This example describes the simple configuration but you can generalize the steps to multiple workers on separate computers.

Configure and Start Job Scheduler Services on Head Node and Worker Node

Start a job scheduler service (the MATLAB Job Scheduler or `mjs` service) on the computer that acts as the head node and worker node. Before starting the service, you must perform an initial setup.

Specify Polyspace Installation Paths

MATLAB Parallel Server and Polyspace Server products are installed in two separate folders. The MATLAB Parallel Server installation routes the Polyspace analysis to the Polyspace Server products. To link the two installations, specify the path to the root folder of the Polyspace Server products in your MATLAB Parallel Server installation.

- 1 Navigate to `matlabroot\toolbox\parallel\bin\`. Here, `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2020b`.
- 2 Uncomment and modify the following line in the file `mjs_polyspace.conf`. To edit and save the file, open your editor in administrator mode.

```
POLYSPACE_SERVER_ROOT=polyspaceserverroot
```

Here, `polyspaceserverroot` is the installation path of the server products, for instance:

```
C:\Program Files\Polyspace Server\R2020b
```

The Polyspace Server product offloading the analysis must belong to the same release as the Polyspace Server product running the analysis. If you offload an analysis from an R2020b Polyspace Server product, the analysis must run using another R2020b Polyspace Server product.

Configure mjs Service Settings

Before starting MATLAB Parallel Server (the `mjs` service), you must perform a minimum configuration.

- 1 Navigate to `matlabroot\toolbox\parallel\bin`, where `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2020b`.
- 2 Modify the file `mjs_def.bat` (Windows®) or `mjs_def.sh` (Linux®). To edit and save the file, open your editor in administrator mode.

Read the instructions in the file and uncomment the lines as needed. At a minimum, uncomment these lines that specify:

- Host name.

Windows:

```
REM set HOSTNAME=%strHostname%.%strDomain%
```

Linux:

```
#HOSTNAME=`hostname -f`
```

Explicitly specify your computer host name.

- Security level.

Windows:

```
REM set SECURITY_LEVEL=
```

Linux:

```
#SECURITY_LEVEL=""
```

Explicitly specify a security level to avoid future errors when starting the job scheduler.

For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission.

Start mjs Service and One Worker

In a command-line terminal, `cd` to `matlabroot\toolbox\parallel\bin`, where `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2020b`. Run these commands (directly at the command line or by using scripts):

```
mjs install
mjs start
startjobmanager -name JobScheduler -remotehost hostname -v
startworker -jobmanagerhost hostname -jobmanager JobScheduler
               -remotehost hostname -v
```

Here, `hostname` is the host name of your computer. This name is the host name that you specified in the file `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux).

For more details and configuring services with multiple workers, see:

- “Install and Configure MATLAB Parallel Server for MATLAB Job Scheduler and Network License Manager” (MATLAB Parallel Server)
- `mjs`

Offload Analysis from Client Node

Once you have set up the computer that acts as the head node and worker node, you are ready to offload a Polyspace analysis from the client node (the CI server running scripts on Jenkins on another CI system).

To offload an analysis, enter:

```
polyspaceserverroot\polyspace\bin\polyspace-code-prover-server  
-batch -scheduler hostname|MJSName@hostname [options] [-mjs-username name]
```

where:

- *polyspaceserverroot* is the installation folder of Polyspace Server products on the client node, for instance, C:\Program Files\Polyspace Server\R2020b.
- *hostname* is the host name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

MJSName is the name of the MATLAB Job Scheduler on the head node host.

If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *MJSName* is the argument of the option `-name`.

- *options* are the Polyspace analysis options. These options are the same as that of a local analysis. For instance, you can use these options:
 - `-sources-list-file`: Specify a text file that has one source file name per line.
 - `-options-file`: Specify a text file that has one option per line.
 - `-results-dir`: Specify a download folder for storing results after analysis.

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

- *name* is the user name required for job submissions using MATLAB Parallel Server. This credential is required only if you use a security level of 1 or higher for MATLAB Parallel Server submissions. See “Set MATLAB Job Scheduler Cluster Security” (MATLAB Parallel Server).

For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission. To avoid this prompt in the future, you can specify that the password be remembered on the computer.

The analysis executes locally on the CI server up to the end of the compilation phase. After compilation, the analysis job is submitted to the other server. On completion, the analysis results are downloaded back to the CI server. You can then upload the results to Polyspace Access for review, or report the results in some other format.

See Also

`polyspace-access`

More About

- “Run Polyspace Code Prover on Server and Upload Results to Web Interface”

Configure Polyspace Analysis Options in User Interface and Generate Scripts

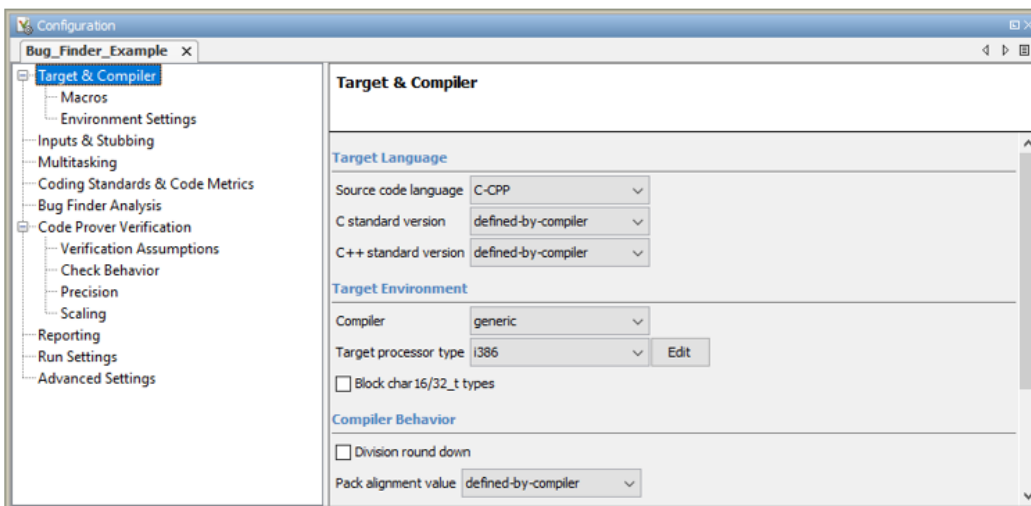
In this section...

“Prerequisites” on page 1-12

“Generate Scripts from Configuration” on page 1-12

“Run Analysis with Generated Scripts” on page 1-13

If you have an installation of the desktop products, Polyspace Bug Finder and/or Polyspace Code Prover, you can configure your project in the user interface of the desktop products. You can then generate a script or an options file from the configuration defined in the user interface and use the script or options file for automated runs with the desktop or server products.



```
polyspace -generate-launching-script-for Bug_Finder_Example.psrpj -bug-finder
polyspace -generate-launching-script-for Code_Prover_Example.psrpj
```

```
-target x86_64
-c-version c11
-compiler gnu4.6
-dos
-sources-list-file source_command.txt
...
```

Unless you create a Polyspace project from existing specifications such as a build command, when setting up the project, you might have to perform a few trial runs first. In these trial runs, if you run into compilation errors or unchecked code, you might have to modify your analysis configuration. It is easier performing this initial setup in the user interface of the desktop products. The user interface provides various features such as:

- Compilation assistant that suggests workarounds for some compilation errors,
- Auto-generation of XML file for constraint specification,
- Context-sensitive help for options.

Prerequisites

You must have at least one license of Polyspace Bug Finder and/or Polyspace Code Prover to open the Polyspace user interface and configure the options.

After generating the scripts, you can run the analysis using either the desktop products (Polyspace Bug Finder and Polyspace Code Prover) or the server products (Polyspace Bug Finder Server and/or Polyspace Code Prover Server).

Generate Scripts from Configuration

This example shows how to generate a script from a Bug Finder configuration. The same steps apply to a Code Prover configuration.

- 1 Add source files to a new project in the Polyspace user interface.

Navigate to *polyspaceroot*\polyspace\bin, where *polyspaceroot* is the Polyspace installation folder; for instance, C:\Program Files\Polyspace\R2020b. Open the Polyspace user interface using the `polyspace` executable and create a new project.

See “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Code Prover).

- 2 Specify the analysis options on the **Configuration** pane in the Polyspace project. To open this pane, in the project browser, click the configuration node in your Polyspace project.

See “Specify Polyspace Analysis Options” (Polyspace Code Prover).

- 3 Run the analysis. Based on compilation errors and analysis results, modify options as needed.

See “Run Polyspace Analysis on Desktop” (Polyspace Code Prover).

- 4 Once your analysis options are set, generate a script from the project (.psprj file).

To generate a script from the demo project, `Bug_Finder_Example`:

- a Load the project. Select **Help > Examples > Bug_Finder_Example.psprj**. A copy of this project is loaded in the `Examples` folder in your default workspace. To find the project location, place your cursor on the project name in the **Project Browser** pane.
- b Navigate to the project location and enter:

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

To generate Code Prover scripts, use the same command without the `-bug-finder` option.

If a project has more than one module (with more than one configuration in each module), the options from the currently active configuration in the currently active module will be extracted in the script.

These files are generated for scripting the analysis:

- `source_command.txt`: Lists source files. This file can be provided as argument to the `-sources-list-file` option.
- `options_command.txt`: Lists analysis options. This file can be provided as argument to the `-options-file` option.
- `launchingCommand.bat` or `launchingCommand.sh`, depending on your operating system. The file uses the `polyspace-bug-finder` or `polyspace-code-prover` executable to run the analysis. The analysis runs on the source files listed in `source_command.txt` and uses the options listed in `options_command.txt`.

Run Analysis with Generated Scripts

After configuring your analysis and generating scripts, you can use the generated files to automate the subsequent analysis. You can automate the subsequent analysis using either the desktop or server products.

To automate a Bug Finder analysis with the desktop product, Polyspace Bug Finder:

- 1 Generate scripts as mentioned in the previous section.
- 2 Execute the script `launchingCommand.bat` or `launchingCommand.sh` at periodic intervals or based on predefined triggers.

To automate a Bug Finder analysis with the server product, Polyspace Bug Finder Server:

- 1 After specifying options in the user interface and before generating scripts, move the Polyspace project (`.psprj` file) to the server where the server product is running.
- 2 Generate scripts as mentioned in the previous section.

The scripts refer to the server product executable instead of the desktop products.

- 3 Execute the script `launchingCommand.bat` or `launchingCommand.sh` at periodic intervals or based on predefined triggers.

Alternatively, you can modify the script generated for the desktop product so that the server product is executed. The script refers to the path to a desktop product executable, for instance:

```
"C:\Program Files\Polyspace\R2020b\polyspace\bin\polyspace-code-prover.exe"
```

Replace this with the path to a server product executable, for instance:

```
"C:\Program Files\Polyspace Server\R2020b\polyspace\bin\
  polyspace-code-prover-server.exe"
```

Sometimes, you might want to override some of the options in the options file. For instance, the option to specify a results folder is hardcoded in the script. You can remove this option or override it when launching the scripts:

```
launchingCommand -results-dir newResultsFolder
```

where *newResultsFolder* is the new results folder. This folder can even be dynamically generated for each run.

If you override multiple options in `options_command.txt`, you can save the overrides in a second options file. Modify the script `launchingCommand.bat` or `launchingCommand.sh` so that both options files are used. The script uses the option `-options-file` to use an options file, for instance:

```
-options-file options_command.txt
```

If you place your option overrides in a second options file `overrides.txt`, modify the script to append a second `-options-file` option:

```
-options-file options_command.txt -options-file overrides.txt
```

See Also

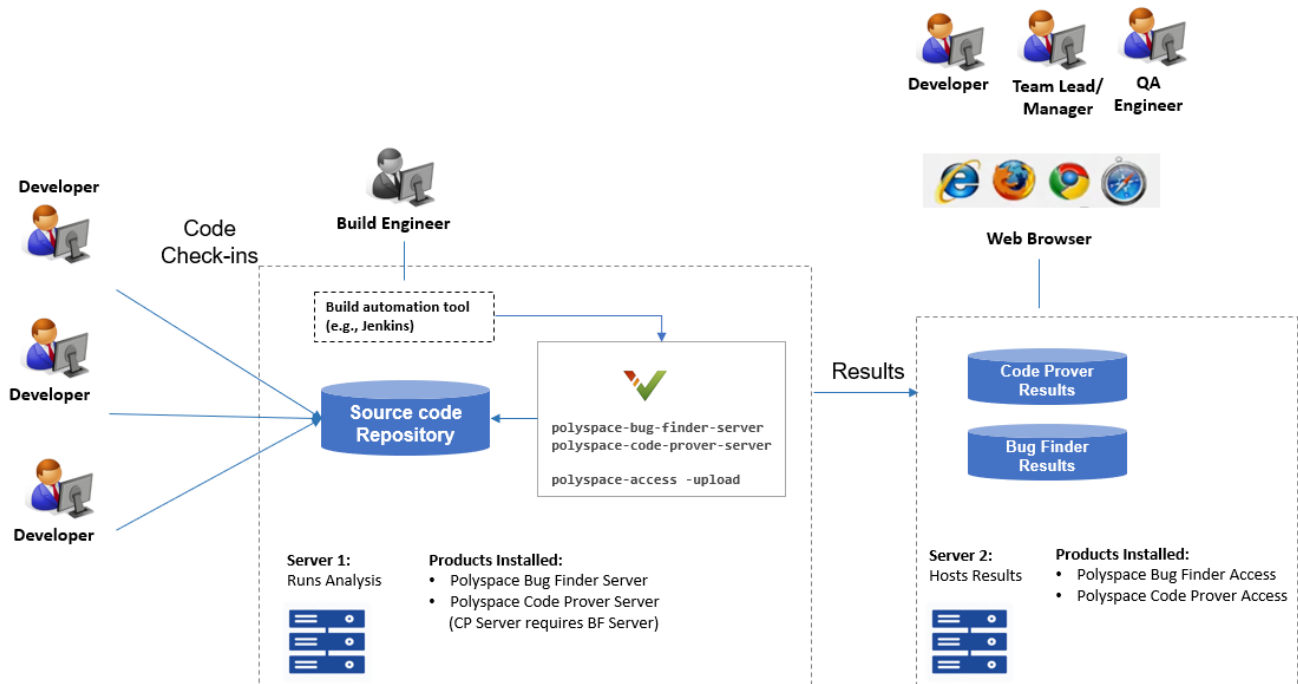
`-generate-launching-script-for`

Related Examples

- “Run Polyspace Code Prover on Server and Upload Results to Web Interface”
- “Prepare Scripts for Polyspace Analysis” on page 1-2

Sample Scripts for Polyspace Analysis with Jenkins

In a continuous integration process, developers submit code to a shared repository. An automated build system using a tool such as Jenkins builds and tests each submission at regular intervals or based on predefined triggers and integrates the code. You can run a Polyspace analysis as part of this process.



Note:

- Depending on the specifications, the same computer can serve as both Server 1 and Server 2.
- Though a server hosts the components for Polyspace web interface, each reviewer requires a Polyspace (BF/CP) Access license to login to the interface.

This topic provides sample Shell scripts that run a Polyspace analysis using Polyspace Bug Finder Server and upload the results for review in the Polyspace Access web interface. The script also sends e-mail notifications to potential reviewers. Notified reviewers can login to the Polyspace Access web interface (if they have a Polyspace Bug Finder Access™ license) and review the results.

Extending Sample Scripts to Your Development Process

The scripts are written for a specific development toolchain but can be easily extended to the processes used in your project, team or organization. The scripts are also meant to be run in a Jenkins freestyle project. If you are using Jenkins Pipelines, see “Sample Jenkins Pipeline Scripts for Polyspace Analysis” on page 1-29.

In particular, the scripts:

- *Run on Linux only.*

The scripts use some Linux-specific commands such as `export`. However, these commands are not an integral part of the Polyspace workflow. If you write Windows scripts (`.bat` files), use the equivalent Windows commands instead.

- *Work only with Jenkins after you install the Polyspace plugin.*

The scripts are designed for the Jenkins plugin in these two ways:

- The scripts uses helper functions `$ps_helper` and `$ps_helper_access` for simpler scripting. The helper functions export Polyspace results for e-mail attachments and use command-line utilities to filter the results.

These helper functions are available only with the Jenkins plugin. However, the underlying commands come with a Polyspace Bug Finder Server installation. On build automation tools other than Jenkins, you can create these helper functions using the `polyspace-report-generator` command or `polyspace-access` command (with the `-export` option). See “Send Email Notifications with Polyspace Code Prover Results”.

If you perform a distributed build in Jenkins, the plugin must be installed in the same folder in the same operating system on both the master node and the agent node executing the Polyspace analysis. Otherwise, you cannot use the helper functions.

- The scripts create text files for e-mail attachments and mail subjects and bodies for personalized e-mails. If you install the Polyspace plugin in Jenkins, an extension of an e-mail plugin is available for use in your Jenkins projects. The e-mail plugin allows you to easily send the personalized e-mails with the previously created subjects, bodies and attachments. Without the Polyspace plugin, you have to find an alternative way to send the e-mails.
- *Run a Bug Finder analysis.*

The scripts run Bug Finder on the demo example `Bug_Finder_Example`. If you install the product Polyspace Bug Finder Server, the folder containing the demo example is `polyspaceserverroot/polyspace/examples/cxx/Bug_Finder_Example`. Here, `polyspaceserverroot` is the installation folder for Polyspace Server products, for instance, `/usr/local/Polyspace Server/R2019a/`.

You can easily adapt the script to run Code Prover. Replace `polyspace-bug-finder-server` with `polyspace-code-prover-server`. You can use the demo example `Code_Prover_Example` specifically meant for Code Prover.

Prerequisites

To run a Polyspace analysis on a server and review the results in the Polyspace Access web interface, you must perform a one-time setup.

- To run the analysis, you must install one instance of the Polyspace Server product.
- To upload results, you must set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you (and each developer reviewing the results) must have one Polyspace license.

Similar requirements apply to a Polyspace Code Prover analysis on a server.

See “Install Polyspace Server and Access Products”.

To install the Polyspace plugin, in the Jenkins interface, select **Manage Jenkins** on the left. Select **Manage Plugin**. Search for the Polyspace plugin and then download and install the plugin.

Set Up Polyspace Plugin in Jenkins

The following steps outline how to set up a Polyspace analysis in Jenkins after installing the Polyspace plugin. Note that the steps refer to Jenkins version 2.150.1. The steps in your Jenkins version and your Polyspace plugin installation might be slightly different.

If you use a different build automation tool, you can perform similar setup steps.

Specify Paths to Polyspace Commands and Server Details for Polyspace Access Web Interface

Specify the full paths of the folder containing the Polyspace commands and host name and port number of the server hosting the Polyspace Access web interface. After you specify the paths, in your scripts, you do not have to use the full paths to the commands or the server details for uploading results.

- 1 In the Jenkins interface, select **Manage Jenkins** on the left. Select **Configure System**.
- 2 In the **Polyspace** section, specify the following:
 - Paths to Polyspace commands.

The path refers to *polyspaceserverroot*/polyspace/bin, where *polyspaceserverroot* is the installation folder for Polyspace Server products, for instance, /usr/local/Polyspace Server/R2019a/.

Polyspace Bin	
Name	Server_install
Binary Path	/usr/local/Polyspace Server/R2019a/polyspace/bin
	Correct Configuration
	Delete

- The host name, port number and protocol (http or https) used by the server hosting the Polyspace Access web interface.



Polyspace Access

Name Polyspace_Access

Protocol https

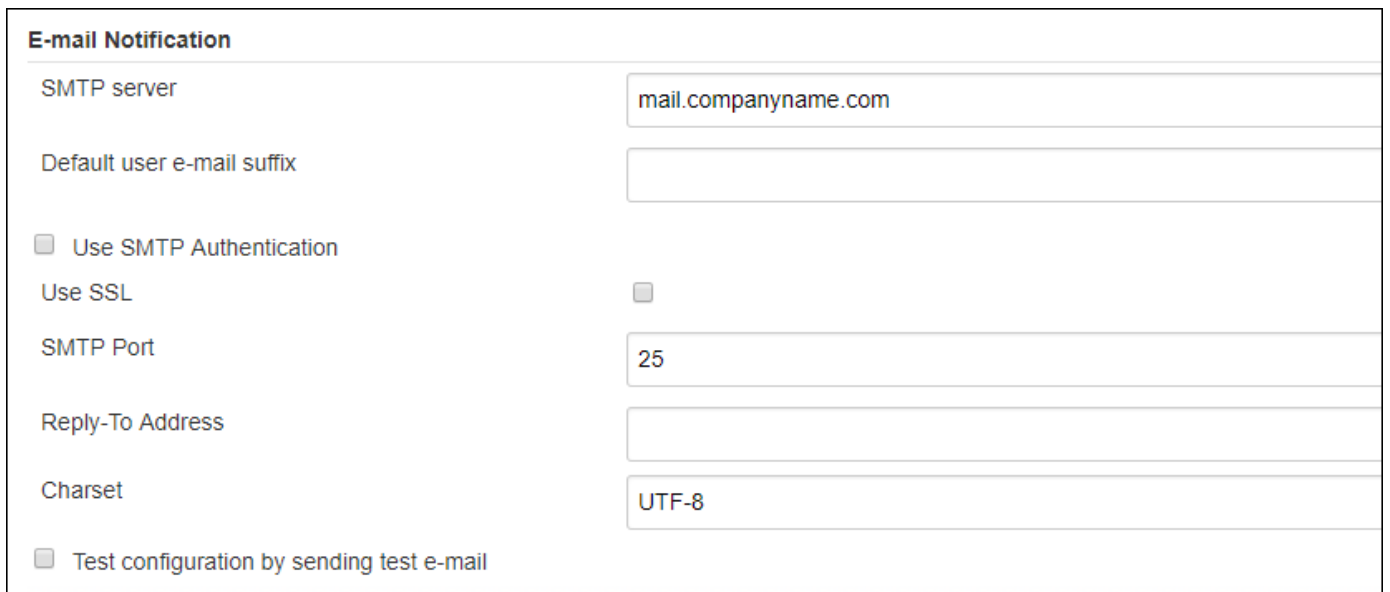
Host doc-server

Port 9443

Delete

The **Name** field allows you to define a convenient shorthand that you use later in Jenkins projects.

- 3 In the **E-mail Notification** section, specify your company's SMTP server (and other details needed for sending e-mails).



E-mail Notification

SMTP server mail.companyname.com

Default user e-mail suffix

Use SMTP Authentication

Use SSL

SMTP Port 25

Reply-To Address

Charset UTF-8

Test configuration by sending test e-mail

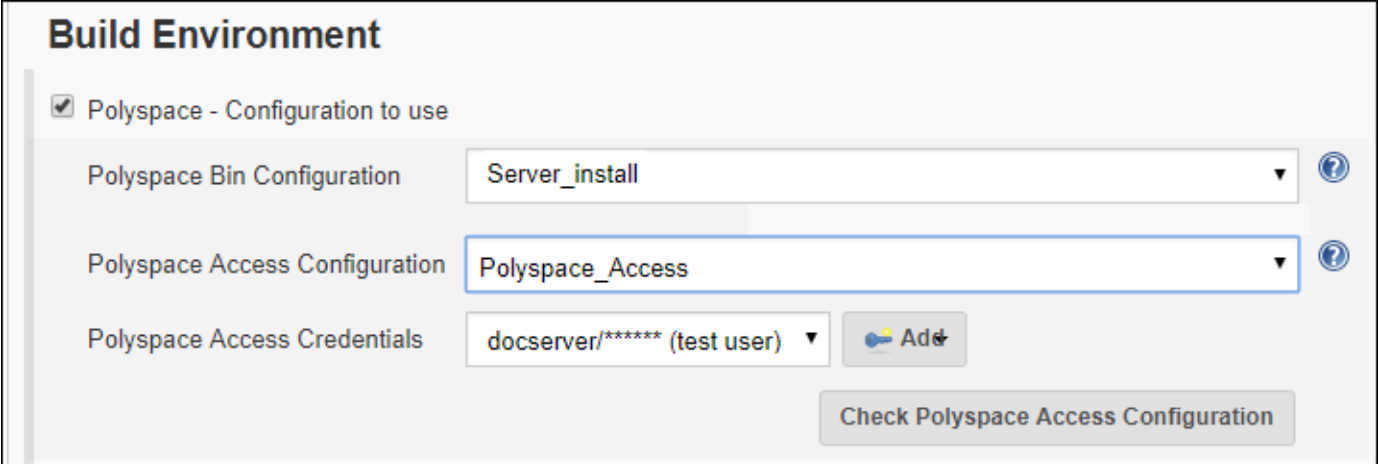
Create Jenkins Project for Running Polyspace

When you create a Jenkins project (for instance, a Freestyle project), you can refer to the Polyspace paths by the global shorthands that you defined earlier.

To create a Jenkins project for running Polyspace:

- 1 In the Jenkins interface, select **New Item** on the left. Select **Freestyle Project**.
- 2 In the **Build Environment** section of the project, enter the two shorthand names you defined earlier:
 - The name for the path to the folder containing the Polyspace commands
 - The name for the details of the server hosting the Polyspace Access web interface.

Also, enter a login and password that can be used to upload to the Polyspace Access web interface. The login and password must be associated with a Polyspace Bug Finder Access license.



The screenshot shows the 'Build Environment' section of the Jenkins configuration interface. It features a checked checkbox for 'Polyspace - Configuration to use'. Below this, there are three configuration fields: 'Polyspace Bin Configuration' with a dropdown menu set to 'Server_install', 'Polyspace Access Configuration' with a dropdown menu set to 'Polyspace_Access', and 'Polyspace Access Credentials' with a dropdown menu set to 'docserver/***** (test user)'. To the right of the credentials dropdown is an 'Add' button. At the bottom right of the configuration area is a 'Check Polyspace Access Configuration' button. Each dropdown menu has a help icon (question mark in a circle) to its right.

- 3 In the **Build** section of the project, you can enter scripts that use the Polyspace commands and details of the server hosting the Polyspace Access web interface.

Build

Execute shell

```
Command set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example_2
export PARENT_PROJECT=testProject
rm -rf Notification && mkdir -p Notification

build_cmd="gcc -c sources/*.c"
polyspace-configure \
  -allow-overwrite \
  -allow-build-error \
  -prog $PROG \
  -author jenkins \
  -output-options-file $PROG.psopts \
  $build_cmd

polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT
```

The scripts run a Polyspace analysis and upload results to the Polyspace Access web interface.

- 4 In the **Post-build Actions** section of the project, configure e-mail addresses and attachments to be sent after the analysis.

Post-build Actions

Polyspace Notification X

Send to Recipients ?

Recipients

File to attach

Mail Subject

Mail Body

Script to Run Bug Finder, Upload Results and Send Common Notification

This script runs a Bug Finder analysis, uploads the results and exports defects with high impact for a common notification email to all recipients.

The script assumes that the current folder contains a folder `sources` with `.c` files. Otherwise modify the line `gcc -c sources/*.c` with the full path to the sources.

```

set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export PARENT_PROJECT=/public/BugFinderExample_PRS_01

# =====
# Trace build command and create an options file

build_cmd="gcc -c sources/*.c"
polyspace-configure \
    -allow-overwrite \
    -allow-build-error \
    -prog $PROG \
    -author jenkins \
    -output-options-file $PROG.psopts \
    $build_cmd

# =====
# Run Bug Finder on the options file

polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT

# =====
# Upload results to Polyspace Access web interface

$ps_helper_access -create-project $PARENT_PROJECT
$ps_helper_access \
    -upload $RESULT \
    -parent-project $PARENT_PROJECT \
    -project $PROG

# =====
# Export results filtered for defects with "High" impact

$ps_helper_access \
    -export $PARENT_PROJECT/$PROG \
    -output Results_All.tsv \
    -defects High

# =====
# Finalize Jenkins status

exit 0

```

After the script is run, you can create a post-build action to send an e-mail to all recipients with the exported file `Results_All.tsv`.

Post-build Actions

Polyspace Notification X

Send to Recipients ?

Recipients

File to attach

Mail Subject

Mail Body

In this script, `$ps_helper_access` is a shorthand for the `polyspace-access` command with the options specifying host name, port, login and encrypted password included. The other `polyspace-access` options are explicitly written in the script.

Script to Run Bug Finder, Upload Results and Send Personalized Notification

This script runs the previous Bug Finder analysis and uploads the results. However, the script differs from the previous script in these ways:

- The script uses a `run_command` function that prints a message when running a command. The function helps determine from the console output which part of the script is running.
- When exporting the results, the script creates a separate results file for different owners.
 - A master file `Results_All.tsv` contains all results. This file is sent in e-mail attachment to a manager. The manager email is configured in the post-build step.

If the file contains more than 10 defects, the build status is considered as a failure. The script sends a status `UNSTABLE` in the e-mail notification.

- The results file `Results_Users_userA.tsv` exported for `userA` contains defects from the group Programming and with impact High.

This result file is sent in e-mail attachment to `userA`.

- The results file `Results_Users_userB.tsv` exported for `userB` contains defects from the function `bug_memstdlib()`.

This result file is sent in e-mail attachment to userB.

- A separate mail subject is created for the manager in the file `mailsubject_manager.txt` and for users `userA` and `userB` in the files `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` respectively.

A mail body is created for the email to the manager in the file `mailbody_manager.txt`.

The script:

- Assumes that the current folder contains a folder `sources` with `.c` files.

Otherwise, modify the line `gcc -c sources/*.c` with the full path to the sources.

- Assumes users named `userA` and `userB`. In particular, the email addresses `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier) must be real e-mail addresses.

Replace the names with real user names.

```
set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export REPORT=Results_List.tsv

# =====
# Define function to print message while running command
run_command()
{
# $1 is a message
# $2 $3 ... is the command to dump and to run
message=$1
shift
cat >> mailbody_manager.txt << EOF
$(date): $message

EOF
"$@"
}

# =====
# Initialize mail body
cat > mailbody_manager.txt << EOF
Dear Manager(s)

Here is the report of the Jenkins Job ${JOB_NAME} #${BUILD_NUMBER}
It contains all Red Defect found in Bug Finder Example project

EOF

# =====
# Trace build command and create options file

build_cmd="gcc -c sources/*.c"
run_command "Tracing build command", \
            polyspace-configure \
            -allow-overwrite \
            -allow-build-error \
            -prog $PROG \
            -author jenkins \
            -output-options-file $PROG.psopts \
            $build_cmd

# =====
# Run Bug Finder on the options file

run_command "Running Bug finder" \
            polyspace-bug-finder-server -options-file $PROG.psopts \
            -results-dir $RESULT

# =====
# Upload results to Polyspace Access web interface

run_command "Creating Project $PARENT_PROJECT" \
```

```

$ps_helper_access -create-project $PARENT_PROJECT

run_command "Uploading on $PARENT_PROJECT/$PROG" \
  $ps_helper_access \
    -upload $RESULT \
    -parent-project $PARENT_PROJECT \
    -project $PROG \
    -output upload.output
PROJECT_RUNID=$(($ps_helper prs_print_runid upload.output)
PROJECT_URL=$(($ps_helper prs_print_projecturl upload.output $POLYSPACE_ACCESS_URL)

# =====
# Export report

run_command "Exporting report from $PARENT_PROJECT/$PROG" \
  $ps_helper_access \
    -export $PROJECT_RUNID \
    -output $REPORT \
    -defects High

# =====
# Filter Reports

run_command "Filtering reports for defects" \
  $ps_helper report_filter \
    $REPORT \
    Results_All.tsv \
    Family Defect \

# =====
# Filter Reports for userA and userB

run_command "Filtering Reports for userA based on Group and Information" \
  $ps_helper report_filter \
    $REPORT \
    Results_Users.tsv \
    userA \
    Group Programming \
    Information "Impact: High"
run_command "Filtering Reports for userB based on Function" \
  $ps_helper report_filter \
    $REPORT \
    Results_Users.tsv \
    userB \
    Function "bug_memstdlib()"

# =====
# Update Jenkins status
# Jenkins build status is unstable when there are more than 10 Defects

BUILD_STATUS=$(($ps_helper report_status Results_All.tsv 10)

# =====
# Update mail body and mail subject

```

```
NB_FINDINGS_ALL=$(($ps_helper report_count_findings Results_All.tsv)
NB_FINDINGS_USERA=$(($ps_helper report_count_findings Results_Users_userA.tsv)
NB_FINDINGS_USERB=$(($ps_helper report_count_findings Results_Users_userB.tsv)
cat >> mailbody_manager.txt << EOF

Number of defects: $NB_FINDINGS_ALL
Number of findings owned by userA: $NB_FINDINGS_USERA
Number of findings owned by userB: $NB_FINDINGS_USERB

All results are uploaded in: $PROJECT_URL

Build Status: $BUILD_STATUS

EOF

cat >> mailsubject_manager.txt << EOF
Polyspace run completed with status $BUILD_STATUS and $NB_FINDINGS_ALL findings
EOF

for user in userA userB
do
echo "$user - $($ps_helper report_count_findings Results_Users_$user.tsv) findings"\
  > mailsubject_user_$user.txt
done

# =====
# Exit with correct build status

[ "$BUILD_STATUS" != "SUCCESS" ] && exit 129
exit 0
```

After the script is run, you can create a post-build action to send an e-mail to a manager with the exported file `Results_All.tsv`. Specify the e-mail address in the **Recipients** field, the email subject in the **Mail Subject** field and the email body in the **Mail Body** field.

In addition, a separate e-mail is sent to `userA` and `userB` with the files `Results_Users_userA.tsv` and `Results_Users_userB.tsv` in attachment (and the content of `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` as mail subjects). The e-mail addresses are `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier).

Post-build Actions

Polyspace Notification
X

Send to Recipients
 ?

Recipients

File to attach

Mail Subject

Mail Body

Send to Owners
 ?

Query Base Name

Mail Subject Base Name

Mail Body Base Name

Unique recipients - Debug only

Add post-build action ▼

The script uses the helper function `$ps_helper` to filter the results based on group, impact and function. The helper function uses command-line utilities to filter the master file for results and perform actions such as create a separate results file for each owner. The function takes these actions as arguments:

- `report_filter`: Filters results from exported text file based on contents of the text file.

For instance:

```
$ps_helper report_filter \
    Results_List.tsv \
    Results_Users.tsv \
    userA \
    Group Programming \
    Information "Impact: High"
```

reads the file `Results_List.tsv` and writes to the file `Results_Users_userA.tsv`. The text file `Results_List.tsv` contains columns for `Group` and `Information`. Only those rows where the `Group` column contains `Programming` and the `Information` column contains `Impact: High` are written to the file `Results_Users_userA.tsv`.

- `report_status`: Returns `UNSTABLE` or `SUCCESS` based on the number of results in a file.

For instance:

```
BUILD_STATUS=$(ps_helper report_status Results_All.tsv 10)
```

returns `UNSTABLE` if the file `Results_All.tsv` contains more than 10 results (10 rows).

- `report_count_findings`: Reports number of results in a file.

For instance:

```
NB_FINDINGS_ALL=$(ps_helper report_count_findings Results_All.tsv)
```

returns the number of results (rows) in the file `Results_All.tsv`.

- `prs_print_projecturl`: Uses the host name and port number to create the URL of the Polyspace Access web interface.

For instance:

```
PROJECT_URL=$(ps_helper prs_print_projecturl Results_All.tsv $POLYSPACE_ACCESS_URL)
```

reads the file `Results_All.tsv` (exported by the `polyspace-access` command) and extracts the URL of the Polyspace Access web interface in `$POLYSPACE_ACCESS_URL` and the URL of the current project in `$PROJECT_URL`.

See Also

`polyspace-access` | `polyspace-bug-finder-server` | `polyspace-code-prover-server` | `polyspace-configure` | `polyspace-report-generator`

More About

- “Run Polyspace Code Prover on Server and Upload Results to Web Interface”
- “Send Email Notifications with Polyspace Code Prover Results”
- “Sample Jenkins Pipeline Scripts for Polyspace Analysis” on page 1-29
- “Offload Polyspace Analysis from Continuous Integration Server to Another Server” on page 1-6

Sample Jenkins Pipeline Scripts for Polyspace Analysis

Jenkins Pipelines enable automating the workflow of a continuous delivery pipeline through scripts in Jenkins. You can write Pipeline scripts that build projects, run test suites and perform all necessary checks before your code is ready for shipping. You can check in these scripts as part of a version control system and subject them to the same review and versioning as the code itself.

You can run a Polyspace analysis in a Jenkins Pipeline script. If you are not using Freestyle Projects instead of Pipelines in Jenkins, use the Polyspace plugin for scripting conveniences. See “Sample Scripts for Polyspace Analysis with Jenkins” on page 1-15. If you are using Pipelines, modify the script provided to run a Polyspace analysis.

Prerequisites

To run a Polyspace analysis on a server and review the results in the Polyspace Access web interface, you must perform a one-time setup.

- To run the analysis, you must install one instance of the Polyspace Server product.
- To upload results, you must set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you and each developer reviewing the results must have one Polyspace license.

See “Install Polyspace Server and Access Products”.

Run Polyspace Analysis in Stages in a Pipeline Script

To create a Jenkins Pipeline script:

- 1 In the Jenkins interface, select **New Item** on the left. Select **Pipeline**.
- 2 In the **Pipeline** section of the project, select Pipeline script for **Definition**. Enter this script.

The parts in bold indicate places where you have to modify the script for your source code and Polyspace installation.

The script is not available in the PDF documentation. Search for Polyspace Jenkins Pipelines in the MathWorks® online documentation and copy the script from the online version of this page.

When you build this project, you can see the various stages of the analysis like this:

Prepare	Checkout	Configure	Analyze	Upload	Notification
1s	1s	14s	4min 22s	1min 32s	369ms
1s	1s	14s	4min 22s	1min 32s	369ms

This script can be part of a larger script that you save in a Jenkinsfile and commit to your version control system. See [Using a Jenkinsfile](#).

You can modify the script as needed:

- The script runs each step of the Polyspace analysis workflow in a separate **stage** section. You can combine several steps together in one **stage**.
- The script runs Linux Shell commands by using the `sh` directive. You can run Windows commands by using the `bat` directive instead.
- The script uses data from the Credentials plugin to extract user name and password. If you save credentials in some other form, you can replace the `withCredentials` command that binds user credentials to variables.
- The script builds source code using a makefile on a Git sandbox with this make command:

```
make -C $git_sandbox
```

If you use a different build command, you can replace this line with your build command.

For more information on the Pipeline-specific syntax in this script, see:

- [Pipeline Syntax](#): Describes `node`, `stage`, `label`.
- [Pipeline Steps Reference](#): Describes `sh`, `mail`.
- [Credentials Binding Plugin](#): Describes `withCredentials`.

For more information on the Polyspace commands in this script, see:

- `polyspace-configure`
- `polyspace-bug-finder-server` (also `polyspace-code-prover-server`)
- `polyspace-access`

See Also

“Sample Scripts for Polyspace Analysis with Jenkins” on page 1-15

Run Polyspace Analysis on Generated Code by Using Packaged Options Files

When you start a Polyspace analysis directly from the Simulink toolstrip, the analysis takes the model-specific context, such as design ranges, into consideration. When running a Polyspace analysis without access to Simulink, you must specify the model-specific information by using options files. Instead of authoring these options files, use the options files generated and packaged by the function `polyspacePackNGo`.

Preserving the Simulink model context information when running a Polyspace analysis can be useful in various situations. For instance:

- **Distributed workflow:** A Simulink user generates code from a model and sends the code to another development environment. In this environment, a Polyspace user, who might not have Simulink, runs a separate analysis of the generated code. By using the packaged options files, the design ranges and other model-specific information is preserved in the Polyspace analysis.
- **Analysis options not available in Simulink:** Some Polyspace analysis options are available only when the Polyspace analysis is run separately from Simulink. Use packaged options files to run a separate Polyspace analysis while preserving the model-specific information. For instance, analyze concurrent threads in generated code by running a Polyspace analysis in the generated code by using the packaged options files.

You must have Simulink to run the function `polyspacePackNGo`. You do not need Polyspace to generate the options files from a Simulink model. The `polyspacePackNGo` function supports code generated by Embedded Coder® and TargetLink®.

Generate and Package Polyspace Options Files

To generate and package Polyspace options file for analyzing code generated from a Simulink model, use `polyspacePackNGo`.

- 1 In the Simulink Editor, open the Configuration Parameters dialog box and configure the model for code generation.
- 2 To configure the model for compatibility with Polyspace, select `ert.tlc` as the **System target file**.
- 3 To enable generating a code archive, select the option **Package code and artifacts**. Optionally, provide a name for the options package in the field **Zip file name**. If your code contains a custom code block, select **Use the same custom code settings as Simulation target** in the **Code Generation> Custom Code** pane.

Alternatively, in the MATLAB Command Window, enter:

```
% Configure the Simulink model mdlName for code generation
configSet = getActiveConfigSet(mdlName);
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', 'CodeArchive.zip');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
set_param(configSet, 'RTWUseSimCustomCode', 'on');
```

- 4 Generate the code archive.

- To generate an archive of standalone generated code from the top model, use the function `rtwbuild`.

- To generate code as a model reference, use the function `slbuild`. After generating code as model reference, create the code archive by using the function `packNGo`.
- Alternatively, you can use `TargetLink` to generate the code. Create the code archive by archiving the generated code into a zip file.

- 5** To generate and package the Polyspace option files, in the MATLAB Command Window ,use the `polyspacePackNGo` function :

```
zipFile = polyspacePackNGo mdlName);
```

See “Generate and Package Polyspace Options Files”.

If you use `TargetLink` to generate code, then use the `TargetLink` subsystem name as the input argument to `polyspacepackngo`.

- 6** Optionally, you can use a `pslinkoptions` object as a second argument to modify the default options for the Polyspace analysis. Create a `pslinkoptions` object containing the additional options and specify the object when creating the archive:

```
psOpt = pslinkoptions(mdlName);  
psOpt.InputRangeMode = 'FullRange';  
psOpt.ParamRangeMode = 'DesignMinMax';  
zipFile = polyspacePackNGo(mdlName,psOpt);
```

See “Package Polyspace Options Files That Have Specific Polyspace Analysis Options”.

- 7** Use the optional third argument to specify whether to generate and package Polyspace options files for code generated as a model reference. Suppose you generated code as a model reference by using the `slbuild` function. To generate and package Polyspace options for the code, at the MATLAB Command Window, enter:

```
zipFile = polyspacePackNGo(mdlName,psOpt,true);
```

See “Package Polyspace Options Files for Code Generated as a Model Reference”.

The function `polyspacepackngo` returns the full path to the archive containing the options files. The files are located in the `polyspace` folder within the archived folder hierarchy. The content of the `polyspace` folder depends on the inputs of `polyspacePackNGo` function.

- If you do not specify the optional second and third arguments, then the folder `polyspace` contains these options files in a flat hierarchy:
 - `optionsFile.txt`: This file specifies the source files, the include files, data range specifications, and analysis options required for analyzing the generated code by using Polyspace. If your code contains custom C code, then this file specifies the relative paths of the custom source and header files.
 - `modelName_drs.xml`: This file specifies the design range specification of the model.
 - `linkdata.xml`: This file links the generated code to the components of the model.
- If you specify `psOpts.ModelbyModelRef = true`, then corresponding options files are generated for all referenced models. These options files are stored in separate folders named `polyspace_<referenced model name>` within the code archive. The folder `polyspace` contains the options files for the top model.

Run Polyspace Analysis by Using the Packaged Options Files

Once the code archive and the Polyspace option files are generated, you can use the archive to run a Polyspace analysis on the generated code in a different development environment without Simulink.

- 1 Unzip the code archive and locate the `polyspace` folder.
- 2 On a Windows or Linux command line, run: `productname -options-file optionsFile.txt -results-dir resultdir`.
 - `productname` corresponds to one of: `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server`, or `polyspace-code-prover-server`.
 - `resultdir` corresponds to the location of the Polyspace results. This argument is optional.

If the file `linkdata.xml` is not there, use the option **Code Generator Support** in Polyspace User Interface to specify which comments in the code act as links to the Simulink model. In the Polyspace User Interface, select **Tools > Preferences** and locate the **Miscellaneous** tab. From the context menu **Code comments that act as code-to-model-link**, select the code generator that you used. If you select **User defined**, then specify the comments that act as a code-to-model link by specifying their prefix in the field **Comments beginning with**. For instance, if you specify the prefix as `//Link_to_model`, then Polyspace interprets comments starting with `//Link_to_model` as links to model.

- 3 To review the result, upload it to Polyspace Access and view the results in a web browser. Alternatively, view the result by using the user interface of the Polyspace desktop products.

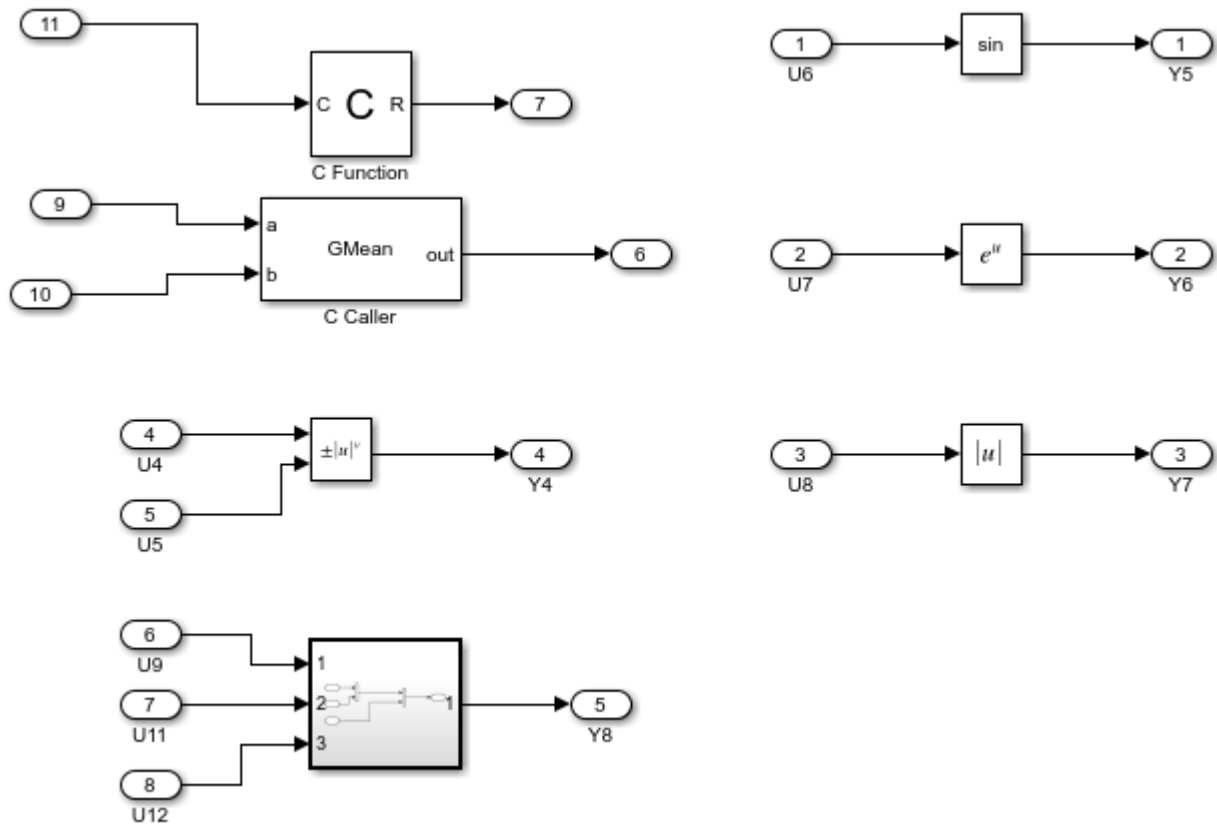
Analyze Code Generated as Standalone Code in a Distributed Workflow

Generate and package Polyspace options files from a Simulink model by using the function `polyspacepackNGo`. Use these options files to run a Polyspace analysis on the generated code that uses model-specific information, such as design range specifications, without requiring Simulink.

Open Model

The model `demo_math_operations` performs various mathematical operations on the model inputs. The model has a C Function block that executes a custom C code. The model also has a C Caller block that calls the C function `GMean`, which is implemented in the source file `GMean.c`. To open the model for code generation and packaging Polyspace options file, search for the current topic in the MATLAB help browser and click the **Open Model** button. Alternatively, in the MATLAB Command Window, paste and run the following code.

```
open_system('demo_math_operations');
```



Configure Model

To configure the model for generating code and packaging Polyspace options files, specify these configuration parameters:

- To create an archive containing the generated code, set 'PackageGeneratedCodeAndArtifacts' to true.
- Specify a name for the code archive. For instance, set the name to `genCodeArchive.zip`.
- To use the custom code setting specified in **Simulation Target** during code generation, set 'RTWUseSimCustomCode' to 'on'.
- To make the model and the generated code compatible with Polyspace, set `ert.tlc` as the system target file. See “Recommended Model Configuration Parameters for Polyspace Analysis” (Polyspace Code Prover).

In Command Window or Editor, enter these parameter configurations:

```
configSet = getActiveConfigSet('demo_math_operations');
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', 'genCodeArchive.zip');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
set_param(configSet, 'RTWUseSimCustomCode', 'on')
```

Generate Code Archive

Specify a folder for storing the generated code. To start code generation, in the Command Window or in the Editor, enter:

```
codegenFolder = 'demo_math_operations_ert_rtw';
if exist(fullfile(pwd,codegenFolder), 'dir') == 0
    rtwbuild('demo_math_operations')
end
```

Because `PackageGeneratedCodeAndArtifacts` is set to `true`, the generated code is packed into the archive `genCodeArchive.zip`.

Generate and Package Polyspace Options File

To generate Polyspace options files for the generated code, in the Command Window or in the Editor, enter:

```
zipFile = polyspacePackNGo('demo_math_operations');
```

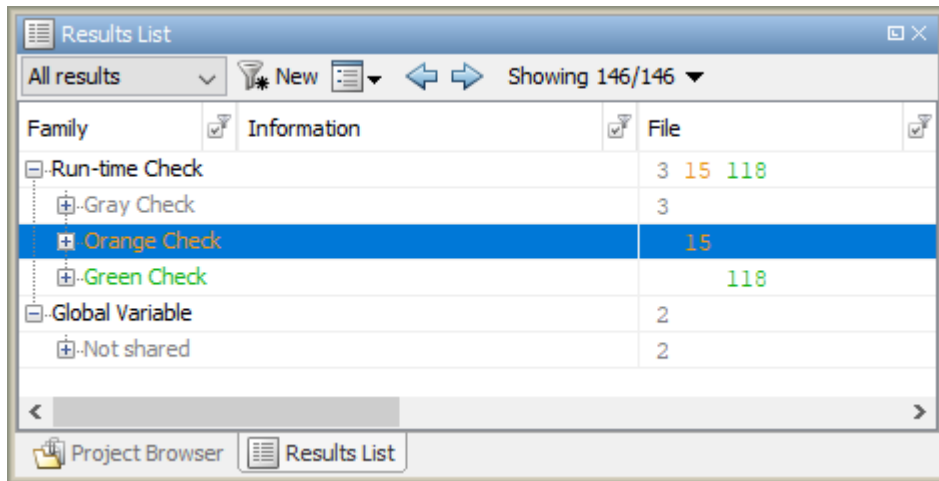
In the archive `genCodeArchive.zip`, find the options files in the folder `<current folder>/polyspace`.

Run Polyspace Analysis by Using the Packaged Options Files

- 1 Unzip the code archive `genCodeArchive.zip` and locate the `<current folder>/polyspace` folder.
- 2 Open a command-line terminal and change your working folder to the `polyspace` subfolder of the unzipped folder by using the `cd` command.
- 3 Start a Polyspace analysis.
 - To run a desktop Polyspace analysis, use either `polyspace-code-prover` or `polyspace-bug-finder`. To run the Polyspace analysis in a server, use either `polyspace-bug-finder-server` or `polyspace-code-prover-server`. Polyspace Bug Finder and Code Prover analyze the code differently. See “Choose Between Polyspace Bug Finder and Polyspace Code Prover”.
 - Specify the file `optionsFile.txt` as the argument to `-options-file`.

To run a Code prover analysis, run this command: `polyspace-code-prover -options-file optionsFile.txt -results-dir Results`.

- 4 Follow the progress of the analysis in the log file that is generated in the `Results` folder.
- 5 To view the results in the desktop user interface, in the command-line interface, enter: `polyspace Results\ps_results.pscp`. The extension of the `ps_results` file changes depending on whether you run a Code Prover analysis or a Bug Finder analysis. The result contains several orange checks.



Alternatively, upload the result to Polyspace Access. See “Upload Results to Polyspace Access” (Polyspace Code Prover Access)

- 6 Address the results. For more information, see “Address Polyspace Results Through Bug Fixes or Justifications” (Polyspace Code Prover Access).

See Also

packNGo | polyspace.Project | rtwbuild | slbuild

More About

- “Integrate Polyspace Server Products with MATLAB and Simulink” on page 4-2

Use Existing Software Development Specifications for Polyspace Analysis

- “Create Polyspace Analysis Configuration from Build Command” on page 2-2
- “polyspace-configure Source Files Selection Syntax” on page 2-4
- “Modularize Polyspace Analysis by Using Build Command” on page 2-6
- “Create Polyspace Analysis Configuration from AUTOSAR Specifications” on page 2-12
- “Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis” on page 2-16
- “Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code” on page 2-19

Create Polyspace Analysis Configuration from Build Command

To run Polyspace on a server during continuous integration, you must configure all analysis options beforehand so that the analysis completes without errors. These options must be updated as necessary to keep up with new code submissions. If you use existing artifacts such as a build command (makefile) to build new code submissions, you can reuse the build command to configure a Polyspace analysis and stay updated with new submissions. With the `polyspace-configure` command, you can monitor the execution of a build command and create an options file for analysis with Polyspace.

This topic shows a simple tutorial illustrating how to create an options file from a build command and use the file for the subsequent analysis. The topic uses a Linux makefile and the GCC compiler, but you can adapt the commands to other operating systems such as Windows and other compilers such as Microsoft® Visual Studio®.

- 1 Cope the demo source files from `polyspaceserverroot\polyspace\examples\cxx\Bug_Finder_Example\sources` to a folder with write permissions. Here, `polyspaceserverroot` is the root installation folder of the Polyspace server products, for instance, `C:\Program Files\Polyspace Server\R2019a`.
- 2 Create a simple makefile that compiles the demo source files. Save the makefile in the same folder as the source files.

For instance, create a file named `makefile` and add this content:

```
CC := gcc
SOURCES := $(wildcard *.c)

all: $(CC) -c $(SOURCES)
```

Check that the makefile builds the source files successfully. Open a command terminal, navigate to the folder (using `cd`) and enter:

```
make
```

The `make` command should complete execution without errors.

- 3 Trace the build command with `polyspace-configure` and create an options file `compile_opts`.

```
polyspace-configure -output-options-file compile_opts make
```

- 4 Create a second options file with additional options. For instance, create a file `run_opts` with this content:

```
-checkers numerical
-report-template BugFinder
-output-format pdf
```

The options run all numerical checkers in Bug Finder and creates a PDF report after analysis using the `BugFinder` template.

- 5 Run a Bug Finder analysis with the two options files: `compile_opts` created from your build command and `run_opts` created manually.

```
polyspace-bug-finder-server -options-file compile_opts -options-file run_opts
```

The analysis should complete without errors. You can open the results in the Polyspace user interface or upload the results to the Polyspace Access web interface (using the `polyspace-access` command).

To run Code Prover instead of Bug Finder, use the `polyspace-code-prover-server` command instead of the `polyspace-bug-finder-server` command.

You can run a similar analysis using MATLAB scripts. Replace `polyspace-bug-finder-server` with the function `polyspaceBugFinderServer` and `polyspace-configure` with the function `polyspaceConfigure`.

See Also

`polyspace-code-prover-server` | `polyspace-configure`

See Also

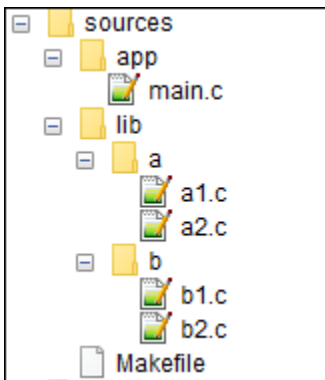
More About

- “Prepare Scripts for Polyspace Analysis” on page 1-2
- “Specify Target Environment and Compiler Behavior” on page 5-2
- “polyspace-configure Source Files Selection Syntax” on page 2-4
- “Modularize Polyspace Analysis by Using Build Command” on page 2-6

polyspace-configure Source Files Selection Syntax

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

This folder structure applies to these examples.



To try these examples, use the demo files in `polyspaceroot\help\toolbox\polyspace_code_prover_server\examples\sources-select`. `polyspaceroot` is the Polyspace installation folder.

Run this command:

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

glob_pattern is the glob pattern that you use to match the paths of the files you want to include or exclude from your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes.

In the table, the examples assume that `sources` is a top-level folder.

Glob Pattern Syntax	Example
No special characters, slashes ('/'), or backslashes ('\'). Pattern matches corresponding files, but not folders.	<code>-include-sources "main.c"</code> matches: <code>/sources/app/main.c</code>
Pattern contains '*' or '?' special characters. '*' matches zero or more characters in file or folder name. '?' matches one character in file or folder name.	<code>-include-sources "b?.c"</code> matches: <code>/sources/lib/b/b1.c</code> <code>/sources/lib/b/b2.c</code>
The matches do not include path separators.	<code>-include-sources "app/*.c"</code> matches: <code>/sources/app/main.c</code>

Glob Pattern Syntax	Example
<p>Pattern starts with slash '/' (UNIX®) or drive letter (Windows).</p> <p>Pattern matches absolute path only.</p>	<p>-include-sources "/a" does not match anything.</p> <p>-include-sources "/sources/app" matches:</p> <pre>/sources/app/main.c</pre>
<p>Pattern ends with a slash (UNIX), backslash (Windows), or '**'.</p> <p>Pattern matches all files under specified folder.</p> <p>'**' is ignored if it is at the start of the pattern.</p>	<p>-include-sources "a/" matches</p> <pre>/sources/lib/a/a1.c /sources/lib/a/a2.c</pre>
<p>Pattern contains '/**/' (UNIX) or '**\' (Windows). Pattern matches zero or more folders in the specified path.</p>	<p>-include-sources "lib/**/?1.c" matches:</p> <pre>/sources/lib/a/a1.c /sources/lib/b/b1.c</pre>
<p>Pattern starts with '.' or '..'.</p> <p>Pattern matches paths relative to the path where you run the command.</p>	<p>If you start polyspace-configure from /sources/lib/a,</p> <p>-include-sources "../lib/**/b?.c" matches:</p> <pre>/sources/lib/b/b1.c /sources/lib/b/b2.c</pre>
<p>Pattern is a UNC path on Windows .</p>	<p>If your files are on server myServer:</p> <pre>\\myServer\sources\lib\b** matches: \\myServer\sources\lib\b\b1.c \\myServer\sources\lib\b\b2.c</pre>

polyspace-configure does not support these glob patterns:

- Absolute paths relative to the current drive on Windows.
For instance, \foo\bar.
- Relative paths to the current folder.
For instance, C:foo\bar.
- Extended length paths in Windows.
For instance, \\?\foo.
- Paths that contain '.' or '..' except at the start of the pattern.
For instance, /foo/bar/./a?.c.
- The '*' character by itself.

Modularize Polyspace Analysis by Using Build Command

To configure the Polyspace analysis, you can reuse the compilation options in your build command such as `make`. First, you trace your build command with `polyspace-configure` (or `polyspaceConfigure` in MATLAB) and create a Polyspace options file. You later specify this options file for the subsequent Polyspace analysis.

If your build command creates several binaries, by default `polyspace-configure` groups the source files for all binaries into one Polyspace options file. If binaries that use the same source files or functions are compiled with different options, you lose this distinction in the subsequent Polyspace analysis. The presence of the same function multiple times can lead to link errors during the Polyspace analysis and sometimes to incorrect results.

This topic shows how to create a separate Polyspace options file for each binary created in your makefile. Suppose that a makefile creates four binaries: two executable (target `cmd1` and `cmd2`) and two shared libraries (target `liba` and `libb`). You can create a separate Polyspace options file for each of these binaries.

To try this example, use the files in `polyspaceroot\help\toolbox\polyspace_code_prover_server\examples\multiple_modules`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2020b` or `C:\Program Files\Polyspace Server\R2020b`.

Build Source Code

Inspect the makefile. The makefile creates four binaries:

```
CC := gcc
LD := ld

LIBA_SOURCES := $(wildcard src/liba/*.c)
LIBB_SOURCES := $(wildcard src/libb/*.c)
CMD1_SOURCES := $(wildcard src/cmd1/*.c)
CMD2_SOURCES := $(wildcard src/cmd2/*.c)
LIBA_OBJ := $(notdir $(LIBA_SOURCES:.c=.o))
LIBB_OBJ := $(notdir $(LIBB_SOURCES:.c=.o))
CMD1_OBJ := $(notdir $(CMD1_SOURCES:.c=.o))
CMD2_OBJ := $(notdir $(CMD2_SOURCES:.c=.o))
LIBB_SOBJ := libb.so
LIBA_SOBJ := liba.so

all: cmd1 cmd2

cmd1: liba libb
    $(CC) -o $@ $(CMD1_SOURCES) $(LIBA_SOBJ) $(LIBB_SOBJ)

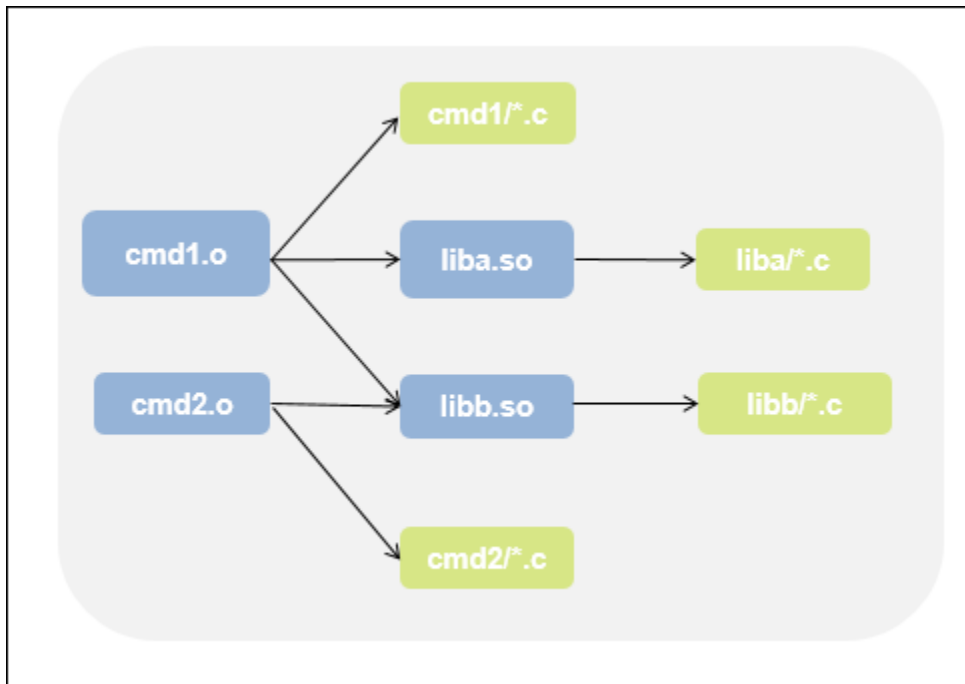
cmd2: libb
    $(CC) -c $(CMD2_SOURCES)
    $(LD) -o $@ $(CMD2_OBJ) $(LIBB_SOBJ)

liba: libb
    $(CC) -fPIC -c $(LIBA_SOURCES)
    $(CC) -shared -o $(LIBA_SOBJ) $(LIBA_OBJ)

libb:
    $(CC) -fPIC -c $(LIBB_SOURCES)
    $(CC) -shared -o $(LIBB_SOBJ) $(LIBB_OBJ)

.PHONY: clean
clean:
    rm *.o
```

The binaries created have the dependencies shown in this figure. For instance, creation of the object `cmd1.o` depends on all `.c` files in the folder `cmd1` and the shared objects `liba.so` and `libb.so`.



Build your source code by using the makefile. Use the `-B` flag to ensure full build.

```
make -B
```

Make sure that the build runs to completion.

Create One Polyspace Options File for Full Build

Trace the build command by using `polyspace-configure`. Use the option `-output-options-file` to create a Polyspace options file `psoptions` from the build command.

```
polyspace-configure -output-options-file psoptions make -B
```

Run Bug Finder or Code Prover by using the previously created options file: Save the analysis results in a `results` subfolder.

```
polyspace-code-prover-server -options-file psoptions -results-dir results
```

You see this link error (warning in Bug Finder):

```
Procedure 'main' multiply defined.
```

The error occurs because the files `cmd1/cmd1_main.c` and `cmd2/cmd2_main.c` both have a `main` function. When you run your build command, the two files are used in separate targets in the makefile. However, `polyspace-configure` by default creates one options file for the full build. The Polyspace options file contains both source files resulting in conflicting definitions of the `main` function.

To verify the cause of the error, open the Polyspace options file `psoptions`. You see these lines that include the files with conflicting definitions of the main function.

```
-sources src/cmd1/cmd1_main.c
-sources src/cmd2/cmd2_main.c
```

Create Options File for Specific Binary in Build Command

To avoid the link error, build the source code for a specific binary when tracing your build command by using `polyspace-configure`.

For instance, build your source code for the binary `cmd1.o`. Specify the makefile target `cmd1` for `make`, which creates this binary.

```
polyspace-configure -output-options-file psoptions make -B cmd1
```

Run Bug Finder or Code Prover by using the previously created options file.

```
polyspace-code-prover-server -options-file psoptions -results-dir results
```

The link error does not occur and the analysis runs to completion. You can open the Polyspace options file `psoptions` and see that only the source files in the `cmd1` subfolder and the files involved in creating the shared objects are included with the `-sources` option. The source files in the `cmd2` subfolder, which are not involved in creating the binary `cmd1.o`, are not included in the Polyspace options file.

Special Considerations for Libraries (Code Prover only)

If you trace the creation of a shared object from libraries, the source files extracted do not contain a main function. In the subsequent Code Prover analysis, you can see an error because of the missing `main`.

Use the Polyspace option `Verify module or library (-main-generator)` to generate a main function. Specify the option in the options file that was created or directly at the command line. See “Verify C Application Without main Function” on page 9-6.

In C++, use these additional options for classes:

- `Class (-class-analyzer)`
- `Functions to call within the specified classes (-class-analyzer-calls)`

Create One Options File Per Binary Created in Build Command

To create an options file for a specific binary created in the build command, you must know the details of your build command. If you are not familiar with the internal details of the build command, you can create a separate Polyspace options file for *every* binary created in the build command. The approach works for binaries that are executables, shared (dynamic) libraries and static libraries.

This approach works only if you use these compilers:

- GNU C or GNU C++
- Microsoft Visual C++

Trace the build command by using `polyspace-configure`. To create a separate options file for each binary, use the option `-module` with `polyspace-configure`.

```
polyspace-configure -module -output-options-path optionsFilesFolder make -B
```

The command creates options files in the folder `optionsFilesFolder`. In the preceding example, the command creates four options files for the four binaries:

- `cmd1.psopts`
- `cmd2.psopts`
- `liba_so.psopts`
- `libb_so.psopts`

You can run Polyspace on the code implementation of a specific binary by using the corresponding options file. For instance, you can run Code Prover on the code implementation of the binary created from the makefile target `cmd1` by using this command:

```
polyspace-code-prover-server -options-file cmd1.psopts -results-dir results
```

For this approach, you do not need to know the details of your build command. However, when you create a separate options file for each binary in this way, each options file contains source files directly involved in the binary and not through shared objects. For instance, the options file `cmd1.psopts` in this example specifies only the source files in the `cmd1` subfolder and not the source files involved in creating the shared objects `liba.so` and `libb.so`. The subsequent analysis by using this options file cannot access functions from the shared objects and uses function stubs instead. In the Code Prover analysis, if you see too many orange checks due to the stubbing, use the approach stated in the section “Create Options File for Specific Binary in Build Command” on page 2-9.

Special Considerations for Libraries (Code Prover only)

If you trace the creation of a shared object from libraries, the source files extracted do not contain a main function. In the subsequent Code Prover analysis, you can see an error because of the missing main.

Use the Polyspace option `Verify module` or `library` (`-main-generator`) to generate a main function. Specify the option in the options file that was created or directly at the command line. See “Verify C Application Without main Function” on page 9-6.

In C++, use these additional options for classes:

- `Class` (`-class-analyzer`)
- Functions to call within the specified classes (`-class-analyzer-calls`)

See Also

`polyspace-code-prover-server` | `polyspace-configure`

More About

- “Create Polyspace Analysis Configuration from Build Command” on page 2-2

Create Polyspace Analysis Configuration from AUTOSAR Specifications

If you use the AUTOSAR methodology for software development, you can create a Polyspace analysis configuration directly from your AUTOSAR specifications. With the product, Polyspace Code Prover Server, you can then run a Code Prover analysis on the code implementation of AUTOSAR Software Components.

To follow the steps in this tutorial, use the demo files in *polyspaceroot\help\toolbox\polyspace_code_prover_server\examples\polyspace_autosar*.

Benefits of Polyspace for AUTOSAR

Polyspace for AUTOSAR runs static program analysis on code implementation of AUTOSAR software components. The analysis looks for possible run-time errors or mismatch with specifications in the AUTOSAR XML (ARXML).

Polyspace for AUTOSAR reads the ARXML specifications that you provide and modularizes the analysis based on the software components in the ARXML specifications. The analysis then checks each module for:

- Mismatch with AUTOSAR specifications: These checks aim to prove that certain functions are implemented or used in accordance with the specifications in the ARXML. The checks apply to runnables (functions provided by the software components) and to the usage of functions supplied by the Run-Time Environment (RTE). See also:
 - AUTOSAR runnable not implemented
 - Invalid result of AUTOSAR runnable implementation
 - Invalid use of AUTOSAR runtime environment function

For instance, if an RTE function argument has a value outside the constrained range defined in the ARXML, the analysis flags a possible issue.

- Run-time errors: These checks aim to prove the absence of certain types of run-time errors in the bodies of the runnables (for instance, overflow). The proof uses the specifications in the ARXML to determine precise ranges for runnable arguments and RTE function return values and output arguments. For instance, the proof considers only those values of runnable arguments that are specified in their AUTOSAR data types.

Run Polyspace on AUTOSAR Code

Run the `polyspace-autosar` command with paths to your ARXML and source code folder. The command parses the ARXML and source files, creates a Polyspace project and analyzes all modules in the project for run-time errors or violation of data constraints in the ARXML.

In the first run, specify the path to your ARXML and source files explicitly. In later runs, specify the file `psar_project.xhtml` created in the previous run. The analysis detects changes in the ARXML and source files since the last run and reanalyzes only those modules where the software component

implementation changed. If the ARXML specification changed since the previous analysis, the new analysis reanalyzes all modules.

For instance, you can run these commands in a `.bat` script. In the first run, this script looks for the ARXML specifications in a folder `arxml` in the current folder, and C source files in a folder `code`. The results are stored in a folder `polyspace` in the current folder. In later runs, the analysis reuses the result from the previous run through the file `psar_project.xhtml` and updates the results only for the software components modified since the last run.

```

echo off
set POLYSPACE_AUTOSAR_PATH=C:\Program Files\Polyspace Server\R2019a\polyspace\bin

IF NOT EXIST polyspace\psar_project.xhtml (
"%POLYSPACE_AUTOSAR_PATH%\polyspace-autosar" -create-project polyspace \
      -arxml-dir arxml -sources-dir code
) ELSE (
"%POLYSPACE_AUTOSAR_PATH%\polyspace-autosar" \
      -update-project polyspace\psar_project.xhtml
)
Pause

```

You can also run Code Prover on code implementation of AUTOSAR software components with MATLAB scripts. See `polyspaceAutosar`.

Upload Results to Polyspace Access Web Interface

For each Software Component behavior, the Code Prover analysis produces an individual result file (with extension `.pscp`). The path to the results file is determined by the fully qualified name of the Software Component. For instance:

- A Software Component behavior with full name `pkg.tst002.swc001.bhv001` has results stored in the file `ps_results.pscp` in the subfolder `AUTOSAR\pkg\tst002\swc001\bhv001\verification\` of the results folder.
- A Software Component behavior with full name `pkg.tst002.swc002.bhv` has results stored in the file `ps_results.pscp` in the subfolder `AUTOSAR\pkg\tst002\swc002\bhv\verification\` of the results folder.

To upload all results, use the `polyspace-access` command. Before using the `polyspace-access` command, you have to write some additional code to find the folders directly containing the results file. You also have to create a name for each result as it would appear on the Polyspace Access web interface (otherwise, all uploads use the same default name and overwrite each other). The basic algorithm is the following:

- Recursively search all subfolders of the results folder for files with extension `.pscp`. Use the subfolder path that directly contains a `.pscp` file as argument for the `-upload` option of the `polyspace-access` command.
- Create a result name based on the path from the top of the results folder to the subfolder directly containing the `.pscp` file. Use this name as argument for the `-project` option of the `polyspace-access` command.

A sample Windows batch file for upload can look like this:

```
echo off
setlocal enabledelayedexpansion
set POLYSPACE_AUTOSAR_PATH=C:\Program Files\Polyspace Server\R2019a\polyspace\bin

rem Recursively search for all files with extension .pscp
dir *.pscp /b /s > file.txt

rem Upload each result file to Polyspace Access web interface
for /f "delims=" %%g in (file.txt) do (
  rem Get full path to result file with extension .pscp
  set filePath=%%~dpg
  rem Remove the current folder from the full path, then replace '\' with '.',
  rem Then remove leading and trailing '.'
  set projectName=!filePath:~cd%=!
  set projectName=!projectName:\=.!
  set projectName=!projectName:~1,-1!
  polyspace-access login -parent-project autosar
    -upload "!filePath:~0,-1!" -project !projectName!
)

Pause
```

In this script, the variable *login* refers to the following combination of options. You provide these options with every use of the `polyspace-access` command.

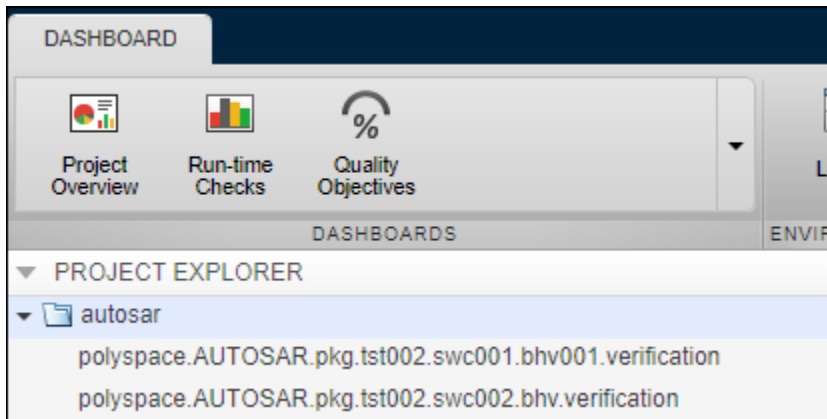
```
-host hostName -port portNumber -login username -encrypted-password pwd
```

Here, *hostName* is the name of the Polyspace Code Prover Access web server. For a locally hosted server, use `localhost`. *portNumber* is the optional port number of the server. If you omit the port number, 9443 is used. *username* and *pwd* refer to the login and an encrypted version of your password. To create an encrypted password, enter:

```
polyspace-access -encrypt-password
```

Copy the encrypted password and provide this password with later uses of the `polyspace-access` command.

Once the results are uploaded, you can see them in the Polyspace Access web interface. In the preceding script, a project name `autosar` is used with the option `-parent-project` for all results files. After upload, all results appear under this parent project.



For more information on how to review the results, see Polyspace Code Prover Access documentation.

See Also

[polyspace-access](#) | [polyspace-autosar](#)

More About

- “Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis” on page 2-16

Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis

If you run Polyspace on an AUTOSAR project by providing only the ARXML and source root folder, you might run into setup errors.

- A typical AUTOSAR root folder contains many extraneous files, for instance, files for testing and documentation. These files might have extensions such as `.arxml` or `.c`, and Polyspace might incorrectly treat them as ARXML or source files and include them in the analysis.
- The implementation of some software components might be in progress and incomplete.

If you are familiar with the structure of your AUTOSAR project, you can exclude extraneous and incomplete files and folders from the analysis.

The `polyspace-autosar` options that support file selections are `-select-arxml-files` and `-select-source-files`. Since the fully qualified names of AUTOSAR behaviors and data types use similar separators as file paths, you can also specify behaviors with `-autosar-behavior` and data types with `-autosar-datatype` using the file selection syntax.

Adapt Linux `find` Command to Select Files

The file selection syntax closely emulates the Linux `find` command. You specify a root folder followed by file inclusions and exclusions by using shell patterns or regular expressions.

For instance, to find all files with extension `.arxml` in a folder `package`, you use the Linux command:

```
find package -name '*.arxml'
```

To specify all ARXML files in the folder `package`, copy the part of the command following `find`. Provide the copied content as a string argument to the option `-select-arxml-files` of the `polyspace-autosar` command:

```
polyspace-autosar -select-arxml-files "package -name '*.arxml'"
```

In other words, you select ARXML files by specifying the root folder plus `find` options as a single string (within double quotes). During analysis, these ARXML files are not copied over to a temporary folder but selected from their respective locations.

If you enter the option in an options file (that you later use with the `polyspace-autosar` option `-options-file`), you do not need the double quotes around the string. You can append the `find` string to the `-select-arxml-files` option. For instance, you can enter this line in the options file:

```
-select-arxml-files package -name '*.arxml'
```

File Selection Options

The following `find` options are commonly required for file selection. Use the `i` format of the options for case-insensitive matching.

- `-name`, `-iname`: Match file names with shell patterns.
- `-path`, `-ipath`: Match file paths with shell patterns.

- `-regex`, `-iregex`: Use regular expressions for matching instead of shell patterns.

Use `-not` before an option to exclude files or folders. To specify multiple patterns in a single string, simply place the patterns one after another. For instance, to exclude immediate subfolders `subpackage1` and `subpackage2` from a root folder, use this syntax:

```
-not -path 'subpackage1/*' -not -path 'subpackage2/*'
```

For more information on:

- Shell patterns, see Shell Pattern Matching.
- `find` options, see Finding Files.

File Selection Examples

Some common uses of the file selection options are:

- To specify only the ARXML files beginning with `swc` in the subfolder `sub_package_windowControl` within the root folder `package`, use this syntax:

```
polyspace-autosar -select-arxml-files "package
                  -path 'sub_package_windowControl/*'
                  -name 'swc*.arxml'"
```

- To exclude ARXML files from the subfolder `test` at any level in the file structure within the root folder `package`, use this syntax:

```
polyspace-autosar -select-arxml-files "package
                  -not -path '*/test/*'
                  -name '*.arxml'"
```

- To exclude ARXML files from the subfolder `test` and subfolder `docs` at any level in the file structure within the root folder `package`, use this syntax:

```
polyspace-autosar -select-arxml-files "package
                  -not -path '*/test/*'
                  -not -path '*/docs/*'
                  -name '*.arxml'"
```

- To include all ARXML files from the root folder `package` except those ARXML files containing the string `test`, use this syntax:

```
polyspace-autosar -select-arxml-files "package
                  -name '*.arxml'
                  -not -name '*test*.arxml'"
```

Likewise, you can include or exclude `.c` and `.h` files from analysis by using the `-select-source-files` option. Unlike `-select-arxml-files`, this option selects `.c` and `.h` files by default. For instance, to exclude source files from the subfolder `test` at any level in the file structure within the root folder `package`, use this syntax:

```
polyspace-autosar -select-source-files "package -not -path '*/test/*'"
```

Note that with `-select-arxml-files` earlier, you also had to specify the additional pattern `-name '*.arxml'`.

Root Folder Specification

If the root folder name contains spaces, enclose the folder name in double quotes. Because the folder name with file inclusions and exclusions is already in double quotes, you have to escape the additional quotes. For instance, if the root folder is C:\sdbx\ARXML dir1, to specify all ARXML files within this folder, use the command:

```
polyspace-autosar -select-arxml-files "\"C:\sdbx\ARXML dir1\" -name '*.arxml'"
```

The additional double quotes around the root folder are escaped as \".

If you specify the option in an options file (that you later use with the `polyspace-autosar` option - `options-file`), you do not need quotes around the `find` string and do not need to escape the additional double quotes. Enter this line in the options file:

```
-select-arxml-files "C:\sdbx\ARXML dir1" -name '*.arxml'
```

See Also

`polyspace-autosar`

More About

- “Create Polyspace Analysis Configuration from AUTOSAR Specifications” on page 2-12

Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code

To analyze code implementation of AUTOSAR software components, Polyspace parses the AUTOSAR XML (ARXML) specifications, detects the corresponding code implementation, compiles this code and runs static analysis to detect run-time errors or mismatch between code and specifications. If an error occurs in any of these steps, you do not see analysis results for the software component containing the error. This tutorial shows how to locate and diagnose a class of errors that can occur during parsing of ARXML specifications.

Even if some ARXML specifications have ill-defined elements, the analysis tries to continue further with an ad hoc substitute for those elements, with the assumption that the code implementation might not use those elements. However, for specific types of issues, this substitution is not possible. Therefore, even if the ARXML extraction phase completes with warnings only, the warnings themselves are of two types:

- Warnings that report issues where the analysis is unable to create a substitute.

For instance, if an event calls a runnable with an undefined port, the analysis cannot model that event.

- Warnings that report issues where the analysis proceeds with a degraded substitute.

For instance, if a data-type used in a runnable or an RTE API function is undefined, the analysis proceeds with a degraded data type.

To follow the steps in this tutorial, use the demo files in `polyspaceroot\help\toolbox\codeprover\examples\troubleshooting_polyspace_autosar`.

Overview of File Structure

The demo files consist of a root folder `src` and two options files:

- The options file `options_ko.txt` selects files from the `src` folder that have deliberately introduced errors.
- The options file `options_ok.txt` selects files from the `src` folder that have the same errors fixed.

The folder `src` has two subfolders:

- `arxml` containing the AUTOSAR XML specifications.
- `impl` containing the code implementation of those specifications.

The `arxml` folder has multiple subfolders. Two of these subfolders, `appli` and `interfaces`, have subsubfolders `ok` and `ko` at different levels within the folder structure. The `ok` and `ko` subsubfolders contain the same set of files, except that the files in `ko` have deliberately introduced errors.

See File Selections

Open the options files `options_ok.txt` and `options_ko.txt` in a text editor and note the files selections in each:

- The options file `options_ok.txt` excludes files in `ko` subfolders at any level of the file hierarchy.

Note the use of the file selection pattern:

```
-not -path '*/ko/*'
```

- The options file `options_ko.txt` excludes files in the `ok` subfolders at any level of the file hierarchy.

Note the use of the file selection pattern:

```
-not -path '*/ok/*'
```

In addition, both options files exclude a specific file in the `types` subfolder named `do_not_use_this_arxml_file.arxml` using the pattern:

```
-path '*/types/*' -not -name 'do_not_use_this_arxml_file.arxml'
```

The full file selection pattern in the file `options_ko.txt` requires that all these criteria must be satisfied:

- The files must not be in a `ko` subfolder at any level of the hierarchy.
- The files must not have the extension `.arxml`.
- The files must be in one of the folders `appli`, `interfaces`, or `types` (except the file `do_not_use_this_arxml_file.arxml`).

For more information on file selection patterns, see “Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis” on page 2-16.

Run Analysis

To run the analysis, in a terminal, enter the command:

```
polyspace-autosar -options-file options_ko.txt
```

This command assumes that the path `polyspaceroot\polyspace\bin` is already added to the `PATH` variable in your operating system. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2020b`. Otherwise, use the full path to the `polyspace-autosar` command.

Repeat the run with the file `options_ok.txt`.


Note that the options files use the option `-do-not-update-verification` to stop the analysis before the code verification phase.

Interpret Warnings

The results of the analysis with options files `options_ok.txt` and `options_ko.txt` are stored in the folders `project_ok` and `project_ko` respectively.

Navigate to the folder `project_ko` and open the file `psar_project.xhtml` in a web browser. You see errors and warnings both in the ARXML parsing and code extraction phase.

For more information on the errors:

- 1 Click the  icon on the upper left. On the left pane, click **Behaviors**.

In the **Detailed status per AUTOSAR Behavior** section, note that:

- The behavior `tst003.app.swc001.bhv` has warnings in the ARXML parsing phase and errors and warnings in the code extraction phase.
- The behavior `tst003.app.swc002.bhv` does not have errors or warnings.

You can also filter out behaviors that do not have errors or warnings. On the left pane, in the **Behavior Selection** section, select **behaviors with error-status** and click **Search**.

- 2 For further details on the behavior that has errors and warnings, `tst003.app.swc001.bhv`:
- a In the section **Read AUTOSAR behavior**, click the link **See key autosar definition for this behavior**.
 - b Expand the error message at the top. This error illustrates a situation where Polyspace cannot model an event because of an issue in the ARXML specification. In this case, a port is not defined.
 - c On the left pane, in the **Function selection** section, select **all having error or warning** and click **Search**. You see one other error message. Expand the error message. This error illustrates a situation Polyspace can create a model despite an error. In this case, a data type is not defined and the analysis continues with a degraded type.

Based on the messages, you can locate the exact errors in the ARXML.

You also see code extraction errors in this behavior. These code extraction errors can be traced back to the issues in the ARXML. If you fix the issues in the ARXML, the code extraction errors are also fixed. You can see a fixed project by running an analysis with the options file `options_ok.txt`.

See Also

`polyspace-autosar`

More About

- “Create Polyspace Analysis Configuration from AUTOSAR Specifications” on page 2-12

Offload Polyspace Analysis to Remote Servers from Desktop

- “Send Polyspace Analysis from Desktop to Remote Servers” on page 3-2
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 3-6

Send Polyspace Analysis from Desktop to Remote Servers

In this section...

“Client-Server Workflow for Running Analysis” on page 3-2

“Prerequisites” on page 3-3

“Offload Analysis in Polyspace User Interface” on page 3-3
--

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated servers. You offload a Polyspace analysis from a Polyspace desktop product such as Polyspace Bug Finder but the analysis runs on the server using a Polyspace server product such as Polyspace Bug Finder Server.

This topic shows how to send a Polyspace analysis from the user interface of the Polyspace desktop products.

- To offload an analysis with scripts, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 3-6.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send Code Prover Analysis from Desktop to Locally Hosted Server”. In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

- 1 Client node:** You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis up to compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

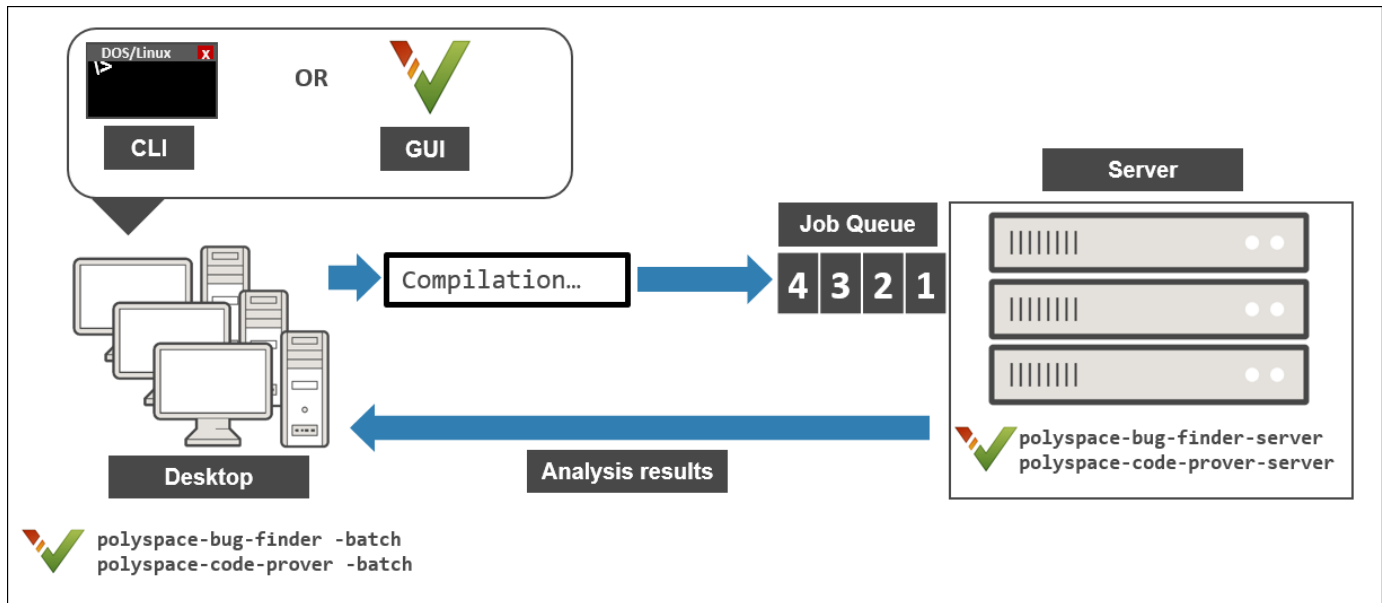
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

You require the product MATLAB Parallel Server on the computer that acts as the head node.

- 3 Worker nodes:** When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server, to run the analysis.



Prerequisites

Before offloading an analysis from the user interface of the Polyspace desktop products, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, for more information on:

- How to add source files, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Code Prover).
- How to set up communication between client and server, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

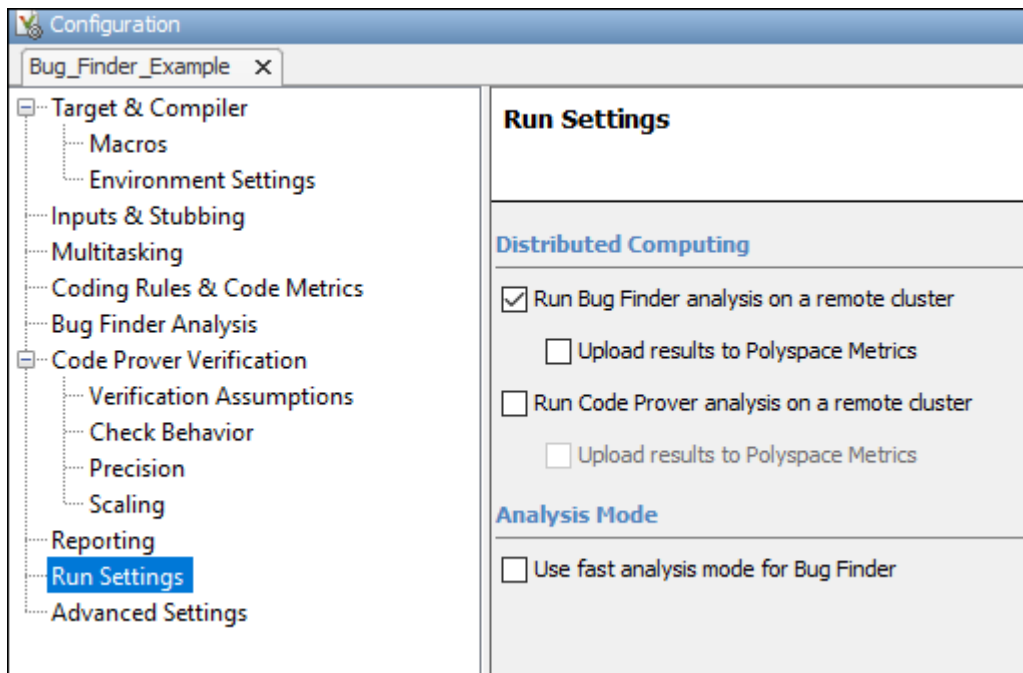
Once you have set up a Polyspace project and established communication between a desktop and a remote server, you are ready to offload a Polyspace analysis.

Offload Analysis in Polyspace User Interface

To start a remote analysis:

- 1 Select a project to analyze.
- 2 On the **Configuration** pane, select **Run Settings**.

Select **Run Bug Finder analysis on a remote cluster** and/or **Run Code Prover analysis on a remote cluster**.



- 3 If you want to store your results in the Polyspace Metrics repository, select **Upload results to Polyspace Metrics**.

Otherwise, clear this check box. After analysis, the results are downloaded to the desktop for review.

- 4 Start the analysis. For instance, to start a Bug Finder analysis, click the **Run Bug Finder** button.

The compilation part of the analysis takes place on the desktop product. After compilation, the analysis is offloaded to the server.

- 5 To monitor the analysis, select **Tools > Open Job Monitor**. In the Polyspace Job Monitor, follow your queued job to monitor progress.

Once the analysis is complete, the results are downloaded back to the user interface of the Polyspace desktop products. You can open the results directly in the user interface. If you uploaded the results to Polyspace Metrics, you have to explicitly download them from the Polyspace Metrics interface.

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

Note If you choose to upload results to Polyspace Metrics, your results are not downloaded automatically after verification. Use the Polyspace Metrics web dashboard to view the results and download them to your desktop. For more information, see “View Code Quality Metrics” (Polyspace Code Prover).

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (-batch)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 3-6

Send Polyspace Analysis from Desktop to Remote Servers Using Scripts

Instead of running a Polyspace analysis on your local desktop, you can send the analysis to a remote cluster. You can use a dedicated cluster for running Polyspace to free up memory on your local desktop.

This topic shows how to use Windows or Linux scripts to send the analysis to a remote cluster and download the results to your desktop after analysis.

- To offload an analysis from the Polyspace user interface, see “Send Polyspace Analysis from Desktop to Remote Servers” on page 3-2.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send Code Prover Analysis from Desktop to Locally Hosted Server”. In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

- 1 Client node:** You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis up to compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

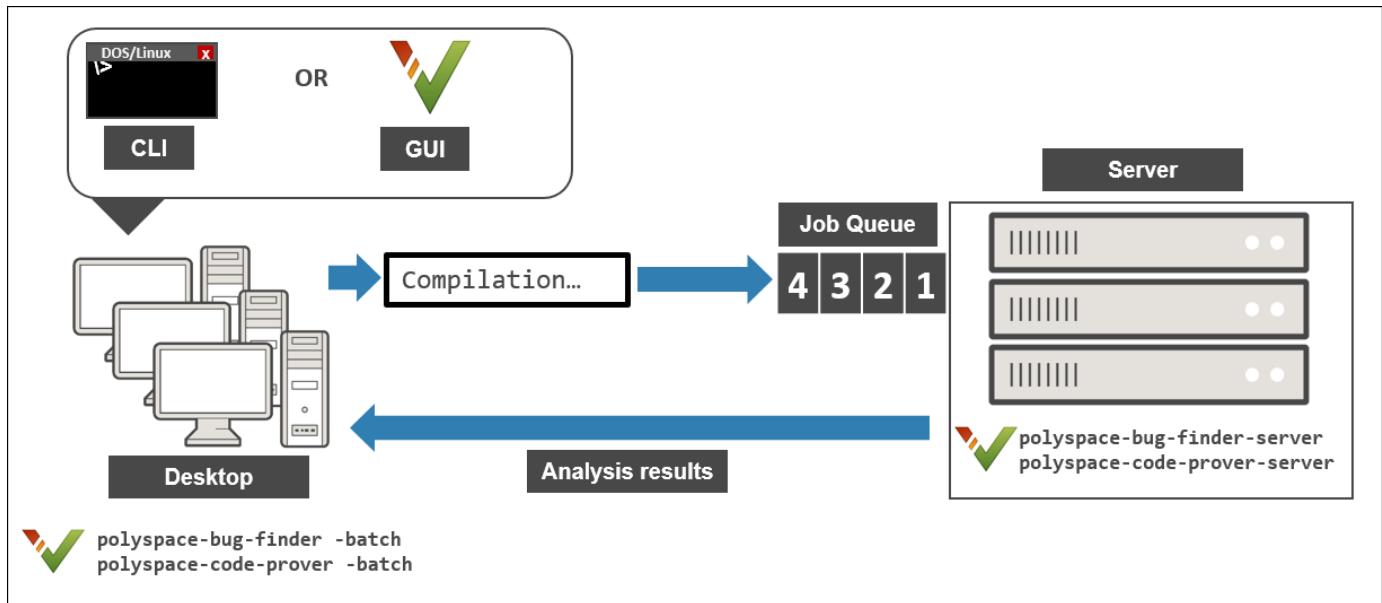
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

You require the product MATLAB Parallel Server on the computer that acts as the head node.

- 3 Worker nodes:** When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server to run the analysis.



Prerequisites

Before you run a remote analysis by using scripts, you must set up communication between a desktop and a remote server. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Run Remote Analysis

To run a remote analysis, use this command:

```
polyspaceroot\polyspace\bin\polyspace-code-prover
  -batch -scheduler NodeHost|MJSName@NodeHost [options] [-mjs-username name]
```

where:

- *polyspaceroot* is the installation folder of Polyspace desktop products, for instance, `C:\Program Files\Polyspace\R2020b`.
- *NodeHost* is the name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

MJSName is the name of the MATLAB Job Scheduler on the head node host.

If you set up communications with a cluster from the Polyspace user interface, you can determine *NodeHost* and *MJSName* from the user interface. Select **Metrics > Metrics and Remote Server Settings**. Open the MATLAB Parallel Server Admin Center. Under **MATLAB Job Scheduler**, see the **Name** and **Hostname** columns for *MJSName* and *NodeHost*.

If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *MJSName* is the argument of the option `-name`. For details, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

- *options* are the analysis options. These options are the same as that of a local analysis. For instance, you can use these options:
 - `-sources-list-file`: Specify a text file with one source file name per line.
 - `-options-file`: Specify a text file with one option per line.
 - `-results-dir`: Specify a download folder for storing results after analysis.

For the full list of options, see “Analysis Options”. Alternatively, you can:

- Start an analysis in the user interface and stop after compilation. You can obtain the text files and scripts for running the analysis at the command line. See “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 1-11.
- Enter `polyspace-codeprover -h`. The list of available options with a brief description are displayed.
- Place your cursor over each option on the **Configuration** pane in the Polyspace user interface. Click the **More Help** button for information on the option syntax and when the option is required.
- *name* is the username required for job submissions using MATLAB Parallel Server. These credentials are required only if you use a security level of 1 or higher for MATLAB Parallel Server submissions. See “Set MATLAB Job Scheduler Cluster Security” (MATLAB Parallel Server).

The analysis executes locally on your desktop up to the end of the compilation phase. After compilation, the software submits the analysis job to the cluster and provides a job ID. You can also read the ID from the file `ID.txt` in the results folder. To monitor your analysis, use the `polyspace-jobs-manager` command with the job ID.

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

Manage Remote Analysis

To manage multiple remote analyses, use the option `-batch`. For instance:

```
polyspaceroot\polyspace\bin\polyspace-jobs-manager action  
-scheduler schedulerName
```

See also Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`). Here:

- *polyspaceroot* is your MATLAB installation folder.
- *schedulerName* is one of the following:
 - Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*NodeHost*).
 - Name of the MATLAB Job Scheduler 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. To see the scheduler name, select **Tools > Preferences**. On the **Server Configuration** tab, see the **Job scheduler host name**.

- `action` refers to the possible action commands to manage jobs on the scheduler:

- `listjobs`:

Generate a list of Polyspace jobs on the scheduler. For each job, the software produces this 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.
- `download -job ID -results-folder FolderPath`:

Download results of analysis with specified ID to folder specified by *FolderPath*.

When the analysis job is queued on the server, the command `polyspace-code-prover` returns a job id. In addition, a file `ID.txt` in the results folder contains the job ID in this format:

```
job_id;server_name:project_name version_number
```

For instance, `92;localhost:Demo 1.0`.

If you do not use the `-results-folder` option, the software downloads the result to the folder that you specified when starting analysis, using the `-results-dir` option.

After downloading results, use the Polyspace user interface to view the results.

- `getlog -job ID`:

Open log for job with specified ID.

- `remove -job ID`:

Remove job with specified ID.

- `promote -job ID`:

Promote job with specified ID in the queue.

- `demote -job ID`

Demote job with specified ID in the queue.

Sample Scripts for Remote Analysis

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 by using a shell script. To create a batch file for running analysis:

- 1 Save your analysis options in a file `listoptions.txt`. See `-options-file`.
- 2 Create a file `launcher.bat` in a text editor like Notepad.

In the file, enter these commands:

```
echo off
set POLYSPACE_PATH=polyspaceroot\polyspace\bin
set RESULTS_PATH=C:\Results
set OPTIONS_FILE=C:\Options\listoptions.txt
"%POLYSPACE_PATH%\polyspace-code-prover.exe" -batch -scheduler localhost
    -results-dir %RESULTS_PATH% -options-file %OPTIONS_FILE%
pause
```

polyspaceroot is the Polyspace installation folder. *localhost* is the name of the computer that hosts the head node of your MATLAB Parallel Server cluster.

- 3 Replace the definitions of these variables in the file:
 - POLYSPACE_PATH: Enter the actual location of the `.exe` file.
 - RESULTS_PATH: Enter the path to a folder. The files generated during compilation are saved in the folder.
 - OPTIONS_FILE: Enter the path to the file `listoptions.txt`.
- 4 Double-click `launcher.bat` to run the analysis.

Tip If you run a Polyspace analysis, a Windows `.bat` or Linux `.sh` file is generated. The file is in the `.settings` subfolder in your results folder. Instead of writing a script from scratch, you can relaunch the analysis using this file.

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers” on page 3-2

Run Polyspace Analysis on Server with MATLAB Scripts

Integrate Polyspace Server Products with MATLAB and Simulink

You can install Polyspace Bug Finder Server and Polyspace Code Prover Server as standalone products and analyze C/C++ code. However, if you have an installation of MATLAB, you can run the Polyspace analysis with MATLAB scripts.

If you install Polyspace server products and MATLAB, you have to run the MATLAB installer twice and install Polyspace in a different root folder from the other products. For instance, in Windows:

- Your default MATLAB root folder is C:\Program Files\MATLAB\R2019a.
- Your default Polyspace root folder is C:\Program Files\Polyspace Server\R2019a for the Polyspace server products.

To run Polyspace from within MATLAB, Simulink or MATLAB Coder™, you have to perform a post-installation step to link your MATLAB and Polyspace installations.

Integrate Polyspace with MATLAB Installation from Same Release

If your Polyspace and MATLAB installations belong to the same release, you can use all MATLAB functions and classes available for running Polyspace.

To link your MATLAB and Polyspace installations:

- 1 Open MATLAB with administrator privileges.
- 2 At the MATLAB command prompt, enter:

```
polyspacesetup('install', 'polyspaceFolder', FOLDER);
```

FOLDER is the path to the folder where you installed Polyspace. The default folder is C:\Program Files\Polyspace\R2020b. You see a prompt stating that the workspace will be cleared and all open models closed. Click **Yes** to continue the linking. The process might take a few minutes to complete. To avoid the prompt during installation, enter:

```
polyspacesetup('install', 'polyspaceFolder', FOLDER, 'silent', true);
```

- 3 Restart MATLAB. You can now use all functions and classes available for running Polyspace server products.

A MATLAB installation can be linked with only one Polyspace installation. To link to a new Polyspace installation, any previous links must be removed. To remove a link between a Polyspace and MATLAB installation, repeat the same steps as before with one difference: At the MATLAB command prompt, enter:

```
polyspacesetup('uninstall')
```

Integrate Polyspace with MATLAB Installation from Different Release

If you upgrade your Polyspace server product installation but not your MATLAB installation, you can link your MATLAB installation with the later release of the Polyspace server product.

Remove the existing link between your Polyspace and MATLAB installation as described in the previous section. Then, in your post-installation step, navigate to `polyspaceserverroot\toolbox\polyspace\pscore\pscore\`, where `polyspaceroot` is the installation folder for the later release of Polyspace Bug Finder Server and/or Polyspace Code Prover Server. At the MATLAB command prompt, enter:

```
polyspacesetup('install')
```

To avoid prompts during installation, enter:

```
polyspacesetup('install', 'silent', true)
```

If you integrate MATLAB with a later release of Polyspace, you cannot use all functions and classes available to run the analysis. In particular, you cannot use the `polyspace.Project` class. Instead, use the `polyspaceCodeProverServer` function to run Code Prover and the `polyspaceBugFinderServer` function to run Bug Finder on handwritten code.

Check Integration Between MATLAB and Polyspace

To check if a MATLAB installation is already linked to a Polyspace installation, open MATLAB and enter:

```
ver
```

You see the list of products installed. If Polyspace is linked to MATLAB (after R2019a) or in the same installation folder as MATLAB (prior to R2019a), you can see the Polyspace products in the list.

The MATLAB-Polyspace integration adds some Polyspace installation subfolders to the MATLAB search path. To see which paths were added, enter:

```
polyspacesetup('showpolyspacefolders')
```

Run Polyspace Server Products with MATLAB Scripts

In a continuous integration process, you can execute MATLAB scripts that run a Polyspace analysis on new code submissions and compares the results against predefined criteria. Use these functions/classes:

- Create a `polyspace.Project` object to configure Polyspace analysis options, run an analysis and read results to MATLAB tables. You can use other MATLAB functions for comparing results against predefined criteria.

To only read existing results without running an analysis, use the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` class with the path to a results folder.

- If you want a more granular selection of checkers for:
 - Coding rules, create a `polyspace.CodingRulesOptions` object.
 - Bug Finder defects, create a `polyspace.DefectsOptions` object.

To create a custom target for the analysis and explicitly specify sizes of data types, create a `polyspace.GenericTargetOptions` object.

You can also use the `polyspaceBugFinderServer` or `polyspaceCodeProverServer` function to run the analysis and then read results with the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` class. If you use build commands to build your source code, you can create a Polyspace configuration from the build command using the `polyspaceConfigure` function.

See Also

Configure Target and Compiler Options

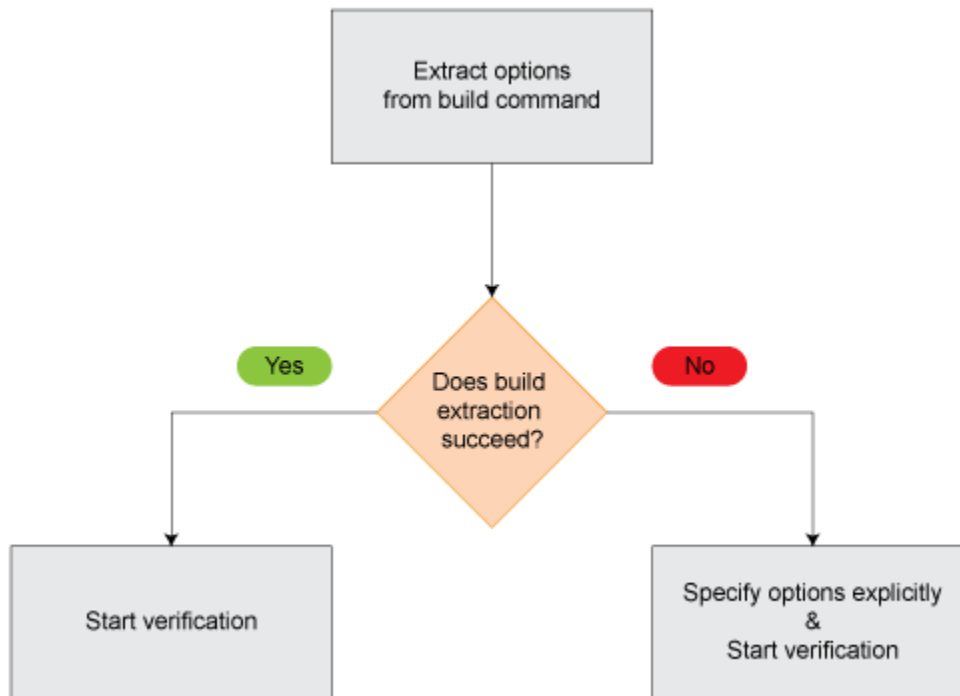
Specify Target Environment and Compiler Behavior

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

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

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

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



Extract Options from Build Command

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

In the Polyspace desktop products, for information on how to trace your build command from the:

- Polyspace user interface, see "Add Source Files for Analysis in Polyspace User Interface" (Polyspace Code Prover).
- DOS or UNIX command line, see `polyspace-configure`.

- MATLAB command line, see `polyspaceConfigure`.

In the Polyspace server products, for information on how to trace your build command, see “Create Polyspace Analysis Configuration from Build Command” on page 2-2.

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See “Requirements for Project Creation from Build Systems” on page 5-20.

Specify Options Explicitly

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

- In the user interface of the Polyspace desktop products, select a project configuration. On the **Configuration** pane, select **Target & Compiler**. Specify the options.
- At the DOS or UNIX command line, specify flags with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command.
- At the MATLAB command line, specify arguments with the `polyspaceBugFinder`, `polyspaceCodeProver`, `polyspaceBugFinderServer` or `polyspaceCodeProverServer` function.

Specify the options in this order.

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

If you cannot find your target processor, you can create your own target and specify the sizes of fundamental types, default signedness of `char`, and endianness of the target machine. See **Generic target options**.

- Language-specific options:
 - **C standard version (-c-version)**: The default C language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. Specify an earlier standard such as C90 or a later standard such as C11.
 - **C++ standard version (-cpp-version)**: The default C++ language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. Specify later standards such as C++11 or C++14.
- Compiler-specific options:

Whether these options are available or not depends on your specification for `Compiler (-compiler)`. For instance, if you select a `visual` compiler, the option `Pack alignment value`

(`-pack-alignment-value`) is available. Using the option, you emulate the compiler option `/Zp` that you use in Visual Studio.

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

- Advanced options:

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

For all advanced options, see “Target and Compiler”.

- Compiler header files:

If you specify the `diab`, `tasking` or `greenhills` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.

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

- *Define macros*: Sometimes, a compilation error occurs because the analysis considers a macro as undefined. Explicitly define these macros. See `Preprocessor definitions (-D)`.
- *Specify include files*: Sometimes, a compilation error occurs because your compiler defines standard library functions differently from Polyspace and you do not provide your compiler include files. Explicitly specify the path to your compiler include files. See “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.

See Also

`C standard version (-c-version)` | `C++ standard version (-cpp-version)` | `Compiler (-compiler)` | `Preprocessor definitions (-D)` | `Source code language (-lang)` | `Target processor type (-target)`

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-5
- “Provide Standard Library Headers for Polyspace Analysis” on page 5-19

C/C++ Language Standard Used in Polyspace Analysis

The Polyspace analysis adheres to a specific language standard for code compilation. The language standard, along with your compiler specification, defines the language elements that you can use in your code. For instance, if the Polyspace analysis uses the C99 standard, C11 features such as use of the thread support library from `threads.h` causes compilation errors.

Supported Language Standards

The Polyspace analysis supports these standards:

- **C:** C90, C99, C11

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. To change the language standard, use the option `C standard version (-c-version)`.

- **C++:** C++03, C++11, C++14

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. To change the language standard, use the option `C++ standard version (-cpp-version)`.

Default Language Standard

The default language standard depends on your specification for the option `Compiler (-compiler)`.

Compiler	C Standard	C++ Standard
generic	C99	C++03
gnu3.4, gnu4.6, gnu4.7, gnu4.8, gnu4.9	C99	C++03
gnu5.x	C11	C++03
gnu6.x	C11	C++14
gnu7.x	C11	C++14
clang3.x	C99	C++03 The analysis accepts some C++11 extensions.
clang4.x	C99	C++03 The analysis accepts C++14 extensions.
clang5.x	C99	C++03 The analysis accepts C++14 extensions.
visual9.0, visual10.0, visual11.0, visual12.0	C99	C++03

Compiler	C Standard	C++ Standard
visual14.0	C99	C++14
visual15.x	C99	C++14
keil	C99	C++03
iar	C99	C++03
armcc	C99	C++03
armclang	C11	C++03
codewarrior	C99	C++03
cosmic	C99	Not supported
diab	C99	C++03
greenhills	C99	C++03
iar-ew	C99	C++03
microchip	C99	Not supported
renesas	C99	C++03
tasking	C99	C++03
ti	C99	C++03

See Also

C standard version (-c-version) | C++ standard version (-cpp-version) | Compiler (-compiler)

More About

- “C11 Language Elements Supported in Polyspace” on page 5-7
- “C++11 Language Elements Supported in Polyspace” on page 5-9
- “C++14 Language Elements Supported in Polyspace” on page 5-12
- “C++17 Language Elements Supported in Polyspace” on page 5-15

C11 Language Elements Supported in Polyspace

This table provides a partial list of C language elements that have been introduced since C11 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C11 Language Element	Supported
<code>alignas</code> and <code>alignof</code> convenience macros	Yes
<code>aligned_alloc</code> function	Yes
<code>noreturn</code> convenience macros	Yes
Generic selection	Yes
Thread support library (<code>threads.h</code>)	Yes
Atomic operations library (<code>stdatomic.h</code>)	Yes
Atomic types with <code>_Atomic</code>	Yes. If you use the Clang compiler, see limitations book for limitations on atomic data types. See “Limitations of Polyspace Verification” (Polyspace Code Prover).
UTF-16 and UTF-32 character utilities	Yes
Bound-checking interfaces or alternative versions of standard library functions that check for buffer overflows (Annex K of C11) For instance, <code>strcpy_s</code> is an alternative to <code>strcpy</code> that checks for certain errors in the string copy.	No. Polyspace checks for certain run-time errors in use of standard library functions. The checking does not extend to these alternatives.
Anonymous structures and unions	Yes
Static assert declaration	Yes
Features related to error handling such as <code>errno_t</code> and <code>rsize_t</code> typedef-s	No. If you see compilation errors from use of these typedef-s, explicitly specify the path to your compiler headers. See “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.
<code>quick_exit</code> and <code>at_quick_exit</code>	Yes. In Bug Finder, functions registered with <code>at_quick_exit</code> appear as uncalled.
<code>CMPLX</code> , <code>CMPLXF</code> and <code>CMPLXL</code> macros	Yes

See Also

C standard version (`-c-version`)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-5

C++11 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++11 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++11 Std Ref	Description	Supported
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N1757	Right angle brackets	Yes
C++2011-N1791	Extended friend declarations	No
C++2011-N1811	long long	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N1988	Extended integral types	Yes
C++2011-N2118	Rvalue references	Yes
C++2011-N2170	Universal character name literals	Yes
C++2011-N2179	Concurrency: Propagating exceptions	No
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2239	Concurrency: Sequence points	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2242	Variadic templates	Yes
C++2011-N2249	New character types	Yes
C++2011-N2253	Extending sizeof	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N2340	<code>__func__</code> predefined identifier	Yes
C++2011-N2341	Alignment support	Yes
C++2011-N2342	Standard Layout Types	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N2347	Strongly typed enums	Yes

C++11 Std Ref	Description	Supported
C++2011-N2427	Concurrency: Atomic operations	No
C++2011-N2429	Concurrency: Memory model	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2431	Null pointer constant	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N2440	Concurrency: Abandoning a process and at_quick_exit	Yes
C++2011-N2442	Unicode string literals	Yes
C++2011-N2442	Raw string literals	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2547	Concurrency: Allow atomics use in signal handlers	No
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2659	Concurrency: Thread-local storage	No
C++2011-N2660	Concurrency: Dynamic initialization and destruction with concurrency	Yes
C++2011-N2664	Concurrency: Data-dependency ordering: atomics and memory model	No
C++2011-N2672	Initializer lists	Yes
C++2011-N2748	Concurrency: Strong Compare and Exchange	No
C++2011-N2752	Concurrency: Bidirectional Fences	No
C++2011-N2756	Nonstatic data member initializers	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2765	User-defined literals	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes

C++11 Std Ref	Description	Supported
C++2011-N3276	decltype and call expressions	Yes

See Also

C++ standard version (-cpp-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-5
- “C++14 Language Elements Supported in Polyspace” on page 5-12
- “C++17 Language Elements Supported in Polyspace” on page 5-15

C++14 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++14 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++14 Std Ref	Description	Supported
C++2014-N3323	Implicit conversion from class type in certain contexts such as <code>delete</code> or <code>switch</code> statement.	This C++14 feature allows implicit conversion from class type in certain contexts. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3462	More SFINAE-friendly <code>std::result_of</code>	Yes
C++2014-N3472	Binary literals, for instance, <code>0b100</code> .	Yes
C++2014-N3545	<code>operator()</code> in <code>integral_constant</code> template of <code>constexpr</code> type	Yes
C++2014-N3637	Relation between <code>std::async</code> and destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3638	Automatic deduction of return type for functions where an explicit return type is not specified	Yes. In some cases, Code Prover can show compilation errors.
C++2014-N3642	Suffixes for user-defined literals indicating time (<code>h</code> , <code>min</code> , <code>s</code> , <code>ms</code> , <code>us</code> , <code>ns</code>) and strings (<code>s</code>)	Yes
C++2014-N3648	Initialization of captured members in lambda functions	Yes. In some cases, during initialization, Code Prover can call the corresponding constructors more number of times than necessary.
C++2014-N3649	Generic (polymorphic) lambda expressions: <ul style="list-style-type: none"> Using <code>auto</code> type-specifier for parameter and return type Conversion of generic capture-less lambda expressions to pointer-to-function. 	Yes

C++14 Std Ref	Description	Supported
C++2014-N3651	Variable templates	Yes
C++2014-N3652	Declarations, conditions and loops in <code>constexpr</code> functions.	Yes
C++2014-N3653	<p>Initialization of aggregate classes with fewer initializers than members</p> <p>For instance, this initialization has fewer initializers than members. The member <code>c</code> is initialized with the value 0 and <code>d</code> is initialized with the value <code>s</code>.</p> <pre>struct S { int a; const char* b; int c; int d = b[a];}; S ss = { 1, "asdf" };</pre>	Yes
C++2014-N3654	<code>std::quoted</code>	Yes
C++2014-N3656	<code>std::make_unique</code>	Yes
C++2014-N3658	<code>std::integer_sequence</code>	Yes
C++2014-N3658	<code>std::shared_lock</code>	No. The use of <code>std::shared_lock</code> does not cause compilation errors but the construct is not semantically supported.
C++2014-N3664	Calling <code>new</code> and <code>delete</code> operators in batches.	This C++14 feature clarifies how successive calls to the <code>new</code> operator are implemented. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3668	<code>std::exchange</code>	Partially supported.
C++2014-N3670	Using <code>std::get</code> with a data type to get one element in an <code>std::tuple</code> (provided there is only one element of the type in the tuple)	Yes
C++2014-N3671	Overloads for <code>std::equal</code> , <code>std::mismatch</code> and <code>std::is_permutation</code> function templates that accept two separate ranges	Yes
C++2014-N3733	Removal of <code>std::gets</code> from <code><cstdio></code>	Yes

C++14 Std Ref	Description	Supported
C++2014-N3776	Wording change for destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3779	<code>std::complex</code> literals representing pure imaginary numbers with suffix <code>i</code> , <code>if</code> or <code>il</code>	Yes
C++2014-N3781	Use of single quotation mark as digit separator, for instance, <code>1'000</code> .	Yes
C++2014-N3786	Prohibiting "out of thin air" results in C++14	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3910	Synchronizing behavior of signal handlers	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3924	Discouraging use of <code>rand()</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3927	Lock-free executions	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.

See Also

C++ standard version (`-cpp-version`)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-5
- “C++11 Language Elements Supported in Polyspace” on page 5-9
- “C++17 Language Elements Supported in Polyspace” on page 5-15

C++17 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++17 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++17 Std Ref	Description	Supported
C++2017-N3921	<code>std::string-view</code> : Observe the content of an <code>std::string</code> object without owning the resource	Yes
C++2017-N3922	<ul style="list-style-type: none"> When used in copy-list-initialization, <code>auto</code> deduces the type to be an <code>std::initializer_list</code> if the elements of the initializer list have an identical type. Otherwise, the <code>auto</code> deduction is ill-formed. When using direct list-initialization with a braced initializer list containing a single element, <code>auto</code> deduces the type from that element. When using direct list-initialization with a braced initializer list containing more than a single element, <code>auto</code> deduction of type is ill-formed. 	Yes
C++2017-N3928	The <code>static_assert</code> declaration no longer requires a second argument. Invoking <code>static_assert</code> with no message is now allowed: <code>static_assert(N > 0);</code>	Yes
C++2017-N4051	C++ has templates that are not class templates, such as a template that takes templates as an argument. Previously, declaring such template-template parameters required the use of the <code>class</code> keyword. In C++17, you can use <code>typename</code> when declaring template-template parameters, such as: <pre>template <template <typename> typename Tmpl> struct X;</pre>	Yes
C++2017-N4086	Starting in C++17, trigraphs are no longer supported.	No
C++2017-N4230	Starting in C++17, use a qualified name in a namespace definition to define several nested namespaces at once. For instance, these code snippets are equivalent: <ul style="list-style-type: none"> <pre>namespace base::derived{ //.. }</pre> <pre>namespace { namespace derived{ //... } }</pre> 	Yes

C++17 Std Ref	Description	Supported
C++2017-N4259	The function <code>std::uncaught_exceptions</code> is introduced in C++17, which returns the number of exceptions in your code that are not handled. The function <code>std::uncaught_exception</code> , which returns a Boolean value, is deprecated.	Yes
C++2017-N4266	Starting in C++17, namespaces and enumerators can be annotated with attributes to allow clearer communication of developer intention.	Yes
C++2017-N4267	Starting in C++17, the prefix <code>u8</code> is supported. This prefix creates a UTF-8 character literal. The value of the UTF-8 character literal is equal to its ISO 10646 code point value if the code point value is in the C0 Controls and Basic Latin Unicode block.	Yes
C++2017-N4268	Allow constant evaluation of nontype template arguments.	Yes
C++2017-N4295	Allow fold expressions	Yes
C++2017-N4508	Allow untyped <code>std::shared_mutex</code>	The use of <code>std::shared_mutex</code> does not cause a compilation error. Polyspace does not support sharing mutex objects by using <code>std::shared_mutex</code> .
C++2017-P0001R1	Remove the use of the <code>register</code> keyword	Yes
C++2017-P0002R1	Remove <code>operator++(bool)</code>	Yes
C++2017-P0003R5	Remove deprecated exception specifications by using <code>throw(<>)</code>	Bug Finder removes the exception specification specified by using <code>throw()</code> statements. Code Prover raises a compilation error when <code>throw()</code> statements are present in C++17 code.
C++2017-P0012R1	Make exception specifications part of the type system	Yes
C++2017-P0017R1	Aggregate initialization of classes with base classes	Yes
C++2017-P0018R3	Allow capturing the pointer <code>*this</code> in Lambda expressions	Yes
C++2017-P0024R2	Standardization of the C++ technical specification for Extension for Parallelism	Polyspace supports this feature when you use the Visual 15.x and Intel C++ 18.0 compilers.

C++17 Std Ref	Description	Supported
C++2017-P002842	Using attribute namespaces without repetition	Yes
C++2017-P0035R4	Dynamic memory allocation for over-aligned data	Yes
C++2017-P0036R0	Unary fold expressions and empty parameter packs	Yes
C++2017-P0061R1	Use of <code>__has_include</code> in preprocessor conditionals	Yes
C++2017-P0067R5	Elementary string conversions	No
C++2017-P0083R3	Splicing maps and sets	Polyspace supports this feature when the compiler you use also supports this feature. For instance, Polyspace supports this feature when you use g++ as compiler.
C++2017-P0088R3	<code>std::variant</code>	Partially supported.
C++2017-P0091R3	Template argument deduction for class templates	Partially supported.
C++2017-P0127R2	Non-type template parameters that have auto type	Yes
C++2017-P0135R1	Guaranteed copy elision	Partially supported.
C++2017-P0136R1	New specification for inheriting constructors	No
C++2017-P0137R1	Replacement of class objects containing reference members	Yes
C++2017-P0138R2	Direct-list-initialization of enumerations	Yes
C++2017-P0145R3	Stricter expression evaluation order	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++17.
C++2017-P0154R1	Hardware interference size	Supported with Visual Studio Compiler
C++2017-P0170R1	<code>constexpr</code> Lambda expressions	Partially supported
C++2017-P018R0	Differing begin and end types in range-based for loops	Yes
C++2017-P0188R1	<code>[[fallthrough]]</code> attribute	Yes

C++17 Std Ref	Description	Supported
C++2017-P0189R1	[[nodiscard]] attribute	Yes
C++2017-P0195R2	Pack expansions in using-declarations	Yes
C++2017-P0212R1	[[maybe_unused]] attribute	Yes
C++2017-P0217R3	Structured Bindings	Polyspace does not support binding by using an rvalue.
C++2017-P0218R1	std::filesystem	No
C++2017-P0220R1	std::any	Yes
C++2017-P0220R1	std::optional	Bug Finder supports the syntax. The semantics are partially supported. Code Prover does not support this feature.
C++2017-P0226R1	Mathematical special functions	No
C++2017-P0245R1	Hexadecimal floating-point literals	Yes
C++2017-P0283R2	Ignore unknown attributes	Yes
C++2017-P0292R2	constexpr if statements	Yes
C++2017-P0298R3	std::byte	Yes
C++2017-P0305R1	init-statements for if and switch	Yes
C++2017-P0386R2	Inline variables	No
C++2017-P0522R0	Invoke partial ordering to determine when a template <i>template-argument</i> is a valid match for a <i>template-parameter</i>	Partially supported

See Also

C++ standard version (-cpp-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-5
- “C++11 Language Elements Supported in Polyspace” on page 5-9
- “C++14 Language Elements Supported in Polyspace” on page 5-12

Provide Standard Library Headers for Polyspace Analysis

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

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

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

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

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

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

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

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

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

- In the user interface (Polyspace desktop products), add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Code Prover).

- At the command line, use the flag `-I` with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command..

For more information, see `-I`.

See Also

More About

- “Errors from Conflicts with Polyspace Header Files” on page 12-58

Requirements for Project Creation from Build Systems

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

Compiler Requirements

- Your compiler must be called locally.

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

- Your compiler must perform a clean build.

If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` or `-W makefileName` option to force a clean build. For the list of options allowed with the GNU® `make`, see `make options`.

- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:

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

If your compiler configuration is not available to Polyspace:

- Write a compiler configuration file for your compiler in a specific format. For more information, see “Compiler Not Supported for Project Creation from Build Systems” on page 12-16.
- Contact MathWorks Technical Support. For more information, see “Contact Technical Support About Issues with Running Polyspace” on page 12-11.
- If you build your code in Cygwin™, you must be using version 2.x or 3.x of Cygwin for Polyspace project creation from your build system (for instance, Cygwin version 2.10 or 3.0).
- With the TASKING compiler, if you use an alternative `sfr` file with extension `.asfr`, Polyspace might not be able to locate your file. If you encounter an error, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

Typically, you use the statement `#include __SFRFILE__(__CPU__)` along with the compiler option `--alternative-sfr-file` to specify an alternative `sfr` file. The path to the file is typically `Tasking_C166_INSTALL_DIR\include\sfr\regCPUNAME.asfr`. For instance, if your

TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

Build Command Requirements

- Your build command must run to completion without any user interaction.
- In Linux, only UNIX shell (sh) commands must be used. If your build uses advanced commands such as commands supported only by bash, tcsh or zsh, Polyspace cannot trace your build.

In Windows, only DOS commands must be used. If your build uses advanced commands such as commands supported only by PowerShell or Cygwin, Polyspace cannot trace your build. To see if Polyspace supports your build command, run the command from `cmd.exe` in Windows. For more information, see “Check if Polyspace Supports Build Scripts” on page 12-23.

- If you use statically linked libraries, Polyspace cannot trace your build. In Linux, you can install the full Linux Standard Base (LSB) package to allow dynamic linking. For example, on Debian® systems, install LSB with the command `apt-get install lsb`.
- Your build command must not use aliases.

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

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

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

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

For example, if your command occurs as

```
command1 | command2
```

And you enter

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

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

- If the System Integrity Protection (SIP) feature is active on the operating system macOS El Capitan (10.11) or a later macOS version, Polyspace cannot trace your build command. Before tracing your build command, disable the SIP feature. You can reenble this feature after tracing the build command.

Similar considerations apply to other security applications such as security-related products from CylanceProtect, Avecto and Tanium.

- If your computer hibernates during the build process, Polyspace might not be able to trace your build.
- When creating projects from build commands in the Polyspace User Interface, you might encounter errors such as `libcurl.so.4: version 'CURL_OPENSSL_3' not found`. In such

cases, create the Polyspace project by using the command `polyspace-configure` in the system command line interface, using the build command as the argument. See `polyspace-configure`.

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

See Also

`polyspace-configure`

Related Examples

- “Create Polyspace Analysis Configuration from Build Command” on page 2-2

Supported Keil or IAR Language Extensions

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

Special Function Register Data Type

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

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

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

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

```
int status,P0;

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

To analyze this type of code, use these options:

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

For example, depending on how you define the `sbit` data type, you use these options:

- `sbit ADST = ADCUP^7;`
Use options: `-compiler keil -sfr-type sfr=8`
- `sbit ADST = ADCUP.7;`
Use options: `-compiler iar -sfr-type sfr=8`

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

Keywords Removed During Preprocessing

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

If you disable the keyword and use it as an identifier instead, you can encounter a compilation error when you compile your code with Polyspace. See “Errors Related to Keil or IAR Compiler” on page 12-41.

These keywords are removed during preprocessing:

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

The `data` keyword is not removed.

Remove or Replace Keywords Before Compilation

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

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

Remove Unrecognized Keywords

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

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

Content of myTpl.pl

```
#!/usr/bin/perl

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


    # DON'T DELETE LINE BELOW: Print the current processed line
    print $OUTFILE $_;
}
```

For reference, see a summary of Perl regular expressions.

Perl Regular Expressions

```
#####
# Metacharacter What it matches
```

```
#####
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurrence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc All of abc respectively
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
#####
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
#####
```

- 2 On the **Configuration** pane, select **Environment Settings**.
- 3 To the right of **Command/script to apply to preprocessed files**, click .
- 4 Use the Open File dialog box to navigate to C:\Polyspace.
- 5 In the **File name** field, enter myTpl.pl.
- 6 Click **Open**. You see C:\Polyspace\myTpl.pl in the **Command/script to apply to preprocessed files** field.

Remove Unrecognized Function Attributes

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

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

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

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

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

```
while ($line = <STDIN>)
{
    # __attribute__ ((noreturn))

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

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

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

See Also

Polyspace Analysis Options

`Command/script` to apply to preprocessed files (`-post-preprocessing-command`) |
Preprocessor definitions (`-D`)

Related Examples

- “Troubleshoot Compilation Errors”

Gather Compilation Options Efficiently

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

- The Polyspace compiler strictly follows a C or C++ Standard (depending on your choice of compiler). See “C/C++ Language Standard Used in Polyspace Analysis” on page 5-5. If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler.
- Your compiler declares standard library functions with argument or return types different from the standard types. Unless you also provide the function definition, for efficient verification, Polyspace uses its own definitions of standard library functions, which have the usual prototype. The mismatch in types causes a linking error.

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

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

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

Benefits of this approach include:

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

Example 5.1. Example

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

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

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

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



```
// Workarounds for errors due to redefining standard library functions

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

See Also

More About

- “Troubleshoot Compilation Errors”

Configure Inputs and Stubbing Options

Specify External Constraints

This example shows how to specify constraints (also known as 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:

- Code Prover can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path comes from an assumption that is too broad, the orange check might indicate a false positive.
- Bug Finder can sometimes produce false positives.

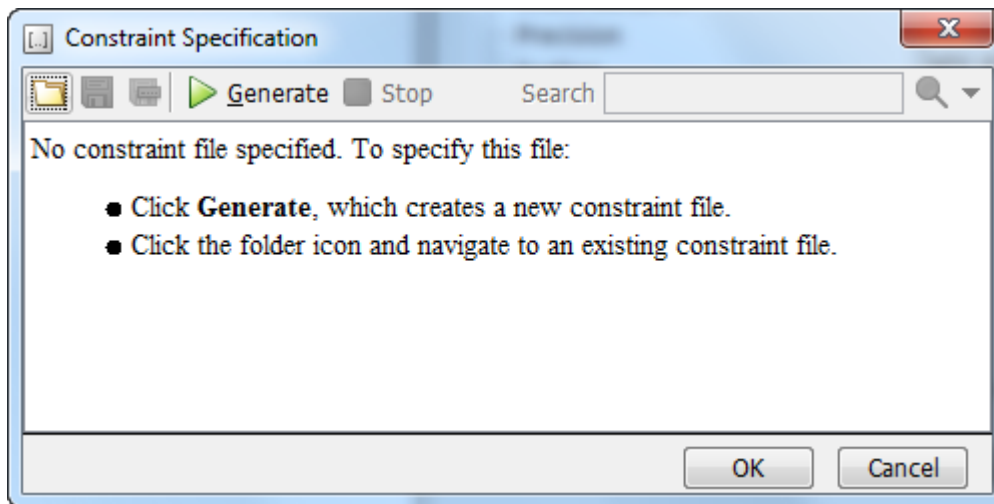
To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values and modifiable arguments of stubbed functions. You save the constraints as an XML file to use them for subsequent analyses. If your source code changes, you can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

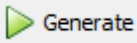
Note In Bug Finder, you can only constrain global variables. You cannot constrain function inputs or return values of stubbed functions.

Create Constraint Template

User Interface (Desktop Products Only)

- 1 Open the project configuration. On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 To the right of **Constraint setup**, click the **Edit** button to open the **Constraint Specification** window.



- 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. To create a new template, click . The software compiles your project and creates a template. The new template is stored in a file *Module_number_Project_name_drs_template.xml* in your project folder.
- 4 Specify your constraints and save the template as an XML file. For more information, see “External Constraints for Polyspace Analysis” on page 6-7.
- 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.

Command Line

Use the option `Constraint setup (-data-range-specifications)` to specify the constraints XML file.

To specify constraints in the XML file:

- 1 First, create a blank XML template. The template lists all global variables, function inputs and modifiable arguments and return values of stubbed functions without specifying any constraints on them.

To create a blank template, run an analysis only up to the compilation phase. In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`. In Code Prover, check for source compliance only. Use the argument `compile` for the option `Verification level (-to)`. After the analysis, a blank template XML *drs-template.xml* is created in the results folder.

For C++ projects, to create a blank constraints template, you have to use the argument `cpp-normalize` for the option `Verification level (-to)`.

- 2 Edit the XML file to specify your constraints.


For examples, see:

- “Constrain Global Variable Range” on page 6-13
- “Constrain Function Inputs” on page 6-16

Create Constraint Template from Code Prover Analysis Results

You can constrain variable ranges based on their expected range in real-world applications. For instance, if a variable represents vehicle speed, you can set a maximum possible value. You can also constrain variable ranges only if they cause too many orange checks from overapproximation.

A Code Prover analysis shows all global variables, function inputs and stubbed functions that lead to orange checks from possible overapproximation. You can constrain only these variables for a more precise analysis.

- 1 Open Code Prover results in the Polyspace user interface or Polyspace Access web interface.
- 2 Open the **Orange Sources** pane. Do one of the following:
 - Select an orange check. If the software can trace an orange check to a root cause, a  icon appears on the **Result Details** pane. Click this icon to open the **Orange Sources** pane.
 - In the Polyspace user interface, select **Window > Show/Hide View > Orange Sources**. In the Polyspace Access web interface, select **Layout > Show/Hide View > Orange Sources**.

You see the full list of variables (function inputs or return values of stubbed functions) that can cause orange checks. Constrain the ranges of these variables.

In the details for individual orange checks, you often see a message similar to this:

```
If appropriate, applying DRS to stubbed function random_float in example.c  
line 44 may remove this orange.
```

The message is an indication that the stubbed function is a possible source of the orange check. You can apply external constraints on the function to enforce more precise assumptions and possibly remove the orange check (in case it came from the broader assumptions).


Update Existing Template

With new code submissions, you might have to specify additional constraints. You can update an existing template to add global variables, function inputs and stubbed functions that come from the new code submissions.

Additionally, 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.

User Interface (Desktop Products Only)

- 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.

Command Line

In a continuous integration workflow, you can use the constraints XML file from the previous run. If new code submissions require additional constraints:

- 1 Specify constraints on variables from new code submissions in a constraints XML file. See Create Constraint Template: Command Line on page 6-3.
- 2 Merge the constraints XML file with the new constraints and the constraints XML file from the previous run.

Specify Constraints in Code

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

To specify constraints *inside* your code, you can use:

- Appropriate error handling tests in your code.

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

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

Polyspace checks your `assert` statements to see if the condition can be false. Following the `assert` statement, Polyspace considers that the `assert` condition is true. Using `assert` statements, you can constrain your variables for the remaining code in the same scope. For examples, see [User assertion \(Polyspace Code Prover\)](#).

See Also

Constraint setup (-data-range-specifications)

Related Examples

- “External Constraints for Polyspace Analysis” on page 6-7
- “Constrain Global Variable Range” on page 6-13
- “Constrain Function Inputs” on page 6-16
- “XML File Format for Constraints” on page 6-19

External Constraints for Polyspace Analysis

For a more precise analysis with Polyspace, you can specify external constraints on:

- Global Variables.
- User-defined Functions.

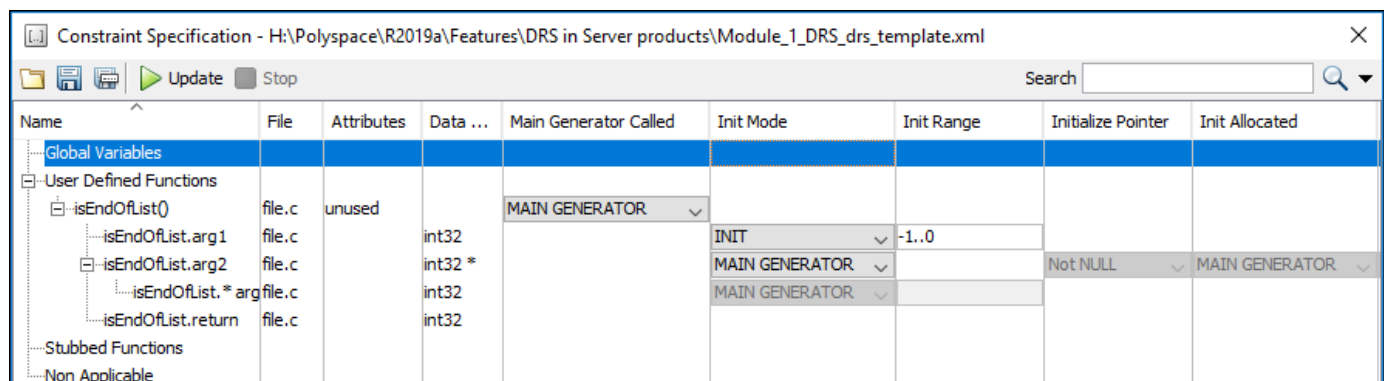
Constraints on user-defined functions do not apply to a Bug Finder analysis.

- Stubbed Functions.

Constraints on stubbed functions do not apply to a Bug Finder analysis.

For more information, see “Specify External Constraints” on page 6-2. For a partial list of limitations, see “Constraint Specification Limitations” on page 6-11.

In the user interface of the Polyspace desktop products, you can specify the constraints through a **Constraint Specification** window. The constraints are saved in an XML file that you can reuse for other projects.



The screenshot shows the 'Constraint Specification' window for a file named 'Module_1_DRS_drs_template.xml'. The window contains a table with the following columns: Name, File, Attributes, Data ..., Main Generator Called, Init Mode, Init Range, Initialize Pointer, and Init Allocated. The table is organized into sections: Global Variables, User Defined Functions, Stubbed Functions, and Non Applicable. Under 'User Defined Functions', there are several entries for the function 'isEndOfList()' and its arguments. The 'Main Generator Called' column for these entries is 'MAIN GENERATOR'. The 'Init Mode' column for 'isEndOfList()' is 'INIT', and for its arguments, it is 'MAIN GENERATOR'. The 'Init Range' column for 'isEndOfList()' is '-1..0', and for its arguments, it is empty. The 'Initialize Pointer' column for 'isEndOfList()' is 'Not NULL', and for its arguments, it is empty. The 'Init Allocated' column for 'isEndOfList()' is empty, and for its arguments, it is 'MAIN GENERATOR'.

Name	File	Attributes	Data ...	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated
Global Variables								
User Defined Functions								
isEndOfList()	file.c	unused		MAIN GENERATOR	INIT	-1..0		
isEndOfList.arg1	file.c		int32		MAIN GENERATOR		Not NULL	MAIN GENERATOR
isEndOfList.arg2	file.c		int32 *		MAIN GENERATOR			
isEndOfList.*arg	file.c		int32		MAIN GENERATOR			
isEndOfList.return	file.c		int32					
Stubbed Functions								
Non Applicable								

This table explains the various columns in the **Constraint Specification** window. If you directly edit the constraint XML file to specify a constraint (for instance, in the Polyspace Server products), this table also shows the correspondence between columns in the user interface and entries in the XML file. The XML entry highlighted in bold appears in the corresponding column of the **Constraint Specification** window.

Column	Settings
Name	<p>Displays the list of variables and functions in your Project for which you can specify data ranges.</p> <p>This Column displays three expandable menu items:</p> <ul style="list-style-type: none"> • Globals - Displays global variables in the project. • User defined functions - Displays user-defined functions in the project. Expand a function name to see its inputs. • Stubbed functions - Displays a list of stub functions in the project. Expand a function name to see the inputs and return values. <p>XML File Entry:</p> <pre><function name = "funcName" ...> <scalar name = "arg1" ...> <pointer name = "arg2" ...></pre>
File	<p>Displays the name of the source file containing the variable or function.</p> <p>XML File Entry:</p> <pre><file name = "C:\Project1\Sources\file.c" ...></pre>
Attributes	<p>Displays information about the variable or function.</p> <p>For example, static variables display <code>static</code>. Uncalled functions display <code>unused</code>.</p> <p>XML File Entry:</p> <pre><function name="funcName" attributes="unused" ...></pre>
Data Type	<p>Displays the variable type.</p> <p>XML File Entry:</p> <pre><scalar name="arg1" complete_type="int32" ...></pre>
Main Generator Called	<p>Applicable only for user-defined functions.</p> <p>Specifies whether the main generator calls the function:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Main generator may call this function, depending on the value of the <code>-functions-called-in-loop</code> (C) or <code>-main-generator-calls</code> (C++) parameter. • NO - Main generator will not call this function. • YES - Main generator will call this function. <p>XML File Entry:</p> <pre><function name="funcName" main_generator_called="MAIN_GENERATOR" ...></pre>

Column	Settings
Init Mode	<p>Specifies how the software assigns a range to the variable:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Variable range is assigned depending on the settings of the main generator options <code>-main-generator-writes-variables</code> and <code>-no-def-init-glob</code>. • IGNORE - Variable is not assigned to any range, even if a range is specified. • INIT - Variable is assigned to the specified range only at initialization, and keeps the range until first write. • PERMANENT - Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the <code>globalassert</code> mode if you need a warning. <p>User-defined functions support only INIT mode.</p> <p>Stub functions support only PERMANENT mode.</p> <p>For C verifications, global pointers support MAIN GENERATOR, IGNORE, or INIT mode.</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Pointer follows the options of the main generator. • IGNORE - Pointer is not initialized • INIT - Specify if the pointer is NULL, and how the pointed object is allocated (Initialize Pointer and Init Allocated options). <p>XML File Entry:</p> <pre><scalar name="arg1" init_mode="INIT" ...></pre>
Init Range	<p>Specifies the minimum and maximum values for the variable.</p> <p>You can use the keywords <code>min</code> and <code>max</code> to denote the minimum and maximum values of the variable type. For example, for the type <code>long</code>, <code>min</code> and <code>max</code> correspond to -2^{31} and $2^{31}-1$ respectively.</p> <p>You can also use hexadecimal values. For example: <code>0x12..0x100</code></p> <p>For <code>enum</code> variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.</p> <p>For <code>enum</code> variables, you can also use the keywords <code>enum_min</code> and <code>enum_max</code> to denote the minimum and maximum values that the variable can take. For example, for an <code>enum</code> variable of the type defined below, <code>enum_min</code> is 0 and <code>enum_max</code> is 5:</p> <pre>enum week{ sunday, monday=0, tuesday, wednesday, thursday, friday, saturday};</pre> <p>XML File Entry:</p> <pre><scalar name="arg1" init_range="-1..0" ...></pre>

Column	Settings
Initialize Pointer	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies whether the pointer should be NULL:</p> <ul style="list-style-type: none"> • May-be NULL - The pointer could potentially be a NULL pointer (or not). • Not Null - The pointer is never initialized as a null pointer. • Null - The pointer is initialized as NULL. <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 6-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" initialize_pointer="Not NULL" ...></pre>
Init Allocated	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies how the pointed object is allocated:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - The pointed object is allocated by the main generator. • None - Pointed object is not written. • SINGLE - Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.) • MULTI - All objects (or array elements) are initialized. <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 6-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" init_pointed="MAIN_GENERATOR" ...></pre>

Column	Settings
# Allocated Objects	<p>Applicable only to pointers.</p> <p>Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).</p> <p>The Init Allocated parameter specifies how many allocated objects are actually initialized. For instance, consider this code:</p> <pre>void func(int *ptr) { assert(ptr[0]==1); assert(ptr[1]==1); }</pre> <p>If you specify these constraints:</p> <ul style="list-style-type: none"> ptr has Init Allocated set to MULTI and # Allocated Objects set to 2, *ptr has Init Range set to 1..1, <p>both assertions are green. However, if you specify these constraints:</p> <ul style="list-style-type: none"> ptr has Init Allocated set to SINGLE *ptr has Init Range set to 1..1, <p>the second assertion is orange. Only the first object that ptr points to initialized to 1. Objects beyond the first can be potentially full range.</p> <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 6-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" number_allocated="10" ...></pre>
Global Assert	<p>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" global_assert="YES" ...></pre>
Global Assert Range	<p>Specifies the minimum and maximum values for the range you want to check.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" assert_range="0..200" ...></pre>
Comment	<p>Remarks that you enter, for example, justification for your DRS values.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" comment="Speed Range" ...></pre>

Constraint Specification Limitations

You cannot specify these constraints:

- *C++ Pointers cannot be constrained:*

In C++, you cannot constrain pointer arguments of functions. Functions that have pointer arguments only do not appear in the constraint specification interface.

Because of polymorphism, a C++ pointer can point to objects of multiple classes in a class hierarchy and can require invoking different constructors. The pre-analysis for constraint specification cannot determine which object type to constrain or which constructor to call.

- *Constraints cannot be relations:*

You cannot specify a constraint that relates the return value of a function to its inputs. You can only specify a constant range for the constraints.

- *Multiple ranges not possible:*

You cannot specify multiple ranges for a constraint. For instance, you cannot specify that a function argument has either the value -1 or a value in the range [1,100]. Instead, specify the range [-1,100] or perform two separate analyses, once with the value -1 and once with the range [1,100].

See Also

More About


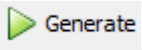
- “Specify External Constraints” on page 6-2

Constrain Global Variable Range

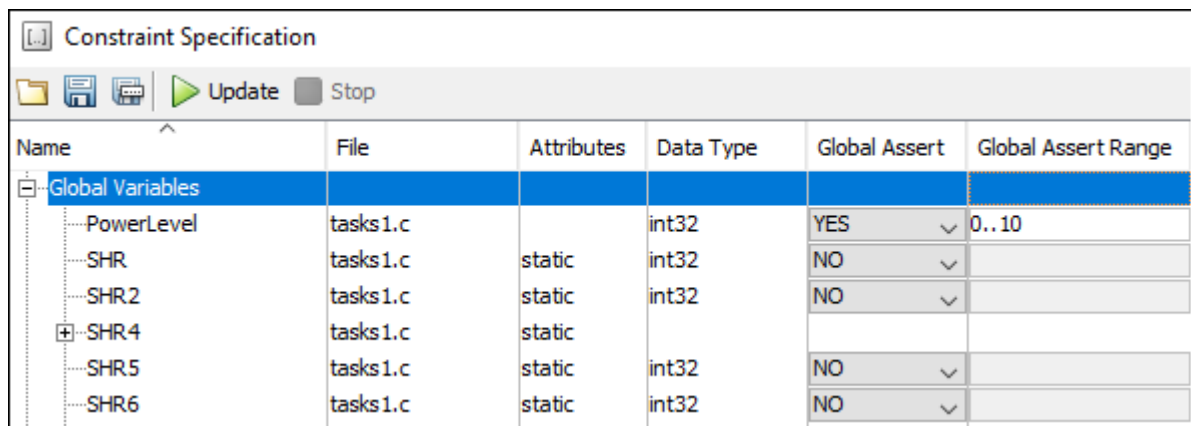
You can impose constraints (also known as data range specifications or DRS) on the range of a global variable and check with Code Prover whether write operations on the variable violate the constraint. For the general workflow, see “Specify External Constraints” on page 6-2.

User Interface (Desktop Products Only)


To constrain a global variable range and also check for violation of the constraint:

- 1 In your project configuration, select **Inputs & Stubbing**. Click the  button next to the **Constraint setup** field.
- 2 In the Constraint Specification window, click .

Under the **Global Variables** node, you see a list of global variables.



Name	File	Attributes	Data Type	Global Assert	Global Assert Range
Global Variables					
PowerLevel	tasks1.c		int32	YES	0..10
SHR	tasks1.c	static	int32	NO	
SHR2	tasks1.c	static	int32	NO	
SHR4	tasks1.c	static			
SHR5	tasks1.c	static	int32	NO	
SHR6	tasks1.c	static	int32	NO	

- 3 For the global variable that you want to constrain:
 - From the drop-down list in the **Global Assert** column, select YES.
 - In the **Global Assert Range** column, enter the range in the format *min*..*max*. *min* is the minimum value and *max* the maximum value for the global variable.
 - 4 To save your specifications, click the  button.
- In **Save a Constraint File** window, save your entries as an xml file.
- 5 Run a verification and open the results.

For every write operation on the global variable, you see a green, orange, or red **Correctness condition** check. If the check is:

- Green, the variable is within the range that you specified.
- Orange, the variable can be outside the range that you specified.
- Red, the variable is outside the range that you specified.

When two or more tasks write to the same global variable, the **Correctness condition** check can appear orange on all write operations to the variable even when only one write operation takes the variable outside the **Global Assert** range.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints” on page 6-2. In the XML file, locate and constrain the global variables. XML tags for global variables appear directly within the `file` tag without an enclosing function tag. For instance, in this constraint XML, `PowerLevel` and `SHR` are global variables:

```
<file name="\\\\home\\Polyspace_Workspace\\Examples\\Code_Prover_Example
          \\sources\\tasks1.c">
  <scalar name="PowerLevel" line="26" .. global_assert="YES" assert_range="0..10"/>
  <scalar name="SHR" line="30" ... global_assert="NO" assert_range="" />
  <function name="Tserver" line="73" .../>
  <function name="initregulate" line="47" .../>
  <function name="orderregulate" line="35" ...>
    <scalar name="return" ... global_assert="unsupported" assert_range="unsupported" />
  </function>
  <function name="procl" line="101" .../>
</file>
```

To specify a constraint on a global variable and check during a Code Prover analysis if the constraint is violated:

- 1 Set the `global_assert` attribute of the variable's `scalar` tag to `YES`.
- 2 Set the `assert_range` attribute to a range in the form `min..max`, for instance, `0..10`.

In the preceding example, the variable `PowerLevel` is constrained this way.

See Also

Polyspace Analysis Options

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

Polyspace Results

Correctness condition

More About

- “Specify External Constraints” on page 6-2
- “External Constraints for Polyspace Analysis” on page 6-7
- “Constrain Function Inputs” on page 6-16

Constrain Function Inputs


For a more precise Code Prover analysis, you can specify constraints (also known as data range specifications or DRS) on function inputs. Code Prover checks your function definition for run-time errors with respect to the constrained inputs. For the general workflow, see “Specify External Constraints” on page 6-2.

For instance, for a function defined as follows, you can specify that the argument `val` has values in the range `[1..10]`. You can also specify that the argument `ptr` points to a 3-element array where each element is initialized:

```
int func(int val, int* ptr) {
    .
    .
}
```

User Interface (Desktop Products Only)

To specify constraints on function inputs:

- 1 In your project configuration, select **Inputs & Stubbing**. Click the  button for **Constraint setup**.

- 2 In the Constraint Specification window, click .

Under the **User Defined Functions** node, you see a list of functions whose inputs can be constrained.

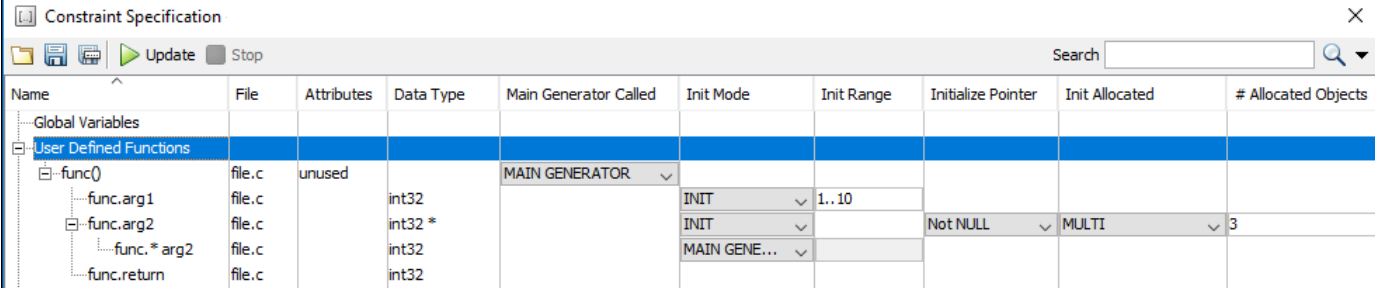
- 3 Expand the node for each function.

You see each function input on a separate row. The inputs have the syntax `function_name.arg1`, `function_name.arg2`, etc.

- 4 Specify your constraints on one or more of the function inputs. For more information, see “External Constraints for Polyspace Analysis” on page 6-7.

For example, in the preceding code:

- To constrain `val` to the range `[1..10]`, select **INIT** for **Init Mode** and enter `1..10` for **Init Range**.
- To specify that `ptr` points to a 3-element array where each element is initialized, select **MULTI** for **Init Allocated** and enter 3 for **# Allocated Objects**.



Name	File	Attributes	Data Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects
Global Variables									
User Defined Functions									
func()	file.c	unused		MAIN GENERATOR					
func.arg1	file.c		int32		INIT	1..10			
func.arg2	file.c		int32 *		INIT		Not NULL	MULTI	3
func.*arg2	file.c		int32		MAIN GENERATOR				
func.return	file.c		int32						

- Run verification and open the results. On the **Source** pane, place your cursor on the function inputs.

The tooltips display the constraints. For example, in the preceding code, the tooltip displays that `val` has values in `1..10`.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints” on page 6-2. In the XML file, locate and constrain the function inputs. The function inputs appear as a scalar or pointer tag in a function tag. The inputs are named as `arg1`, `arg2` and so on. For instance, for the preceding code, the XML structure for the inputs of `func` appear as follows:

```
<function name="func" line="1" attributes="unused"
  main_generator_called="MAIN_GENERATOR" comment="">
  <scalar name="arg1" line="1" base_type="int32"
    complete_type="int32" init_mode="INIT" init_range="1..10" />
  <pointer name="arg2" line="1" complete_type="int32 *"
    init_mode="INIT" initialize_pointer="Not NULL" number_allocated="3"
    init_pointed="MULTI">
    <scalar line="1" base_type="int32" complete_type="int32"
      init_mode="MAIN_GENERATOR" init_range="" />
  </pointer>
  <scalar name="return" line="1" base_type="int32" complete_type="int32"
    init_mode="disabled" init_range="disabled" />
</function>
```

To specify a constraint on a function input, set the attributes `init_mode` and `init_range` for scalar variables, and `init_pointed` and `number_allocated` for pointer variables.

- To constrain `val` to the range `[1..10]`, set the `init_mode` attribute of the tag with name `arg1` to `INIT` and `init_range` to `1..10`.

- To specify that `ptr` points to a 3-element array where each element is initialized, set the `init_mode` attribute of the tag with name `arg2` to `INIT`, `init_pointed` to `MULTI` and `number_allocated` to 3.

See Also

Constraint setup (-data-range-specifications)

More About

- “Specify External Constraints” on page 6-2
- “External Constraints for Polyspace Analysis” on page 6-7
- “Constrain Global Variable Range” on page 6-13

XML File Format for Constraints

For a more precise Polyspace analysis, you can specify constraints on global variables, function inputs and stubbed functions. You can specify the constraints in the user interface of the Polyspace desktop products or at the command line as an XML file. For the general workflow, see “Specify External Constraints” on page 6-2.

This topic describes details of the constraint XML file schema. You typically require this information only if you create a constraint XML from scratch. If you run a verification once, the software automatically generates a template constraint file `drs-template.xml` in your results folder. Instead of creating a constraint XML file from scratch, it is easier to edit this template XML file to specify your constraints. For some examples, see:

- “Constrain Global Variable Range” on page 6-13
- “Constrain Function Inputs” on page 6-16

For another explanation of what the XML tags mean, see “External Constraints for Polyspace Analysis” on page 6-7.

You can also see the information in this topic and the underlying XML schema in `polyspaceroot\polyspace\drs`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Syntax Description — XML Elements

The constraints file contains the following XML elements:

- `<global>` element — Declares the global scope, and is the root element of the XML file.
- `<file>` element — Declares a file scope. Must be enclosed in the `<global>` element. May enclose any variable or function declaration. Static variables must be enclosed in a file element to avoid conflicts.
- `<scalar>` element— Declares an integer or a floating point variable. May be enclosed in any recognized element, but cannot enclose any element. Sets `init/permanent/global` asserts on variables.
- `<pointer>` element — Declares a pointer variable. May enclose any other variable declarations (including itself), to define the pointed objects. Specifies what value is written into pointer (NULL or not), how many objects are allocated and how the pointed objects are initialized.
- `<array>` element — Declares an array variable. May enclose any other variable definition (including itself), to define the members of the array.
- `<struct>` element — Declares a structure variable or object (instance of class). May enclose any other variable definition (including itself), to define the fields of the structure.
- `<function>` element — Declares a function or class method scope. May enclose any variable definition, to define the arguments and the return value of the function. Arguments should be named `arg1`, `arg2`, ..., `argn` and the return value should be called `return`.

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

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field `line` contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the

GUI to compute the min and max values. The field comment is used to add information about any node.

- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a struct field.
- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:
 - **1**: The mode “NO” is allowed.
 - **2**: The mode “INIT” is allowed.
 - **4**: The mode “PERMANENT” is allowed.
 - **8**: The mode “MAIN_GENERATOR” is allowed.

For example, the value “**10**” means that modes “INIT” and “MAIN_GENERATOR” are allowed. To see how this value is computed, refer to “Valid Modes and Default Values” on page 6-23.

- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to `SINGLE`, `MULTI`, `SINGLE_CERTAIN_WRITE` or `MULTI_CERTAIN_WRITE`.
- **(*****)** — `SINGLE_CERTAIN_WRITE` or `MULTI_CERTAIN_WRITE` are available for parameters and return values of stubbed functions only if they are pointers. If the parameter or return value is a structure and the structure has a pointer field, they are also available for the pointer field.

<file> Element

Field	Syntax
name	<i>filepath_or_filename</i>
comment	<i>string</i>

<scalar> Element

Field	Syntax
name (**)	<i>name</i>
line (*)	<i>line</i>
base_type (*)	intx uintx floatx
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>

Field	Syntax
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
init_range	<i>range</i> disabled unsupported
global_assert	YES NO disabled unsupported
assert_range	<i>range</i> disabled unsupported
comment(*)	<i>string</i>

<pointer> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
initialize_pointer	May be: NULL Not NULL NULL
number_allocated	<i>single value</i> disabled unsupported

Field	Syntax
init_pointed (*****)	MAIN_GENERATOR NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE disabled
comment	<i>string</i>

<array> and <struct> Elements

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

<function> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	<i>string</i>

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/static/const scalar	MAIN_GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependent
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT PERMANENT	YES NO			INIT min..max
	Struct	Struct field	Refer to field type				
	Array	Array element	Refer to element type				
Global variables	Pointer	Unqualified/static/const scalar	MAIN_GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependent
		Volatile pointer	un-supported		un-supported	un-supported	
		Extern pointer	IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Pointed volatile scalar	un-supported	un-supported			
		Pointed extern scalar	INIT	un-supported			INIT min..max
		Pointed other scalars	MAIN_GENERATOR INIT	un-supported			MAIN_GENERATOR dependent
		Pointed pointer	MAIN_GENERATOR INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_GENERATOR dependent
		Pointed function	un-supported	un-supported			
Function parameters	Userdef function	Scalar parameters	MAIN_GENERATOR INIT	un-supported			INIT min..max

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default	
		Pointer parameters	MAIN_GENERATOR INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI	
		Other parameters	Refer to parameter type					
	Stubbed function	Scalar parameter	disabled		un-supported			
		Pointer parameters	disabled			disabled	NONE SINGLE MULTI SINGLE_CERTAIN_ WRITE MULTI_CERTAIN_ WRITE	MULTI
		Pointed parameters	PERMANENT		un-supported			PERMANENT min..max
		Pointed const parameters	disabled		un-supported			
Function return	Userdef function	Return	disabled	un-supported	disabled	disabled		
	Stubbed function	Scalar return	PERMANENT	un-supported			PERMANENT min..max	
		Pointer return	PERMANENT		un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI SINGLE_CERTAIN_ WRITE MULTI_CERTAIN_ WRITE	PERMANENT May be NULL max MULTI

See Also

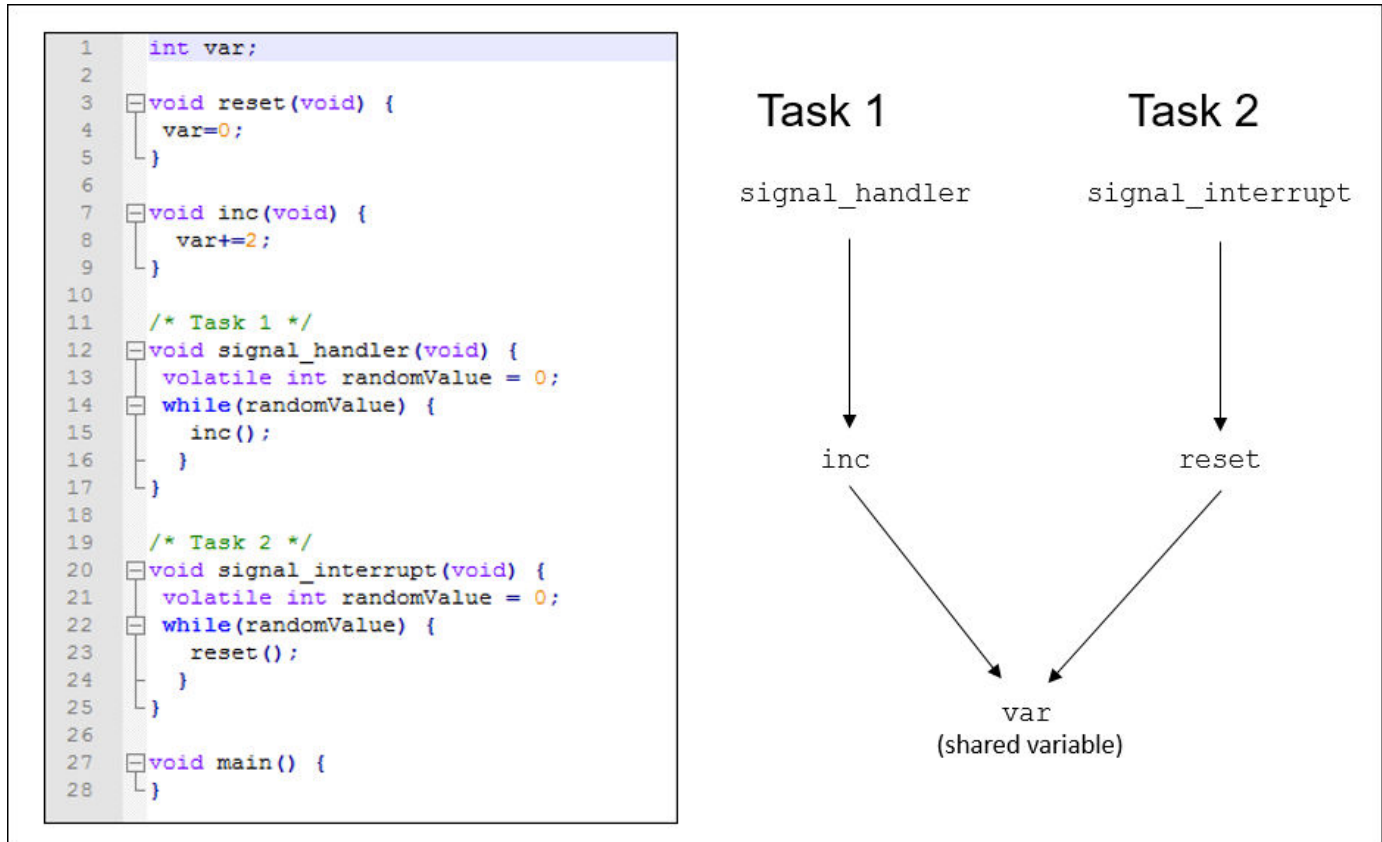
More About

- “Specify External Constraints” on page 6-2
- “Constrain Global Variable Range” on page 6-13
- “Constrain Function Inputs” on page 6-16

Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace

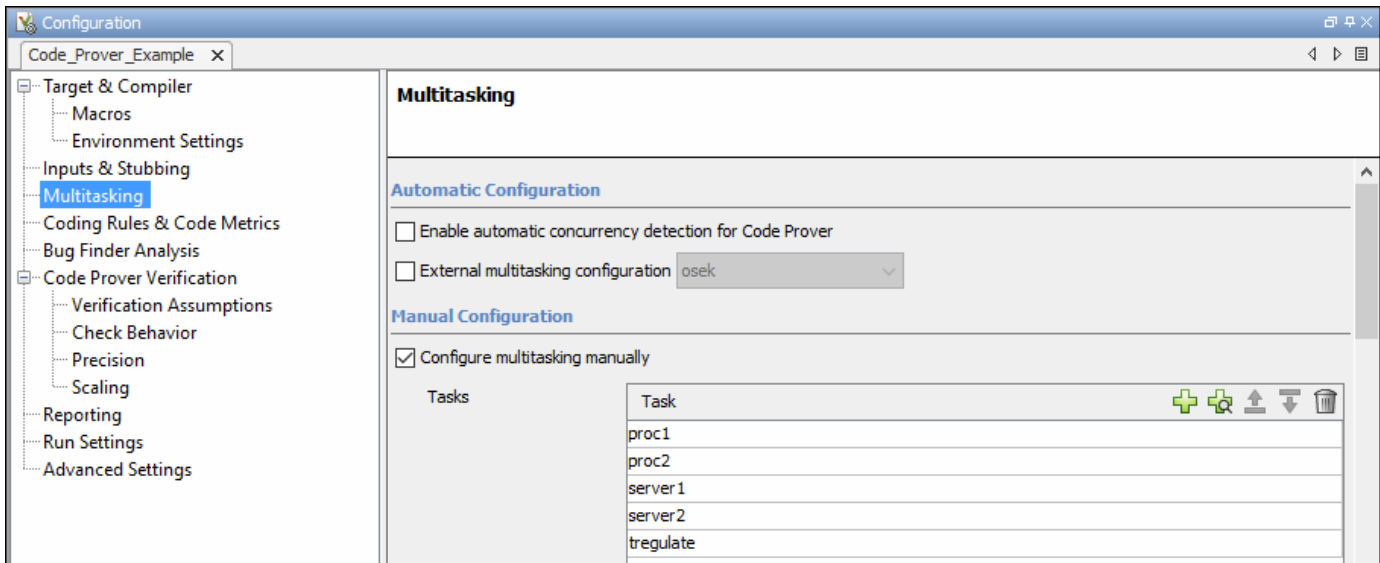
With Polyspace, you can analyze programs where multiple threads (tasks) run concurrently.



In addition to regular run-time checks, the analysis looks for issues specific to concurrent execution:

- Data races, deadlocks, consecutive or missing locks and unlocks (Bug Finder)
- Unprotected shared variables (Code Prover)

Configure Analysis



If your code uses multitasking primitives from certain families, for instance, `pthread_create` for thread creation:

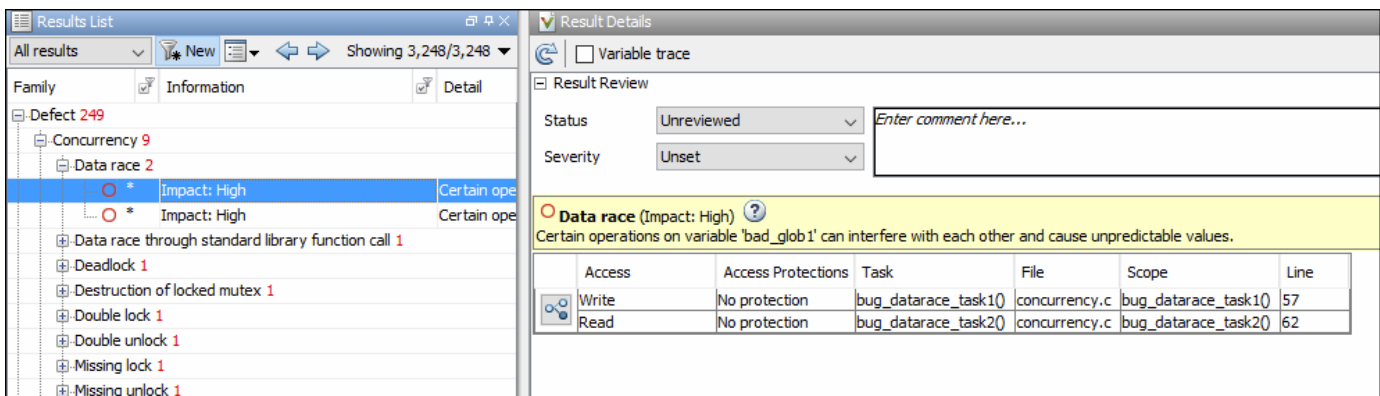
- In Bug Finder, the analysis detects them and extracts your multitasking model from the code.
- In Code Prover, you must enable this automatic detection explicitly.

See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-5.

Alternatively, define your multitasking model through the analysis options. In the user interface, the options are on the **Multitasking** node in the **Configuration** pane. For more information, see “Configuring Polyspace Multitasking Analysis Manually” on page 7-16.

Review Analysis Results

Bug Finder




The Bug Finder analysis shows concurrency defects such as data races and deadlocks. See “Concurrency Defects” (Polyspace Bug Finder Access).

Code Prover

The screenshot displays the Polyspace Code Prover interface. The left pane, titled "Results List", shows a tree view of analysis results. Under "Global Variable", the "Potentially unprotected variable" category is expanded, listing variables like PowerLevel, SHR4, and SHR2. The right pane, titled "Result Details", shows the details for a "Potentially unprotected variable" defect. It includes a "Result Review" section with "Status" set to "Unreviewed" and "Severity" set to "Unset". Below this, a yellow warning box states: "Variable 'tasks1.PowerLevel' is shared among several tasks. Some operations on variable 'tasks1.PowerLevel' have no common protection. Read by task: server1 server2 tregulate. Written by task: server1 server2 tregulate." A table below the warning lists the conflicting operations:

Event	File	Scope	Line
Written value: -10000	main.c	main()	36
Written value: 0	tasks1.c	_init_globals()	26
Written value: [-2147483639 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks1.c	orderregulate()	40
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Compute_Injection()	34
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Get_PowerLevel()	41

The Code Prover analysis exhaustively checks if shared global variables are protected from concurrent access. See “Global Variables” (Polyspace Code Prover Access).

Review the results using the message on the **Result Details** pane. See a visual representation of conflicting operations using the  (graph) icon.

See Also

More About

- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-5
- “Configuring Polyspace Multitasking Analysis Manually” on page 7-16
- “Protections for Shared Variables in Multitasking Code” on page 7-20

Auto-Detection of Thread Creation and Critical Section in Polyspace

With Polyspace, you can analyze programs where multiple threads run concurrently. Polyspace can analyze your multitasking code for data races, deadlocks and other concurrency defects, if the analysis is aware of the concurrency model in your code. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. Bug Finder detects them by default. In Code Prover, you enable automatic detection using the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 7-2.

If your thread creation function is not detected automatically:

- You can also map the function to a thread-creation function that Polyspace can detect automatically. Use the option `-code-behavior-specifications`.
- Otherwise, you must manually model your multitasking threads by using configuration options. See “Configuring Polyspace Multitasking Analysis Manually” on page 7-16.

Multitasking Routines that Polyspace Can Detect

Polyspace can detect thread creation and critical sections if you use primitives from these groups. Polyspace recognizes calls to these routines as the creation of a new thread or as the beginning or end of a critical section.

POSIX

Thread creation: `pthread_create`

Critical section begins: `pthread_mutex_lock`

Critical section ends: `pthread_mutex_unlock`

VxWorks

Thread creation: `taskSpawn`

Critical section begins: `semTake`

Critical section ends: `semGive`

To activate automatic detection of concurrency primitives for VxWorks®, in the user interface of the Polyspace desktop products, use the VxWorks template. For more information on templates, see “Create Project Using Configuration Template” (Polyspace Code Prover). At the command-line, use these options:

```
-D1=CPU=I80386  
-D2=__GNUC__=2  
-D3=__OS_VXWORKS
```

Concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do one of the following:

- Provide a `main` function.
- Preprocessor definitions (-D): In preprocessor definitions, set `vxworks_entry_point=main`.

Windows

Thread creation: `CreateThread`

Critical section begins: `EnterCriticalSection`

Critical section ends: `LeaveCriticalSection`

µC/OS II

Thread creation: `OSTaskCreate`

Critical section begins: `OSMutexPend`

Critical section ends: `OSMutexPost`

C++11

Thread creation: `std::thread::thread`

Critical section begins: `std::mutex::lock`

Critical section ends: `std::mutex::unlock`

For autodetection of C++11 threads, explicitly specify paths to your compiler header files or use `polyspace-configure`.

For instance, if you use `std::thread` for thread creation, explicitly specify the path to the folder containing `thread.h`.

See also “Limitations of Automatic Thread Detection” on page 7-11.

C11

Thread creation: `thr_create`

Critical section begins: `mtx_lock`

Critical section ends: `mtx_unlock`

Example of Automatic Thread Detection

The following multitasking code models five philosophers sharing five forks. The example uses POSIX[®] thread creation routines and illustrates a classic example of a deadlock. Run Bug Finder on this code to see the deadlock.

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

pthread_mutex_t forks[5];

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

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

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

```

        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 3 puts down left fork\n");
    }
    return NULL;
}

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

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

int main(void)
{
    pthread_t ph[5];
    pthread_create(&ph[0], NULL, philo1, NULL);
    pthread_create(&ph[1], NULL, philo2, NULL);
    pthread_create(&ph[2], NULL, philo3, NULL);
    pthread_create(&ph[3], NULL, philo4, NULL);
    pthread_create(&ph[4], NULL, philo5, NULL);

    pthread_join(ph[0], NULL);
    pthread_join(ph[1], NULL);
    pthread_join(ph[2], NULL);
    pthread_join(ph[3], NULL);
    pthread_join(ph[4], NULL);
}

```

```
    return 1;
}
```

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

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

Naming Convention for Automatically Detected Threads

If you use a function such as `pthread_create()` to create new threads (tasks), each thread is associated with a unique identifier. For instance, in this example, two threads are created with identifiers `id1` and `id2`.

```
pthread_t* id1, id2;

void main()
{
    pthread_create(id1, NULL, start_routine, NULL);
    pthread_create(id2, NULL, start_routine, NULL);
}
```

If a data race occurs between the threads, the analysis can detect it. When displaying the results, the threads are indicated as `task_id`, where `id` is the identifier associated with the thread. In the preceding example, the threads are identified as `task_id1` and `task_id2`.

If a thread identifier is:

- Local to a function, the thread name shows the function.

For instance, the thread created below appears as `task_f:id`

```
void f(void)
{
    pthread_t* id;
    pthread_create(id, NULL, start_routine, NULL);
}
```

- A field of a structure, the thread name shows the structure.

For instance, the thread created below appears as `task_a#id`

```
struct {pthread_t* id; int x;} a;
pthread_create(a.id, NULL, start_routine, NULL);
```

- An array member, the thread name shows the array.

For instance, the thread created below appears as `task_tab[1]`.

```
pthread_t* tab[10];
pthread_create(tab[1],NULL,start_routine,NULL);
```

If you create two threads with distinct thread identifiers, but you use the same local variable name for the thread identifiers, the name of the second thread is modified to distinguish it from the first thread. For instance, the threads below appear as `task_func:id` and `task_func:id:1`.

```
void func()
{
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);
    }
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);
    }
}
```

Limitations of Automatic Thread Detection

The multitasking model extracted by Polyspace does not include some features. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace.
- Recursive semaphores.
- Unbounded thread identifiers, such as `extern pthread_t ids[]` — Warning.
- Calls to concurrency primitive through high-order calls — Warning.
- Aliases on thread identifiers — Polyspace over-approximates when the alias is used.
- Termination of threads — Polyspace ignores `pthread_join` and `thrd_join`. Polyspace replaces `pthread_exit` and `thrd_exit` by a standard `exit`.
- (Polyspace Bug Finder only) Creation of multiple threads through multiple calls to the same function with different pointer arguments.

Example

In this example, Polyspace considers that only one thread is created.

```
pthread_t id1, id2;
void start(pthread_t* id)
{
    pthread_create(id, NULL, start_routine, NULL);
}
void main()
{
    start(&id1);
    start(&id2);
}
```

- (Polyspace Code Prover only) Shared local variables — Only global variables are considered shared. If a local variable is accessed by multiple threads, the analysis does not take into account the shared nature of the variable.

Example

In this example, the analysis does not take into account that the local variable `x` can be accessed by both `task1` and `task2` (after the new thread is created).

```
#include <pthread.h>
#include <stdlib.h>

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    int x;
    x = 2;
    pthread_t id;
    (void)pthread_create(&id, NULL, task2, (void*) &x);
    /* x (local var) passed to task2 */
    x = 3 ;

    /* Unknown thread priority means x = 1 OR x = 3.*/
    /* However, the analysis considers x = 3 */
    /* Assertion below is green */
    assert(x == 3);
}

int main(void)
{
    task1();
    return 0;
}
```

- (Polyspace Code Prover only) Shared dynamic memory — Only global variables are considered shared. If a dynamically allocated memory region is accessed by multiple threads, the analysis does not take into account its shared nature.

Example

In this example, the analysis does not take into account that `lx` points to a shared memory region. The region can be accessed by both `task1` and `task2` (after the new thread is created). The Code Prover analysis also reports `lx` as a non-shared variable.


```

#include <pthread.h>
#include <stdlib.h>

static int* lx;

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    pthread_t id;
    lx = (int*)malloc(sizeof(int));

    if (lx == NULL) exit(1);

    (void)pthread_create(&id, NULL, task2, (void*) lx);

    *lx = 3 ;

    /* Unknown thread priority means *lx = 1 OR *lx = 3.*/
    /* However, the analysis considers *lx = 3 */
    /* Assertion below is green */
    assert(*lx == 3);
}

int main(void)
{
    task1();
    return 0;
}

```

- Number of tasks created with CreateThread when threadId is set to NULL— When you create multiple threads that execute the same function, if the last argument of CreateThread is NULL, Polyspace only detects one instance of this function, or task.

Example

In this example, Polyspace detects only one instance of `thread_function1()`, but 10 instances of `thread_function2()`.

```

#include <windows.h>

#define MAX_LOOP_THREADS 10

DWORD WINAPI thread_function1(LPVOID data) {}
DWORD WINAPI thread_function2(LPVOID data) {}

HANDLE hds1[MAX_LOOP_THREADS];
HANDLE hds2[MAX_LOOP_THREADS];
DWORD threadId[MAX_LOOP_THREADS];

int main(void)
{
    for (int i = 0; i < MAX_LOOP_THREADS; i++) {
        hds1[i] = CreateThread(NULL, 0, thread_function1, NULL, 0, NULL);
        hds2[i] = CreateThread(NULL, 0, thread_function2, NULL, 0, &threadId[i]);
    }

    return 0;
}

```

- (C++11 only) If you use lambda expressions as start functions during thread creation, Polyspace does not detect shared variables in the lambda expressions.

Example

In this example, Polyspace does not detect that the variable `y` used in the lambda expressions is shared between two threads. As a result, Bug Finder, for instance, does not show a **Data race** defect.

```

#include <thread>
int y;
int main() {
    std::thread t1([] {y++;});
    std::thread t2([] {y++;});
    t1.join();
    t2.join();
    return 0;
}

```

- (C++11 threads with Polyspace Code Prover only) String literals as thread function argument — Code Prover shows a red **Illegally dereferenced pointer** error if the thread function has an `std::string&` parameter and you pass a string literal argument.

Example

In this example, the thread function `foo` has an `std::string&` parameter. When starting a thread, a string literal is passed as argument to this function, which undergoes an implicit conversion to `std::string` type. Code Prover loses track of the original string literal in this conversion. Therefore, a dashed red underline appears on `operator<<` in the body of `foo` and a red **Illegally dereferenced pointer** check in the body of `operator<<`.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::thread t1(foo,"foo_arg");
}
```

To work around this issue, assign the string literal to a temporary variable and pass the variable as argument to the thread function.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::string str = "foo_arg";
    std::thread t1(foo, str);
}
```

See Also

-code-behavior-specifications | Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Configuring Polyspace Multitasking Analysis Manually” on page 7-16

Configuring Polyspace Multitasking Analysis Manually

With Polyspace, you can analyze programs where multiple threads run concurrently. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-5.

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

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 7-2.

Specify Options for Multitasking Analysis

Use these options to specify cyclic tasks, interrupts and protections for shared variables. In the Polyspace user interface, the options are on the **Multitasking** node in the **Configuration** pane.

- **Entry points (-entry-points)**: Specify noncyclic entry point functions.
Do not specify `main`. Polyspace implicitly considers `main` as an entry point function.
- **Cyclic tasks (-cyclic-tasks)**: Specify functions that are scheduled at periodic intervals.
- **Interrupts (-interrupts)**: Specify functions that can run asynchronously.
- **Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)**: Specify functions that disable and reenable interrupts (Bug Finder only).
- **Critical section details (-critical-section-begin -critical-section-end)**: Specify functions that begin and end critical sections.
- **Temporally exclusive tasks (-temporal-exclusions-file)**: Specify groups of functions that are temporally exclusive.
- **-preemptable-interrupts**: Specify functions that have lower priority than interrupts, but higher priority than tasks (preemptable or non-preemptable).
Only the Bug Finder analysis considers priorities.
- **-non-preemptable-tasks**: Specify functions that have higher priority than tasks, but lower priority than interrupts (preemptable or non-preemptable).
Only the Bug Finder analysis considers priorities.

Adapt Code for Code Prover Multitasking Analysis

The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a strict model.

Tasks and interrupts must be void-void functions.

Functions that you specify as tasks and interrupts must have the prototype:

```
void func(void);
```

Suppose you want to specify a function `func` that takes `int` arguments and has return type `int`:

```
int func(int);
```

Define a wrapper void-void function that calls `func` with a volatile value. Specify this wrapper function as a task or interrupt.

```
void func_wrapper() {
    volatile int arg;
    (void)func(arg);
}
```

You can save the wrapper function definition along with a declaration of the original function in a separate file and add this file to the analysis.

The main function must end.

Code Prover assumes that the `main` function ends before all tasks and interrupts begin. If the `main` function contains an infinite loop or run-time error, the tasks and interrupts are not analyzed. If you see that there are no checks in your tasks and interrupts, look for a token underlined in dashed red to identify the issue in the `main` function. See “Reasons for Unchecked Code” (Polyspace Code Prover).

Suppose you want to specify the `main` function as a cyclic task.

```
void performTask1Cycle(void);
void performTask2Cycle(void);
```

```
void main() {
    while(1) {
        performTask1Cycle();
    }
}
```

```
void task2() {
    while(1) {
        performTask2Cycle();
    }
}
```

Replace the definition of `main` with:

```
#ifdef POLYSPACE
void main() {
}
void task1() {
    while(1) {
        performTask1Cycle();
    }
}

#else
void main() {
    while(1) {
        performTask1Cycle();
    }
}
#endif
```

The replacement defines an empty main and places the content of main into another function task1 if a macro POLYSPACE is defined. Define the macro POLYSPACE using the option Preprocessor definitions (-D) and specify task1 for the option Tasks (-entry-points).

This assumption does not apply to automatically detected threads. For instance, a main function can create threads using pthread_create.

All tasks and interrupts can interrupt each other.

The Bug Finder analysis considers priorities of tasks. A function that you specify as a task cannot interrupt a function that you specify as an interrupt because an interrupt has higher priority.

The Code Prover analysis considers that all tasks and interrupts can interrupt each other.

The Polyspace multitasking analysis assumes that a task or interrupt cannot interrupt itself.

All tasks and interrupts can run any number of times in any sequence.

The Code Prover analysis considers that all tasks and interrupts can run any number of times in any sequence.

Suppose in this example, you specify reset and inc as cyclic tasks. The analysis shows an overflow on the operation var+=2.

```
void reset(void) {
    var=0;
}

void inc(void) {
    var+=2;
}
```

Suppose you want to model a scheduling of tasks such that reset executes after inc has executed five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of reset and inc.

```
void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        inc();
        inc();
        inc();
        inc();
        inc();
        reset();
    }
}
```

Suppose you want to model a scheduling of tasks such that reset executes after inc has executed zero to five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of reset and inc.

```
void task() {
    volatile int randomValue = 0;
```

```
while(randomValue) {  
    if(randomValue)  
        inc();  
    if(randomValue)  
        inc();  
    if(randomValue)  
        inc();  
    if(randomValue)  
        inc();  
    if(randomValue)  
        inc();  
    if(randomValue)  
        inc();  
    reset();  
}  
}
```

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-5

Protections for Shared Variables in Multitasking Code

If your code is intended for multitasking, tasks in your code can access a common shared variable. To prevent data races, you can protect read and write operations on the variable. This topic shows the various protection mechanisms that Polyspace can recognize.

Detect Unprotected Access

The screenshot shows the Polyspace interface. On the left, the 'Results List' panel displays a hierarchy of defects: Defect 249, Concurrency 9, and Data race 2. The 'Data race 2' defect is selected, showing its details: 'Impact: High' and 'Certain operations'. On the right, the 'Result Details' panel shows the 'Data race (Impact: High)' description: 'Certain operations on variable 'bad_glob1' can interfere with each other and cause unpredictable values.' Below this is a table with the following data:

Access	Access Protections	Task	File	Scope	Line
Write	No protection	bug_datarace_task1()	concurrency.c	bug_datarace_task1()	57
Read	No protection	bug_datarace_task2()	concurrency.c	bug_datarace_task2()	62

You can detect an unprotected access using either Bug Finder or Code Prover. Code Prover is more exhaustive and proves if a shared variable is protected from concurrent access.

- Bug Finder detects an unprotected access using the result **Data race**. See [Data race](#).
- Code Prover detects an unprotected access using the result **Shared unprotected global variable**. See [Potentially unprotected variable](#).

Suppose you analyze this code, specifying `signal_handler_1` and `signal_handler_2` as cyclic tasks. Use the analysis option `Cyclic tasks (-cyclic-tasks)`.

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void signal_handler_1(void) {
    reset();
    inc();
    inc();
}

void signal_handler_2(void) {
    shared_var = INT_MAX;
}
```



```

}

void main() {
}

```

Bug Finder shows a data race on `shared_var`. Code Prover shows that `shared_var` is a potentially unprotected shared variable. Code Prover also shows that the operation `shared_var += 2` can overflow. The overflow occurs if the call to `inc` in `signal_handler_1` immediately follows the operation `shared_var = INT_MAX` in `signal_handler_2`.

Protect Using Critical Sections

One possible solution is to protect operations on shared variables using critical sections.

In the preceding example, modify your code so that operations on `shared_var` are in the same critical section. Use the functions `take_semaphore` and `give_semaphore` to begin and end the critical sections. To specify these functions that begin and end critical sections, use the analysis options `Critical section details (-critical-section-begin -critical-section-end)`.

```

#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

/* Declare lock and unlock functions */
void take_semaphore(void);
void give_semaphore(void);

void signal_handler_1() {
    /* Begin critical section */
    take_semaphore();
    reset();
    inc();
    inc();
    /* End critical section */
    give_semaphore();
}

void signal_handler_2() {
    /* Begin critical section */
    take_semaphore();
    shared_var = INT_MAX;
    /* End critical section */
    give_semaphore();
}

```

```
void main() {  
}
```

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-5.

Protect Using Temporally Exclusive Tasks

Another possible solution is to specify a group of tasks as temporally exclusive. Temporally exclusive tasks cannot interrupt each other.

In the preceding example, specify that `signal_handler_1` and `signal_handler_2` are temporally exclusive. Use the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

Protect Using Priorities

Another possible solution is to specify that one task has higher priority over another.

In the preceding example, specify that `signal_handler_1` is an interrupt. Retain `signal_handler_2` as a cyclic task. Use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`.

Bug Finder does not show the data race defect anymore. The reason is this:

- The operation `shared_var = INT_MAX` in `signal_handler_2` is atomic. Therefore, the operations in `signal_handler_1` cannot interrupt it.
- The operations in `signal_handler_1` cannot be interrupted by the operation in `signal_handler_2` because `signal_handler_1` has higher priority.

You can specify up to four different priorities with these options (with highest priority listed first):

- `Interrupts (-interrupts)`
- `-preemptable-interrupts`
- `-non-preemptable-tasks`
- `Cyclic tasks (-cyclic-tasks)`

A task with higher priority is atomic with respect to a task with lower priority. Note that the checker `Data race including atomic operations` ignores the difference in priorities and continues to show the data race. See also “Define Preemptable Interrupts and Nonpreemptable Tasks”.

Code Prover does not consider priorities of tasks. Therefore, Code Prover still shows `shared_var` as a potentially unprotected global variable.

Protect By Disabling Interrupts

In a Bug Finder analysis, you can protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Define Atomic Operations in Multitasking Code” on page 7-24

Define Atomic Operations in Multitasking Code

In code with multiple threads, you can use Polyspace Bug Finder to detect data races or Polyspace Code Prover to list potentially unprotected shared variables.

To determine if a variable shared between multiple threads is protected against concurrent access, Polyspace checks if the operations on the variable are atomic.

Nonatomic Operations

If an operation is nonatomic, Polyspace considers that the operation involves multiple steps. These steps do not need to occur together and can be interrupted by operations in other threads.

For instance, consider these two operations in two different threads:

- Thread 1: `var++;`

This operation is nonatomic because it takes place in three steps: reading `var`, incrementing `var`, and writing back `var`.

- Thread 2: `var = 0;`

This operation is atomic if the size of `var` is less than the word size on the target. See details below for how Polyspace determines the word size.

If the two operations are not protected (by using, for instance, critical sections), the operation in the second thread can interrupt the operation in the first thread. If the interruption happens after `var` is incremented in the first thread but before the incremented value is written back, you can see unexpected results.

What Polyspace Considers as Nonatomic

Code Prover considers all operations as nonatomic unless you protect them, for instance, by using critical sections. See “Define Specific Operations as Atomic” on page 7-25.

Bug Finder considers an operation as nonatomic if it can translate into more than one machine instruction. For instance:

- The operation can involve both a read and write operation. For example, `var++` involves reading the value of `var`, increasing the value by one and writing the increased value back to `var`.
- The operation can involve a 64-bit variable on a 32-bit target. For example, the operation

```
long long var1, var2;  
var1=var2;
```

involves two steps in copying the content of `var2` to `var1` on certain targets.

Polyspace uses the **Pointer** size for your **Target processor type** as the threshold to compute atomicity. For instance, if you use `i386` as your **Target processor type**, the **Pointer** size is 32 bits and **Long long** and **Double** sizes are both 64 bits. Therefore, Polyspace considers copying one `long long` or `double` variable to another as nonatomic.

See also `Target processor type (-target)`.

- The operation can involve writing the return value of a function call to a shared variable. For example, the operation `x=func()` involves calling `func` and writing the return value of `func` to `x`.

To detect data races where at least one of the two interrupting operations is nonatomic, enable the Bug Finder checker `Data race`. To remove this constraint on the checker, enable `Data race including atomic operations`.

Define Specific Operations as Atomic

You might want to define a group of operations as atomic. This group of operations cannot be interrupted by operations in another thread or task.

Use one of these techniques:

- **Critical sections**

Protect a group of operations with critical sections.

A critical section begins and ends with calls to specific functions. You can use a predefined set of primitives to begin or end critical sections, or use your own functions.

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same beginning and ending function).

Specify critical sections using the option `Critical section details (-critical-section-begin -critical-section-end)`.

- **Temporally exclusive tasks**

Protect a group of operations by specifying certain tasks as temporally exclusive.

If a group of tasks are temporally exclusive, all operations in one task are atomic with respect to operations in the other tasks.

Specify temporal exclusion using the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

- **Task priorities** (Bug Finder only)

Protect a group of operations by specifying that certain tasks have higher priorities. For instance, interrupts have higher priorities over cyclic tasks.

You can specify up to four different priorities with these options (with highest priority listed first):

- `Interrupts (-interrupts)`
- `-preemptable-interrupts`
- `-non-preemptable-tasks`
- `Cyclic tasks (-cyclic-tasks)`

All operations in a task with higher priority are atomic with respect to operations in tasks with lower priorities. See also “Define Preemptable Interrupts and Nonpreemptable Tasks”.

- **Routine disabling interrupts** (Bug Finder only)

Protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

For a tutorial, see “Protections for Shared Variables in Multitasking Code” on page 7-20.

See Also

Critical section details (`-critical-section-begin -critical-section-end`) |
Cyclic tasks (`-cyclic-tasks`) | Interrupts (`-interrupts`) | Temporally exclusive
tasks (`-temporal-exclusions-file`)

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Protections for Shared Variables in Multitasking Code” on page 7-20

Define Preemptable Interrupts and Nonpreemptable Tasks

Bug Finder detects data races between concurrent tasks. Using Bug Finder analysis options, you can fix data race detection by specifying that certain tasks have higher priorities over others. A task with higher priority is atomic with respect to tasks with lower priority and cannot be interrupted by those tasks.

Emulating Task Priorities

You can specify up to four different priorities with these options (with highest priority listed first):

- Interrupts (nonpreemptable): Use option `Interrupts (-interrupts)`.
- Interrupts (preemptable): Use options `Interrupts (-interrupts)` and `-preemptable-interrupts`.
- Cyclic tasks (nonpreemptable): Use options `Cyclic tasks (-cyclic-tasks)` and `-non-preemptable-tasks`.

You can also define preemptable noncyclic tasks with the option `Entry points (-entry-points)` and `-non-preemptable-tasks`.

- Cyclic tasks (preemptable): Use option `Cyclic tasks (-cyclic-tasks)`.

You can also define noncyclic tasks with the option `Entry points (-entry-points)`.

For instance, interrupts have the highest priority and cannot be preempted by other tasks. To define a class of interrupts that can be preempted, lower their priority by making them preemptable.

Examples of Task Priorities

Consider this example with three tasks. A variable `var` is shared between the two tasks `task1` and `task2` without any protection such as a critical section. Depending on the priorities of `task1` and `task2`, Bug Finder shows a data race. The third task is not relevant for the example (and is added only to include a critical section, otherwise data race detection is disabled).

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void task1(void) {
    var++;
}

void task2(void) {
    var=0;
}

void task3(void){
    begin_critical_section();
    /* Some atomic operation */
}
```

```
    end_critical_section();  
}
```

Adjust the priorities of `task1` and `task2` and see whether a data race is detected. For instance:

1 Configure these multitasking options:

- `Interrupts (-interrupts)`: Specify `task1` and `task2` as interrupts.
- `Cyclic tasks (-cyclic-tasks)`: Specify `task3` as a cyclic task.
- `Critical section details (-critical-section-begin -critical-section-end)`: Specify `begin_critical_section` as a function beginning a critical section and `end_critical_section` as a function ending a critical section.

2 Run Bug Finder.

You do not see a data race. Since `task1` and `task2` are nonpreemptable interrupts, the shared variable cannot be accessed concurrently.

3 Change `task1` to a preemptable interrupt by using the option `-preemptable-interrupts`.

4 Run Bug Finder again. You now see a data race on the shared variable `var`.

Further Explorations

Modify this example in the following ways and see the effect of the modification:

- Change the priorities of `task1` and `task2`.

For instance, you can leave `task1` as a nonpreemptable interrupt but change `task2` to a preemptable interrupt by using the option `-preemptable-interrupts`.

The data race disappears. The reason is:

- `task1` has higher priority and cannot be interrupted by `task2`.
- The operation in `task2` is atomic and cannot be interrupted by `task1`.
- Enable the checker `Data race including atomic operations` (not enabled by default). Use the option `Find defects (-checkers)`.

You see the data race again. The checker considers all operations as potentially nonatomic and the operation in `task2` can now be interrupted by the higher priority operation in `task1`.

Try other modifications to the analysis options and see the result of the checkers.

See Also

Polyspace Analysis Options

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)`

Polyspace Results

`Data race` | `Data race including atomic operations`

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Protections for Shared Variables in Multitasking Code” on page 7-20
- “Define Atomic Operations in Multitasking Code” on page 7-24

Define Critical Sections with Functions That Take Arguments

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock and unlock function.

```
lock();  
/* Critical section code */  
unlock();
```

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same lock and unlock function). See also “Define Atomic Operations in Multitasking Code” on page 7-24.

Polyspace Assumption on Functions Defining Critical Sections

Polyspace ignores arguments to functions that begin and end critical sections.

For instance, Polyspace treats the two code sections below as the same critical section if you specify `my_task_1` and `my_task_2` as entry points, `my_lock` as the lock function and `my_unlock` as the unlock function.

```
int shared_var;  
  
void my_lock(int);  
void my_unlock(int);  
  
void my_task_1() {  
    my_lock(1);  
    /* Critical section code */  
    shared_var=0;  
    my_unlock(1);  
}  
  
void my_task_2() {  
    my_lock(2);  
    /* Critical section code */  
    shared_var++;  
    my_unlock(2);  
}
```

As a result, the analysis considers that these two sections are protected from interrupting each other even though they might not be protected. For instance, Bug Finder does not detect the data race on `shared_var`.

Often, the function arguments can be determined only at run time. Since Polyspace models the critical sections prior to the static analysis and run-time error checking phase, the analysis cannot determine if the function arguments are different and ignores the arguments.

Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments

When the arguments to the functions defining critical sections are compile-time constants, you can adapt the analysis to work around the Polyspace assumption.

For instance, you can use Polyspace analysis options so that the code in the preceding example appears to Polyspace as shown here.

```
int shared_var;

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);

void my_task_1() {
    my_lock_1();
    /* Critical section code */
    shared_var=0;
    my_unlock_1();
}

void my_task_2() {
    my_lock_2();
    /* Critical section code */
    shared_var++;
    my_unlock_2();
}
```

If you then specify `my_lock_1` and `my_lock_2` as the lock functions and `my_unlock_1` and `my_unlock_2` as the unlock functions, the analysis recognizes the two sections of code as part of different critical sections. For instance, Bug Finder detects a data race on `shared_var`.

To adapt the analysis for lock and unlock functions that take compile-time constants as arguments:

- 1 In a header file `common_polyspace_include.h`, convert the function arguments into extensions of the function name with `#define`-s. In addition, provide a declaration for the new functions.

For instance, for the preceding example, use these `#define`-s and declarations:

```
#define my_lock(X) my_lock_##X()
#define my_unlock(X) my_unlock_##X()

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);
```

- 2 Specify the file name `common_polyspace_include.h` as argument for the option `Include (-include)`.

The analysis considers this header file as `#include`-d in all source files that are analyzed.

- 3 Specify the new function names as functions beginning and ending critical sections. Use the options `Critical section details (-critical-section-begin -critical-section-end)`.

See Also

`Critical section details (-critical-section-begin -critical-section-end)`

More About

- “Protections for Shared Variables in Multitasking Code” on page 7-20

Configure Coding Rules Checking and Code Metrics Computation

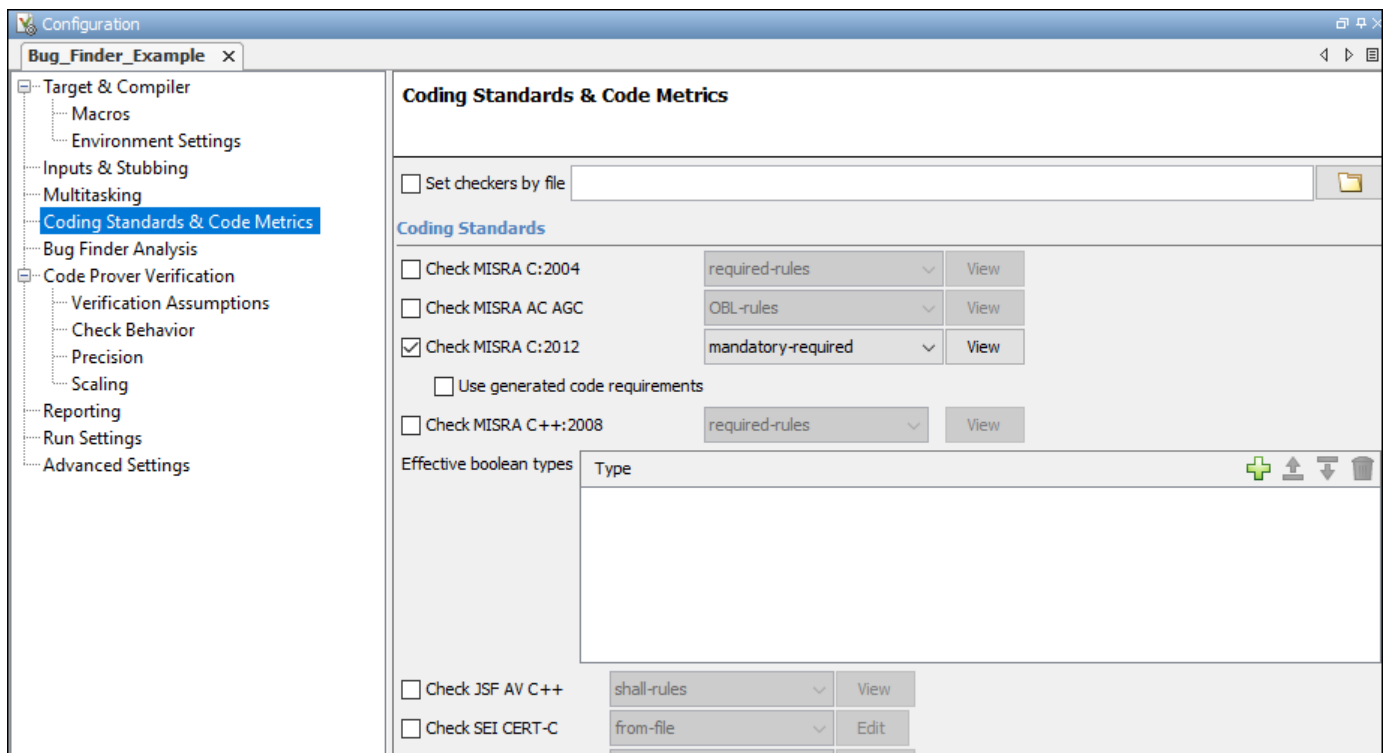
Check for Coding Standard Violations

With Polyspace, you can check your C/C++ code for violations of coding rules such as MISRA C:2012 rules. Adhering to coding rules can reduce the number of defects and improve the quality of your code.

Polyspace can detect the violations of these rules:

- MISRA C:2004
- MISRA C:2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 (*Bug Finder only*)
- CERT C (*Bug Finder only*)
- CERT C++ (*Bug Finder only*)
- ISO®/IEC TS 17961 (*Bug Finder only*)

Configure Coding Rules Checking



Specify Standard and Predefined Checker Subsets

Specify the coding rules through Polyspace analysis options. When you run Bug Finder or Code Prover, the analysis looks for coding rule violations in addition to other checks. You can disable the other checks and look for coding rule violations only.

In the Polyspace user interface (desktop products), the options are on the **Configuration** pane under the **Coding Standards & Code Metrics** node.

For C code, use one of these options:

- Check MISRA C:2004 (-misra2)

For generated code, enable the option specific to generated code.

- Check MISRA C:2012 (-misra3)

For generated code, enable the option specific to generated code.

- Check SEI CERT-C (-cert-c)
- Check ISO/IEC TS 17961 (-iso-17961)

For C++ code, use one of these options:

- Check MISRA C++ rules (-misra-cpp)
- Check JSF C++ rules (-jsf-coding-rules)
- Check AUTOSAR C++ 14 (-autosar-cpp14)
- Check SEI CERT-C++ (-cert-cpp)

You can specify a predefined subset of rules, for instance, mandatory for MISRA C:2012. These subsets are typically defined by the standard.

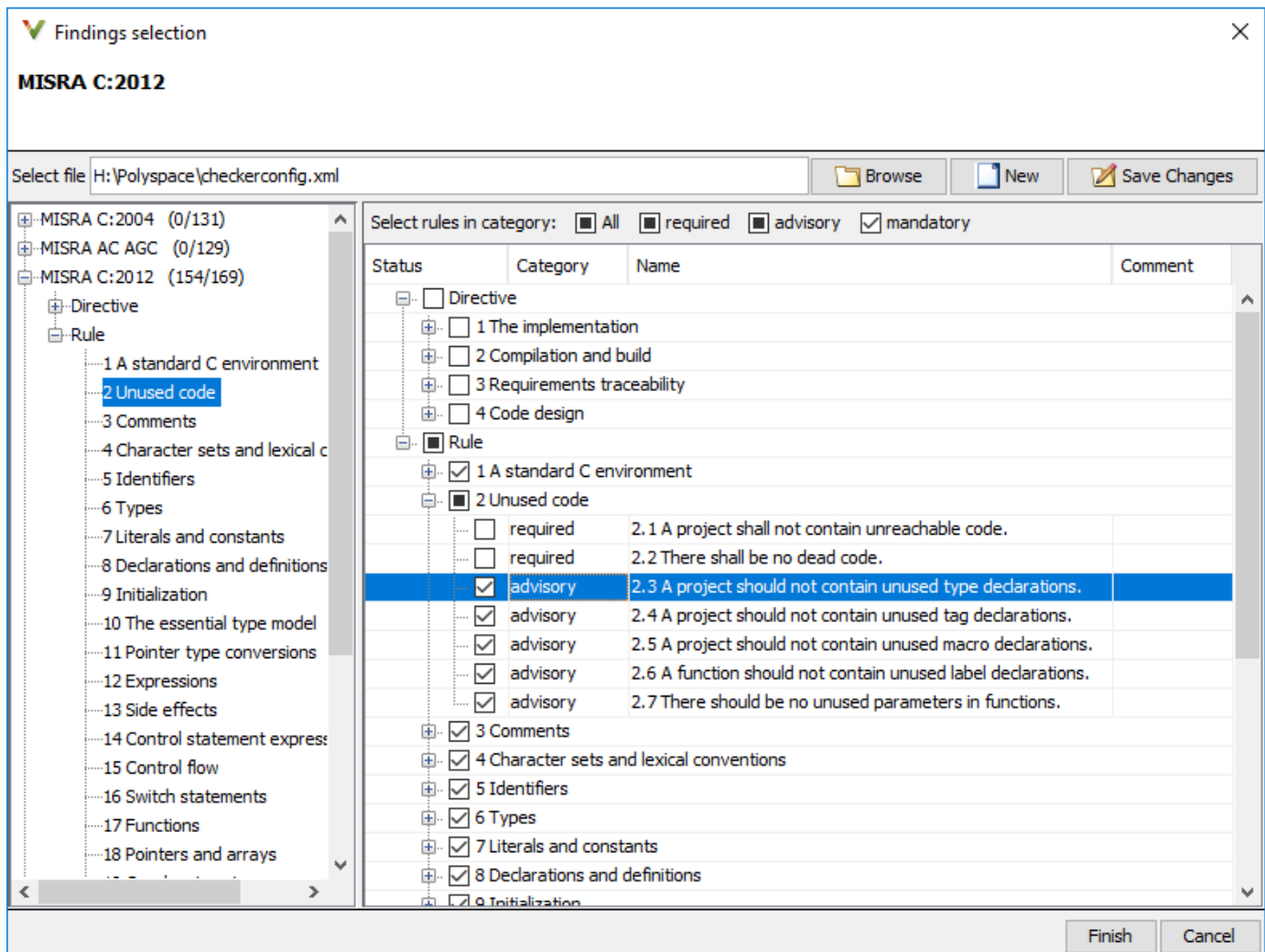
You can also define naming conventions for identifiers using regular expressions. See “Create Custom Coding Rules” on page 8-42.

Customize Checker Subsets

Instead of the predefined subsets, you can specify your own subset of rules from a coding standard.

User Interface (Desktop Products Only)

- 1 Select the coding standard. From the drop-down list for the subset of rules, select `from-file`. Click **Edit**.
- 2 In the **Findings selection** window, the coding standard is highlighted on the left pane. On the right pane, select the rules that you want to include in your analysis.



When you save the rule selections, the configuration is saved in an XML file that you can reuse for multiple analyses. The same file contains rules selected for all coding standards. You can reuse this file across multiple projects to enforce common coding standards in a team or organization. To reuse this file in another project in the Polyspace user interface:

- Choose a coding standard in the project configuration. From the drop-down list for the subset of rules, select from-file.
- Click **Edit** and browse to the file location. Alternatively, enter the file name as argument for the option Set checkers by file (-checkers-selection-file).

Command Line

With the Polyspace desktop products, you can create a coding standard XML file in the user interface and then use this file for command-line analysis. Provide this XML file with the option Set checkers by file (-checkers-selection-file).

With the Polyspace Server products, you have to create a coding standard XML from scratch. Depending on the standard that you want to enable, make a writeable copy of one of the files in *polyspaceserverroot*\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML and turn off rules using entries in the XML file (all rules from a standard are enabled in the template). Here, *polyspaceserverroot* is the root installation folder for the Polyspace Server products, for instance, C:\Program Files\Polyspace Server\R2019a.

For instance, to turn off MISRA C:2012 rule 8.1, use this entry in a copy of the file *misra_c_2012_rules.xml*:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="off">
      </check>
    ...
  </section>
  ...
</standard>
```

To use the XML file for a MISRA C:2012 analysis in Bug Finder, enter:

```
polyspace-bug-finder -sources filename -misra3 from-file
                    -checkers-selection-file misra_c_2012_rules.xml
```

For full list of rule id-s and section names, see:

-
-
-
-
- “Custom Coding Rules” (Polyspace Code Prover Access)
- “JSF C++ Rules” (Polyspace Code Prover Access)
- “MISRA C:2004 Rules” (Polyspace Code Prover Access)
- “MISRA C:2012 Directives and Rules” (Polyspace Code Prover Access)
- “MISRA C++:2008 Rules” (Polyspace Code Prover Access)

Note The XML format of the checker configuration file can change in future releases.

Check for Coding Standards Only

To check for coding standards only:

- In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`.
- In Code Prover, check for source compliance only. Use the option `Verification level (-to)`.

These rules are checked in the later stages of a Code Prover analysis: MISRA C:2004 rules 9.1, 13.7, and 21.1, and MISRA C:2012 rules 2.2, 9.1, 14.3, and 18.1. If you stop Code Prover at source

compliance checking, the analysis might not find all violations of these rules. You can also see a difference in results based on your choice for the option `Verification level (-to)`. For example, it is possible that Code Prover suspects in the first pass that a variable may be uninitialized but proves in the second pass that the variable is initialized. In that case, you see a violation of MISRA C:2012 Rule 9.1 in the first pass but not in the second pass.

Review Coding Rule Violations

Result Details

Variable trace

Result Review

Status: To fix

Severity: Medium

MISRA C:2012 5.1 (Required) ?
 External identifiers shall be distinct.
 External function `demo_corrected_sighandlerasynccunsafestrict` conflicts with the external identifier `demo_corrected_sighandlerasynccunsafer` (`programming.c` line 1171).


	Event	File	Scope	Line
1	Violation site	<code>programming.c</code>	<code>programming.c</code>	1171
2	MISRA C:2012 5.1	<code>programming.c</code>	File Scope	1230

Source

```

programming.c x
1226 void Corrected_sighandlerasynccunsafestrict(int signum) {
1227     int s0 = signum; /* Fix: avoid raise() */
1228 }
1229
1230 int demo_corrected_sighandlerasynccunsafestrict(void) {
1231     if (signal(SIGTERM, demo_term_handler) == SIG_ERR) {
1232         /* Handle error */
1233     }
1234     if (signal(SIGINT, corrected_sighandlerasynccunsafestrict) == SIG_ERR) {
1235         /* Handle error */
1236     }
1237     /* Program code */
1238     if (raise(SIGINT) != 0) {
1239         /* Handle error */
1240     }
1241     /* More code */
1242     return 0;
1243 }
  
```

After analysis, you see the coding standard violations on the **Results List** pane. Select a violation to see further details on the **Result Details** pane and the source code on the **Source** pane.

Violations of coding standards are indicated in the source code with the  icon.

For further steps, see “Review Results in Polyspace Code Prover Access” (Polyspace Code Prover Access).

Generate Reports

You can generate reports using templates that are explicitly defined for coding standards. Use the `CodingStandards` template. This template:

- Reports only coding standard violations in your analysis results, and omits other types of results such as defects, run-time errors or code metrics.
- Creates a separate chapter in the report for each coding standard. the chapter provides an overview of all violations of the standard and then lists each violation.

To specify a report template, use the option `Bug Finder and Code Prover report (-report-template)`.

See Also

More About

- “Interpret Polyspace Code Prover Access Results” (Polyspace Code Prover Access)

Avoid Violations of MISRA C:2012 Rules 8.x

MISRA C:2012 rules 8.1-8.14 enforce good coding practices surrounding declarations and definitions. If you follow these practices, you are less likely to have conflicting declarations or to unintentionally modify variables.

If you do not follow these practices *during coding*, your code might require major changes later to be MISRA C-compliant. You might have too many MISRA C violations. Sometimes, in fixing a violation, you might violate another rule. Instead, keep these rules in mind when coding. Use the MISRA C:2012 checker to spot any issues that you might have missed.

- **Explicitly specify all data types in declarations.**

Avoid implicit data types like this declaration of `k`:

```
extern void foo (char c, const k);
```

Instead use:

```
extern void foo (char c, const int k);
```

That way, you do not violate MISRA C:2012 Rule 8.1 (Polyspace Code Prover Access).

- **When declaring functions, provide names and data types for all parameters.**

Avoid declarations without parameter names like these declarations:

```
extern int func(int);  
extern int func2();
```

Instead use:

```
extern int func(int arg);  
extern int func2(void);
```

That way, you do not violate MISRA C:2012 Rule 8.2 (Polyspace Code Prover Access).

- **If you want to use an object or function in multiple files, declare the object or function once in only one header file.**

To use an object in multiple source files, declare it as `extern` in a header file. Include the header file in all the source files where you need the object. In one of those source files, define the object. For instance:

```
/* header.h */  
extern int var;  
  
/* file1.c */  
#include "header.h"  
/* Some usage of var */  
  
/* file2.c */  
#include "header.h"  
int var=1;
```

To use a function in multiple source files, declare it in a header file. Include the header file in all the source files where you need the function. In one of those source files, define the function.

That way, you do not violate MISRA C:2012 Rule 8.3 (Polyspace Code Prover Access), MISRA C:2012 Rule 8.4 (Polyspace Code Prover Access), MISRA C:2012 Rule 8.5 (Polyspace Code Prover Access), or MISRA C:2012 Rule 8.6 (Polyspace Code Prover Access).

- **If you want to use an object or function in one file only, declare and define the object or function with the static specifier.**

Make sure that you use the `static` specifier in all declarations and the definition. For instance, this function `func` is meant to be used only in the current file:

```
static int func(void);
static int func(void){
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 (Polyspace Code Prover Access) and MISRA C:2012 Rule 8.8 (Polyspace Code Prover Access).

- **If you want to use an object in one function only, declare the object in the function body.**

Avoid declaring the object outside the function.

For instance, if you use `var` in `func` only, do declare it outside the body of `func`:

```
int var;
void func(void) {
    var=1;
}
```

Instead use:

```
void func(void) {
    int var;
    var=1;
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 (Polyspace Code Prover Access) and MISRA C:2012 Rule 8.9 (Polyspace Code Prover Access).

- **If you want to inline a function, declare and define the function with the static specifier.**

Every time you add `inline` to a function definition, add `static` too:

```
static inline double func(int val);
static inline double func(int val) {
}
```

That way, you do not violate MISRA C:2012 Rule 8.10 (Polyspace Code Prover Access).

- **When declaring arrays, explicitly specify their size.**

Avoid implicit size specifications like this:

```
extern int32_t array[];
```

Instead use:

```
#define MAXSIZE 10
extern int32_t array[MAXSIZE];
```

That way, you do not violate MISRA C:2012 Rule 8.11 (Polyspace Code Prover Access).

- **When declaring enumerations, try to avoid mixing implicit and explicit specifications.**

Avoid mixing implicit and explicit specifications. You can specify the first enumeration constant explicitly, but after that, use either implicit or explicit specifications. For instance, avoid this type of mix:

```
enum color {red = 2, blue, green = 3, yellow};
```

Instead use:

```
enum color {red = 2, blue, green, yellow};
```

That way, you do not violate MISRA C:2012 Rule 8.12 (Polyspace Code Prover Access).

- **When declaring pointers, point to a const-qualified type unless you want to use the pointer to modify an object.**

Point to a const-qualified type by default unless you intend to use the pointer for modifying the pointed object. For instance, in this example, `ptr` is not used to modify the pointed object:

```
char last_char(const char * const ptr){  
}
```

That way, you do not violate MISRA C:2012 Rule 8.13 (Polyspace Code Prover Access).

Software Quality Objective Subsets (C:2004)

In this section...
“Rules in SQO-Subset1” on page 8-11
“Rules in SQO-Subset2” on page 8-12

Rules in SQO-Subset1

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

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
14.4	The <i>goto</i> statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
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.

Rule number	Description
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	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.2	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

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

Rule number	Description
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

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

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

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

See Also

Check MISRA C:2004 (-misra2)

More About

- “Check for Coding Standard Violations” on page 8-2

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQO-Subset1” on page 8-15
“Rules in SQO-Subset2” on page 8-15

Rules in SQO-Subset1

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

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
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.

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

Rule number	Description
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.

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

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

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

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

See Also

Check MISRA AC AGC (-misra-ac-agc)

More About

- “Check for Coding Standard Violations” on page 8-2

Software Quality Objective Subsets (C:2012)

In this section...
“Guidelines in SQO-Subset1” on page 8-18
“Guidelines in SQO-Subset2” on page 8-19

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

The following set of MISRA C:2012 coding guidelines typically reduces the number of unproven results in Polyspace Code Prover.

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
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

Rule	Description
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
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

Rule	Description
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
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)

More About

- “Check for Coding Standard Violations” on page 8-2

Software Quality Objective Subsets (C++)

In this section...

“SQO Subset 1 - Direct Impact on Selectivity” on page 8-21

“SQO Subset 2 - Indirect Impact on Selectivity” on page 8-22

SQO Subset 1 - Direct Impact on Selectivity

The following set of MISRA C++ coding rules will typically improve the number of unproven results in Polyspace Code Prover.

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.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

MISRA C++ Rule	Description
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
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.
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 number of unproven results in Polyspace Code Prover. The following set of coding rules may help to address design issues in your code. 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.

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

MISRA C++ Rule	Description
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.
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.

See Also

Check MISRA C++:2008 (-misra-cpp)

More About

- “Check for Coding Standard Violations” on page 8-2

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. The subsets are available with the options Check MISRA C:2004 (-misra2), Check MISRA AC AGC (-misra-ac-agc), and Check MISRA C:2012 (-misra3).

Argument	Purpose
single-unit-rules	<p>Check rules that apply only to single translation units.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the compilation phase.</p>
system-decidable-rules	<p>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.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the linking phase.</p>

See also “Check for Coding Standard Violations” on page 8-2.

MISRA C:2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

Environment

Rule	Description
1.1*	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use /* */ style comments.
2.3	The character sequence /* shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the #pragma directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.1*	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
5.3*	A typedef name shall be a unique identifier.
5.4*	A tag name shall be a unique identifier.
5.5*	No object or function identifier with a static storage duration should be reused.
5.6*	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
5.7*	No identifier name should be reused.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	typedefs 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 <code>unsigned int</code> or <code>signed int</code> .
6.5	Bit fields of type <code>signed int</code> shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4*	If objects or functions are declared more than once their types shall be compatible.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8*	An external object or function shall be declared in one file and only one file.
8.9*	An identifier with external linkage shall have exactly one external definition.
8.10*	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
8.11	The <code>static</code> 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 <code>=</code> 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: <ul style="list-style-type: none"> • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if <ul style="list-style-type: none"> • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type <code>unsigned char</code> or <code>unsigned short</code> , 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 <code>unsigned</code> 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 <code>void</code> .
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 <code>const</code> or <code>volatile</code> 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 <code>sizeof</code> operator should not be used on expressions that contain side effects.
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.
12.6	Operands of logical operators (<code>&&</code> , <code> </code> and <code>!</code>) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> or <code>!</code>).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	The comma operator shall not be used.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (<code>++</code>) and decrement (<code>--</code>) 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 <code>for</code> statement shall not contain any objects of floating type.
13.5	The three expressions of a <code>for</code> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a <code>for</code> 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 <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change.
14.4	The <code>goto</code> statement shall not be used.
14.5	The <code>continue</code> statement shall not be used.
14.6	For any iteration statement, there shall be at most one <code>break</code> 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 <code>switch</code> , <code>while</code> , <code>do while</code> or <code>for</code> statement shall be a compound statement.
14.9	An <code>if</code> (expression) construct shall be followed by a compound statement. The <code>else</code> keyword shall be followed by either a compound statement, or another <code>if</code> statement.
14.10	All <code>if else if</code> constructs should contain a final <code>else</code> clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between <code>switch</code> statement and first <code>case</code> .
15.1	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement
15.2	An unconditional <code>break</code> statement shall terminate every non-empty <code>switch</code> clause.
15.3	The final clause of a <code>switch</code> statement shall be the <code>default</code> clause.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.
15.5	Every <code>switch</code> statement shall have at least one <code>case</code> 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 <code>void</code> .
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 <code>&</code> , 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	<code>#include</code> 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 <code>#include</code> directives.
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> 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 <code>#defined</code> and <code>#undefd</code> within a block.
19.6	<code>#undef</code> 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 <code>#</code> or <code>##</code> .
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator.
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
19.13	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
19.14	The <code>defined</code> 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 <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.

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 <code>errno</code> shall not be used.
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.
20.7	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.
20.10	The library functions <code>atof</code> , <code>atoi</code> and <code>atoll</code> from library <code><stdlib.h></code> shall not be used.
20.11	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.
20.12	The time handling functions of library <code><time.h></code> 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 <code>/*</code> and <code>//</code> shall not be used within a comment.
3.2	Line-splicing shall not be used in <code>//</code> comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.1*	External identifiers shall be distinct.
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.
5.6*	A typedef name shall be a unique identifier.
5.7*	A tag name shall be a unique identifier.
5.8*	Identifiers that define objects or functions with external linkage shall be unique.
5.9*	Identifiers that define objects or functions with internal linkage should be unique.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.3*	All declarations of an object or function shall use the same names and type qualifiers.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5*	An external object or function shall be declared once in one and only one file.
8.6*	An identifier with external linkage shall have exactly one external definition.
8.7*	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	The <code>static</code> 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 <code>static</code> 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 <code>restrict</code> type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the <code>sizeof</code> operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an <code>if</code> statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The <code>goto</code> statement should not be used.
15.2	The <code>goto</code> statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a <code>goto</code> statement shall be declared in the same block, or in any block enclosing the <code>goto</code> statement.
15.4	There should be no more than one <code>break</code> or <code>goto</code> 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 <code>if ... else if</code> constructs shall be terminated with an <code>else</code> statement.

Switch Statements

Rule	Description
16.1	All <code>switch</code> statements shall be well-formed.
16.2	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement.
16.3	An unconditional <code>break</code> statement shall terminate every <code>switch</code> -clause.
16.4	Every <code>switch</code> statement shall have a <code>default</code> label.
16.5	A <code>default</code> label shall appear as either the first or the last <code>switch</code> label of a <code>switch</code> statement.
16.6	Every <code>switch</code> statement shall have at least two <code>switch</code> -clauses.
16.7	A <code>switch</code> -expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <stdarg.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 <code>static</code> 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 <code>union</code> keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	<code>#include</code> directives should only be preceded by preprocessor directives or comments.
20.2	The <code>'</code> , <code>"</code> , or <code>\</code> characters and the <code>/*</code> or <code>//</code> character sequences shall not occur in a header file name.
20.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	<code>#undef</code> 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 <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define</code> 'd before evaluation.
20.10	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
20.11	A macro parameter immediately following a <code>#</code> operator shall not immediately be followed by a <code>##</code> operator.
20.12	A macro parameter used as an operand to the <code>#</code> or <code>##</code> 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 <code>#</code> shall be a valid preprocessing directive.
20.14	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> , <code>#ifdef</code> or <code>#ifndef</code> directive to which they are related.

Standard Libraries

Rule	Description
21.1	<code>#define</code> and <code>#undef</code> 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 <code><stdlib.h></code> shall not be used.
21.4	The standard header file <code><setjmp.h></code> shall not be used.
21.5	The standard header file <code><signal.h></code> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <code><stdlib.h></code> shall not be used.
21.8	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> of <code><stdlib.h></code> shall not be used.
21.9	The library functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <code><tgmath.h></code> shall not be used.
21.12	The exception handling features of <code><fenv.h></code> 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.

See Also

Check MISRA AC AGC (`-misra-ac-agc`) | Check MISRA C:2004 (`-misra2`) | Check MISRA C:2012 (`-misra3`)

More About

- “Check for Coding Standard Violations” on page 8-2

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);
}
```

User Interface (Desktop Products Only)

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

The **Findings selection** window opens, displaying in the left pane all the coding standards Polyspace supports, and with the **Custom** node highlighted.

- 4 Specify the rules to check for in the right pane.

Expand the **4 Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
Status	Select <input checked="" type="checkbox"/> .
Convention	Enter All struct fields must begin with s_ and have capital letters or digits
Pattern	Enter 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 that you had entered, All struct fields must begin with `s_` and have capital letters
- 6 Right-click 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.

Command Line

With the Polyspace desktop products, you can create a coding standard XML file in the user interface and then use this file for command-line analysis. Provide this XML file with the option `Set checkers by file (-checkers-selection-file)`.

With the Polyspace Server products, you have to create a coding standard XML from scratch. Make a writable copy of the file `custom_rules.xml` in `polyspaceserverroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML` and turn off rules using entries in the XML file (all rules from a standard are enabled in the template). Here, `polyspaceserverroot` is the root installation folder for the Polyspace Server products, for instance, `C:\Program Files\Polyspace Server\R2019a`.

For instance, for custom rule 4.3 to be disabled, the configuration file must contain these lines:

```
<standard name="CUSTOM RULES">
  ...
  <section name="4 Structs">
    ...
    <check id="4.3" state="off">
    </check>
    ...
  </section>
  ...
</standard>
```

Provide this file as argument for the option `Set checkers by file (-checkers-selection-file)` during analysis, along with the option `Check custom rules (-custom-rules)`. For instance, for custom rules checking with Polyspace Code Prover Server, enter:

```
polyspace-code-prover-server -sources file -custom-rules from-file
                             -checkers-selection-file custom_rules.xml
```

See Also

Check custom rules (`-custom-rules`)

Compute Code Complexity Metrics

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

Polyspace does not compute code complexity metrics by default. To compute them during analysis, use the option `Calculate code metrics (-code-metrics)`.

After analysis, the software displays project, file and function metrics on the **Results List** pane. You can compare the computed metric values against predefined limits. If a metric value exceeds limits, you can redesign your code to lower the metric value. For instance, if the number of called functions is high and several of those functions are always called together, you can write one function that fuses the bodies of those functions. Call that one function instead of the group of functions that are called together.

Impose Limits on Metrics (Desktop Products Only)

In the user interface of the Polyspace desktop products, open some results with metrics computations. Then impose limits on the metric values and update results on the **Results List** pane to show only metric values that exceed the limits.

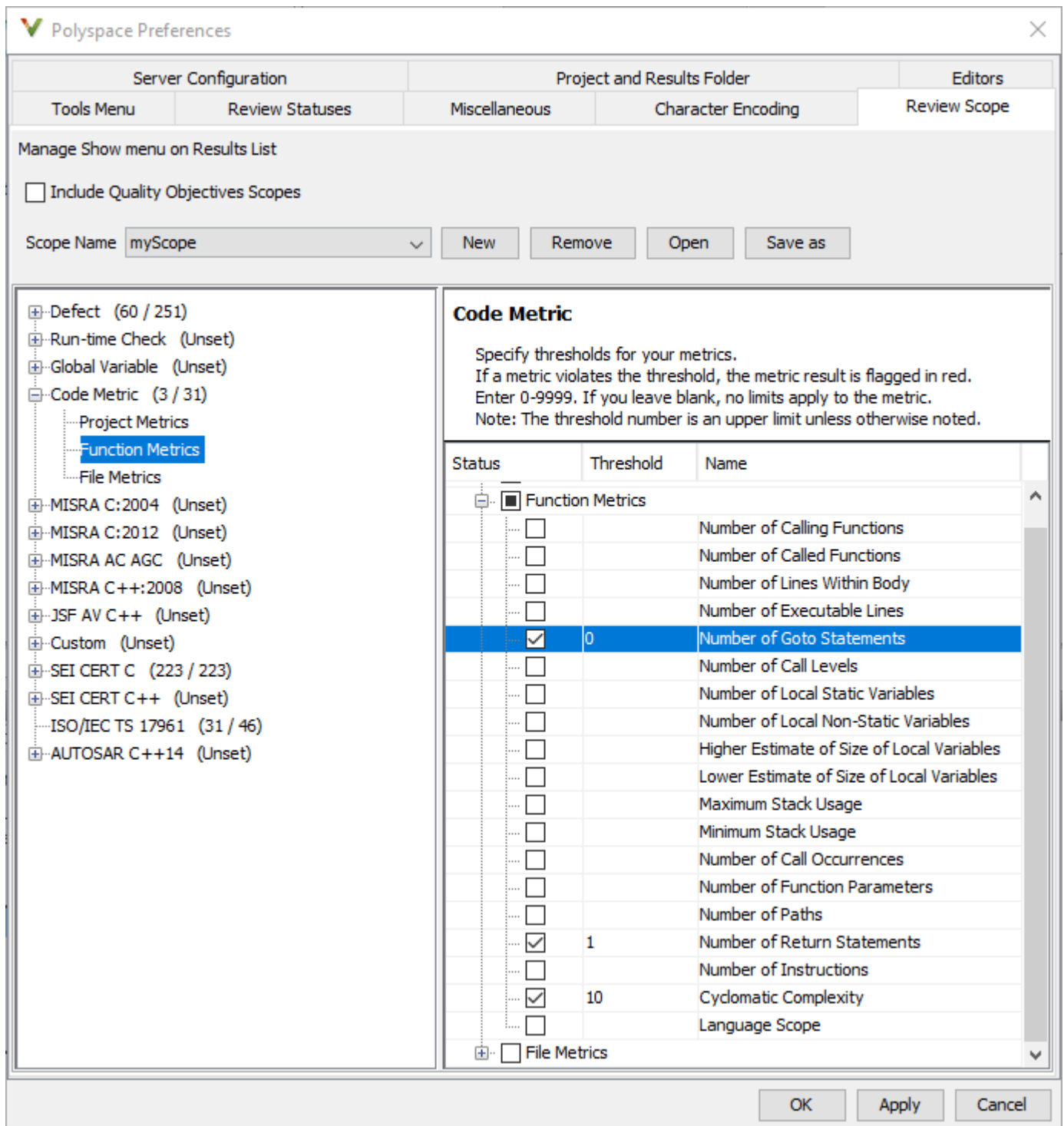
- 1 Select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To use a predefined limit, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows the additional option **HIS**. The option **HIS** displays the **HIS** code metrics on page 8-47 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” (Polyspace Code Prover Access). If only a some metrics in a category are selected, the check box next to the category name displays a symbol.



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

The drop-down list in the left of the **Results List** pane toolbar displays additional options.

- If you use predefined limits, the option HIS appears. This option displays code metrics only.

- If you define your own limits, the option corresponding to your limits file name appears.
- 4 Select the option corresponding to the limits that you want. Only metric values that violate your limits appear on the **Results List** pane.
 - 5 Review each violation and decide how to rework your code to avoid the violation.

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.

Impose Limits on Metrics (Server and Access products)

In the Polyspace Access web interface, limits on code complexity metrics are predefined. In the **Dashboard** perspective, if you select **Code Metric**, a **Code Metrics** window shows the metric values and limits.

To find the limits used, see “HIS Code Complexity Metrics” on page 8-47.

See Also

Calculate code metrics (-code-metrics)

More About

- “Code Metrics” (Polyspace Code Prover Access)
- “HIS Code Complexity Metrics” on page 8-47

HIS Code Complexity Metrics

The following list shows the Hersteller Initiative Software (HIS) standard metrics that Polyspace evaluates. These metrics and the recommended limits for their values are part of a standard defined by a major group of Original Equipment Manufacturers or OEMs. For more information on how to focus your review to this subset of code metrics, see “Compute Code Complexity Metrics” on page 8-44.

Project

Polyspace evaluates the following HIS metrics at the project level.

Metric	Recommended Upper Limit
Number of direct recursions	0
Number of recursions	0

File

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

Function

Polyspace evaluates the following HIS metrics at the function level.

Metric	Recommended Upper Limit
Cyclomatic complexity	10
Language scope	4
Number of call levels	4
Number of calling functions	5
Number of called functions	7
Number of function parameters	5
Number of goto statements	0
Number of instructions	50
Number of paths	80
Number of return statements	1

See Also

More About

- “Compute Code Complexity Metrics” on page 8-44
- “Code Metrics” (Polyspace Code Prover Access)

Configure Verification of Modules or Libraries

- “Provide Context for C Code Verification” on page 9-2
- “Provide Context for C++ Code Verification” on page 9-4
- “Verify C Application Without main Function” on page 9-6
- “Verify C++ Classes” on page 9-9

Provide Context for C Code Verification

This example shows how to provide context for your C code verification. If you use default options and do not provide a `main` function, Polyspace Code Prover checks your code for robustness against all verification conditions. For instance, the software:

- Considers that global variables and inputs of uncalled functions are full range.
- Generates a `main` that calls uncalled functions in arbitrary order.

In addition, if you do not define a function but declare and call it in your code, Polyspace stubs the function. For a detailed list of assumptions, see “Code Prover Analysis Assumptions” (Polyspace Code Prover).

You can use analysis options on the **Configuration** pane to change the default behavior and provide more context about your code. Performing contextual verification can result in more proven code and therefore fewer orange checks.

Control Variable Range

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
Variables to initialize (-main-generator-writes-variables)	Specify the global variables that Polyspace must consider as initialized despite no explicit initialization in the code.
Constraint setup (-data-range-specifications)	Specify range for global variables.

Control Function Call Sequence

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
Initialization functions (-functions-called-before-main)	Specify the functions that the generated <code>main</code> must call first.
Functions to call (-main-generator-calls)	Specify the functions that the generated <code>main</code> must call later.

Control Stubbing Behavior

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Inputs & Stubbing** node.

Option	Purpose
Functions to stub (-functions-to-stub)	Specify the functions that Polyspace must stub.

See Also

More About

- “Verify C Application Without main Function” on page 9-6

Provide Context for C++ Code Verification

This example shows how to provide context to your C++ code verification. If you use default options and do not provide a `main` function, Polyspace Code Prover checks your code for robustness against all verification conditions. For instance, the software:

- Considers that global variables and inputs of uncalled functions and methods are full range.
- Generates a `main` that calls uncalled functions in arbitrary order.

In addition, if you do not define a function but declare and call it in your code, Polyspace stubs the function. For a detailed list of assumptions, see “Code Prover Analysis Assumptions” (Polyspace Code Prover).

You can use analysis options on the **Configuration** pane to change the default behavior and provide more context about your code. Performing contextual verification can result in more proven code and therefore fewer orange checks.

Control Variable Range

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
Variables to initialize (-main-generator-writes-variables)	Specify the global variables that Polyspace must consider as initialized despite no explicit initialization in the code.
Constraint setup (-data-range-specifications)	Specify range for global variables.

Control Function Call Sequence

- 1 Use the following options to call class methods. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
Class (-class-analyzer)	Specify classes whose methods the generated <code>main</code> must call.
Functions to call within the specified classes (-class-analyzer-calls)	Specify methods that the generated <code>main</code> must call.
Analyze class contents only (-class-only)	Specify that the generated <code>main</code> must call class methods only.
Skip member initialization check (-no-constructors-init-check)	Specify that the generated <code>main</code> must not check whether each class constructor initializes all class members.

- 2 Use the following options to call functions that are not class methods. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
Initialization functions (-functions-called-before-main)	Specify the functions that the generated main must call first.
Functions to call (-main-generator-calls)	Specify the functions that the generated main must call later.

See Also

More About

- “Verify C++ Classes” on page 9-9

Verify C Application Without main Function

Polyspace verification requires that your code must have a `main` function. You can do one of the following:

- Provide a `main` function in your code.
- Specify that Polyspace must generate a `main`.

Generate main Function

Before verification, specify one of the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Description
Verify whole application	The verification stops if the software does not detect a <code>main</code> .
Verify module or library (-main-generator)	<p>Before verification, Polyspace checks if your code contains a <code>main</code> function.</p> <p>If a <code>main</code> function exists, the software uses that <code>main</code>. Otherwise, the software generates a <code>main</code> using the options that you specify:</p> <ul style="list-style-type: none"> • Variables to initialize (-main-generator-writes-variables) • Initialization functions (-functions-called-before-main) • Functions to call (-main-generator-calls)

Manually Write main Function

During automatic `main` generation, the software makes certain assumptions about the function call sequence or behavior of global variables. For instance, the default automatically generated `main` models the following behavior:

- The functions that you specify using the option `Functions to call` (-main-generator-calls) can be called in arbitrary order.
- In the beginning of each function body, global variables can have the full range of values allowed by their type.

To provide a more accurate model of the call sequence, you can manually write a `main` function for the purposes of verification. You can add this `main` function in a separate file to your project. In some cases, providing an accurate call sequence can reduce the number of orange checks. For example, in the following code, Polyspace assumes that `f` and `g` can be called in any order. Therefore, it produces an orange overflow for the case when `f` is called before `g`. If you know that `f` is called after `g`, you can write a `main` function to model this sequence.

```
static char x;
static int y;
```

```

void f(void)
{
    y = 300;
}

void g(void)
{
    x = y;
}

```

Example 1: main Calls One Function Before Another

Suppose you want to verify two functions `func1` and `func2` that have the following prototypes.

```

int func1(void *ptr, int x);
void func2(int x, int y);

```

You have the requirement that `func1` is always called before `func2`.

To manually define a `main` that models this behavior:

- 1 Write a `main` containing declarations of a `volatile` variable for each function parameter type.
- 2 Write a loop with a `volatile` termination condition.

The verification assumes that a `volatile` variable can have any value allowed by its type. Because the loop potentially terminates after any run, this condition models the fact that you call `func1` and `func2` an arbitrary number of times.

- 3 Inside this loop, call `func2` after `func1`.

You can write the following `main`:

```

void main()
{
    volatile int random=0;
    volatile void * volatile ptr;
    while(random)
    {
        random = func1(ptr, random);
        func2(random, random);
    }
}

```

Example 2: main Calls One Function 10 Times Before Another

Suppose you want to verify two functions `func1` and `func2` with the following prototypes:

```

void func1(int);
void func2(void);

```

You know that when both `func1` and `func2` are called, `func1` is always called 10 times before `func2`.

To manually define a `main` that models this behavior:

- 1 Write a `main` containing declarations of a `volatile` variable for each function parameter type.
- 2 In your `main` function, call `func1` in a loop 10 times before `func2`.

For instance, you can write the following `main`:

```
void main(void) {
    int i=0;
    volatile int random=0;

    while (++i <= 10)
        func1(random);

    func2();
}
```

See Also

More About

- “Provide Context for C Code Verification” on page 9-2

Verify C++ Classes

In this section...
“Verification of Classes” on page 9-9
“Methods and Class Specifics” on page 9-11

Verification of Classes

Object-oriented languages such as C++ are designed for reusability. When developing code in such a language, you do not necessarily know every contexts in which the class is deployed. A class or a class family is safe for reuse if it free of defects for all possible contexts.

To make your classes safe against all possible contexts, perform a robustness verification and remove as many run-time errors as possible.

Polyspace Code Prover performs a robustness verification by default. If you provide the software the class definition together with the definition of the class methods, the software simulates all uses of the class. If some of the method definitions are missing, the software automatically stubs them.

- 1 The software verifies each constructor by creating an object using the constructor. If a constructor does not exist, the software uses the default constructor.
- 2 The software verifies the public, static and protected class methods of those objects assuming that:
 - The methods can be called in arbitrary order.
 - The method parameters can have any value in the range allowed by their data type.

To perform this verification, by default, it generates a `main` function that calls the methods that are not called elsewhere in the code. If you want all your methods to be verified for all contexts, modify this behavior so that the generated `main` calls all public and protected methods instead of just the uncalled ones. For more information, see `Functions to call within the specified classes (-class-analyzer-calls)`.

- 3 The software calls the destructor of those objects (if they exist) and verifies them.

When verifying classes, Polyspace makes certain assumptions.

Code Construct	Assumption
Global variable	<p>Unless explicitly initialized, in each method, global variables can have any value allowed by their type.</p> <p>For instance, in the following code, Polyspace assumes that <code>globvar1</code> can have any value allowed by its type. Therefore, an orange Division by zero appears on the division by <code>globvar1</code>. However, because <code>globvar2</code> is explicitly initialized, the Division by zero check on division by <code>globvar2</code> is green.</p> <pre>extern int fround(float fx); // global variables int globvar1; int globvar2 = 100; class Location { private: int x; public: Location(int intx = 0) { x = intx; }; void setx(int intx) { x = intx; }; void fsetx(float fx) { int tx = fround(fx); if (tx / globvar1 != 0) { tx = tx / globvar2; setx(tx); } }; };</pre>

Code Construct	Assumption
Classes with undefined constructors	<p>The members of the classes can be non-initialized.</p> <p>In the following example, Polyspace assumes that <code>m_loc.x</code> can be non-initialized. Therefore, an orange Non-initialized variable error appears on <code>x</code> in the <code>getMember</code> method. Following the check, Polyspace assumes that the variable can have any value allowed by its type. Therefore, an orange Overflow appears on the addition operation in the <code>show</code> method.</p> <pre> class OtherClass { protected: int x; public: OtherClass (int intx); int getMember(void) { return x; }; }; class MyClass { OtherClass m_loc; public: MyClass(int intx) : m_loc(0) {}; void show(void) { int wx, wl; wx = m_loc.getMember(); wl = wx + 2; }; }; </pre>

Methods and Class Specifics

- “Simple Class” on page 9-11
- “Template Classes” on page 9-13
- “Abstract Classes” on page 9-13
- “Static Classes” on page 9-14
- “Inherited Classes” on page 9-14
- “Simple Inheritance” on page 9-15
- “Multiple Inheritance” on page 9-16
- “Virtual Inheritance” on page 9-17
- “Class Integration” on page 9-17

Simple Class

Consider the following class:

Stack.h

```
#define MAXARRAY 100
```

```
class stack
{
    int array[MAXARRAY];
    long toparray;

public:
    int top (void);
    bool isempty (void);
    bool push (int newval);
    void pop (void);
    stack ();
};
```

stack.cpp

```
1 #include "stack.h"
2
3 stack::stack ()
4 {
5     toparray = -1;
6     for (int i = 0 ; i < MAXARRAY; i++)
7         array[i] = 0;
8 }
9
10 int stack::top (void)
11 {
12     int i = toparray;
13     return (array[i]);
14 }
15
16 bool stack::isempty (void)
17 {
18     if (toparray >= 0)
19         return false;
20     else
21         return true;
22 }
23
24 bool stack::push (int newvalue)
25 {
26     if (toparray < MAXARRAY)
27     {
28         array[++toparray] = newvalue;
29         return true;
30     }
31
32     return false;
33 }
34
35 void stack::pop (void)
36 {
37     if (toparray >= 0)
38         toparray--;
39 }
```

The class analyzer calls the constructor and then the methods in any order many times.

The verification of this class highlights two problems:

- The `stack::push` method may write after the last element of the array, resulting in the OBAI orange check at line 28.
- If called before `push`, the `stack::top` method will access element -1, resulting in the OBAI and NIV checks at line 13.

Fixing these problems will eliminate run-time errors in this class.

Template Classes

A template class allows you to create a class without explicit knowledge of the data type that the class operations handle. Polyspace cannot verify a template class directly. The software can only verify a specific instance of the template class. To verify a template class:

- 1 Create an explicit instance of the class.
- 2 Define a `typedef` of the instance and provide that `typedef` for verification.

In the following example, `calc` is a template class that can handle any data type through the identifier `myType`.

```
template <class myType> class calc
{
public:
    myType multiply(myType x, myType y);
    myType add(myType x, myType y);
};
template <class myType> myType calc<myType>::multiply(myType x,myType y)
{
    return x*y;
}
template <class myType> myType calc<myType>::add(myType x, myType y)
{
    return x+y;
}
```

To verify this class:

- 1 Add the following code to your Polyspace project.

```
template class calc<int>;
typedef calc<int> my_template;
```
- 2 Provide `my_template` as argument of the option **Class**. See `Class (-class-analyzer)` (Polyspace Code Prover).

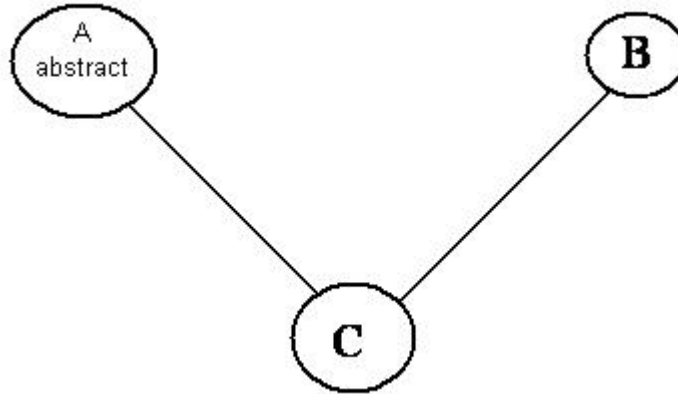
Abstract Classes

In the real world, an instance of an abstract class cannot be created, so it cannot be analyzed. However, it is easy to establish a verification by removing the pure declarations. For example, this can be accomplished via an abstract class definition change:

```
void abstract_func () = 0; by void abstract_func ();
```

If an abstract class is provided for verification, the software will make the change automatically and the virtual pure function (`abstract_func` in the example above) will then be ignored during the verification of the abstract class.

This means that no call will be made from the generated main, so the function is completely ignored. Moreover, if the function is called by another one, the pure virtual function will be stubbed and an orange check will be placed on the call with the message “call of virtual function [f] may be pure.”



Consider the following classes:

A is an abstract class

B is a simple class.

A and B are base classes of C.

C is not an abstract class.

As it is not possible to create an object of class A, this class cannot be analyzed separately from other classes. Therefore, you are not allowed to specify class A to the Polyspace class analyzer. Of course, class C can be analyzed in the same way as in the previous section “Multiple Inheritance.”

Static Classes

If a class defines a static methods, it is called in the generated main as a classical one.

Inherited Classes

When a function is not defined in a derived class, even if it is visible because it is inherited from a father's class, it is not called in the generated main. In the example below, the class Point is derived from the class Location:

```

class Location
{
protected:
    int x;
    int y;
    Location (int intx, int inty);
public:
    int getx(void) {return x;};
    int gety(void) {return y;};
};
class Point : public Location
{
protected:
    bool visible;
  
```

```

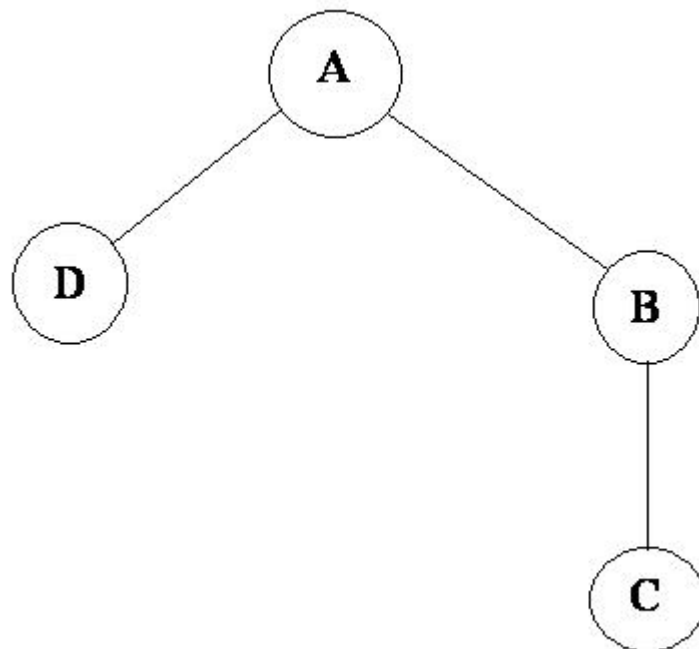
public :
    Point(int intx, int inty) : Location (intx, inty)
    {
        visible = false;
    };
    void show(void) { visible = true;};
    void hide(void) { visible = false;};
    bool isvisible(void) {return visible;};
};

```

Although the two methods `Location::getx` and `Location::gety` are visible for derived classes, the generated main does not include these methods when analyzing the class `Point`.

Inherited members are considered to be volatile if they are not explicitly initialized in the father's constructors. In the example above, the two members `Location::x` and `Location::y` will be considered volatile. If we analyze the above example in its current state, the method `Location::Location(constructor)` will be stubbed.

Simple Inheritance



Consider the following classes:

A is the base class of B and D.

B is the base class of C.

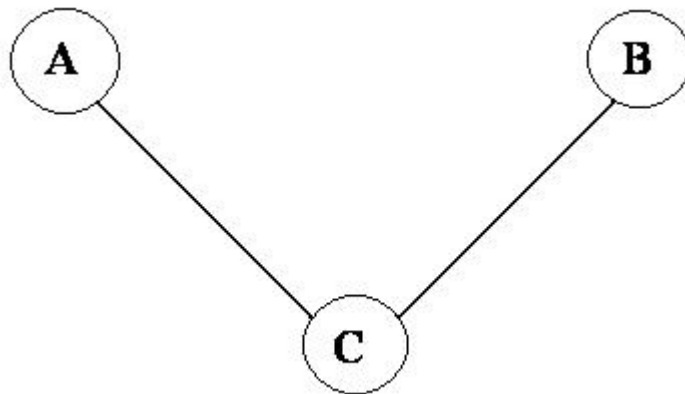
In a case such a this, Polyspace software allows you to run the following verifications:

- 1 You can analyze class A just by providing its code to the software. This corresponds to the previous “Simple Class” section in this chapter.
- 2 You can analyze class B class by providing its code and the class A declaration. In this case, A code will be stubbed automatically by the software.

- 3 You can analyze class B class by providing B and A codes (declaration and definition). This is a “first level of integration” verification. The class analyzer will not call A methods. In this case, the objective is to find bugs only in the class B code.
- 4 You can analyze class C by providing the C code, the B class declaration and the A class declaration. In this case, A and B codes will be stubbed automatically.
- 5 You can analyze class C by providing the A, B and C code for an integration verification. The class analyzer will call all the C methods but not inherited methods from B and A. The objective is to find only defects in class C.

In these cases, there is no need to provide D class code for analyzing A, B and C classes as long as they do not use the class (e.g., member type) or need it (e.g., inherit).

Multiple Inheritance



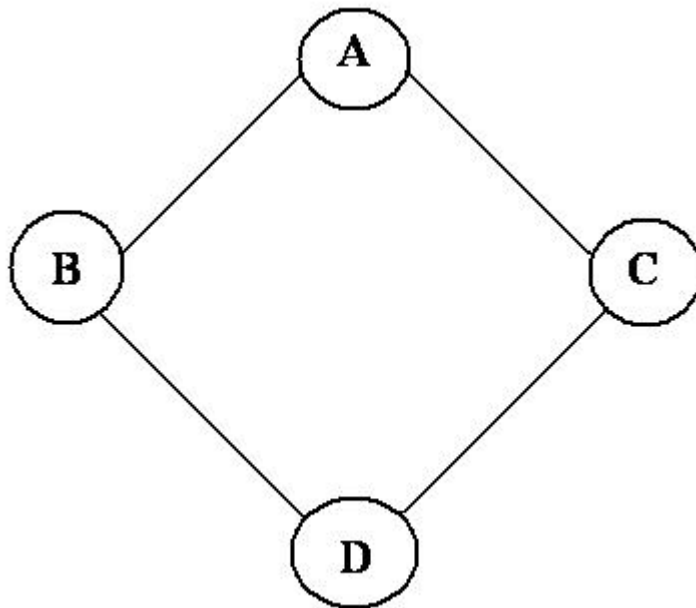
Consider the following classes:

A and B are base classes of C.

In this case, Polyspace software allows you to run the following verifications:

- 1 You can analyze classes A and B separately just by providing their codes to the software. This corresponds to the previous “Simple Class” section in this chapter.
- 2 You can analyze class C by providing its code with A and B declarations. A and B methods will be stubbed automatically.
- 3 You can analyze class C by providing A, B and C codes for an integration verification. The class analyzer will call all the C methods but not inherited methods from A and B. The objective is to find bugs only in class C.

Virtual Inheritance



Consider the following classes:

B and C classes virtually inherit the A class

B and C are base classes of D.

A, B, C and D can be analyzed in the same way as described in the previous section "Abstract Classes."

Virtual inheritance has no impact on the way of using the class analyzer.

Class Integration

Consider a C class that inherits from A and B classes and has object members of AA and BB classes.

A class integration verification consists of verifying class C and providing the codes for A, B, AA and BB. If some definitions are missing, the software will automatically stub them.

See Also

"Provide Context for C++ Code Verification" on page 9-4

Configure Code Prover Run-Time Checks

Modify or Disable Code Prover Run-Time Checks

A Code Prover analysis exhaustively checks source code for common run-time errors. The exhaustive nature of the static analysis is designed to prevent the errors from occurring at run time in safety critical software. Because of the safety critical intent of the analysis, Code Prover does not allow you to:

- Selectively disable specific run-time checks.
- Hide results of run-time checks from the results list through source code annotations. You can justify the results through annotations but the justified results continue to appear in the results list.

You can choose to suppress specific results by applying filters *after the analysis* or even creating filtered reports. If you create a filtered report from the Code Prover results, the report shows the filters and reflects your choices. For more information, see “Filter and Sort Results” (Polyspace Code Prover Access) and `polyspace-report-generator`.

However, you can modify the default behavior of certain checks and completely disable the checks for initialization. When you generate a report from the analysis results, the report shows the use of these options.

This topic lists the options that modify the default behavior of certain run-time checks. Note that though an option primarily addresses a specific type of check, checks of other types are also impacted. See “Code Prover Analysis Following Red and Orange Checks” (Polyspace Code Prover Access).

Integer Overflow

Check	Default Behavior	Option
Invalid shift operations	Left shifts are not allowed on signed operands.	Allow negative operand for left shifts (<code>-allow-negative-operand-in-shift</code>)
Overflow	Signed integer overflows are forbidden.	Overflow mode for signed integer (<code>-signed-integer-overflows</code>)
Overflow	Unsigned integer overflows are not detected.	Overflow mode for unsigned integer (<code>-unsigned-integer-overflows</code>)

Floating Point Overflow

Check	Default Behavior	Option
<ul style="list-style-type: none"> • Overflow • Invalid operation on floats 	Non-finite floats are not considered.	<ul style="list-style-type: none"> • Consider non finite floats (-allow-non-finite-floats) • Infinities (-check-infinite) • NaNs (-check-nan)
Subnormal float	Subnormal results are not detected.	Subnormal detection mode (-check-subnormal)

Initialization

Check	Default Behavior	Option
<ul style="list-style-type: none"> • Non-initialized local variable • Non-initialized variable • Non-initialized pointer • Return value not initialized 	Checks for initialization are performed only when a variable is read.	Disable checks for non-initialization (-disable-initialization-checks)
Global variable not assigned a value in initialization code	Checks for global variable initialization is performed only when the variable is read.	<ul style="list-style-type: none"> • Ignore default initialization of global variables (-no-def-init-glob) • Check that global variables are initialized after warm reboot (-check-globals-init)

Library Functions

Check	Default Behavior	Option
Invalid use of standard library routine	Only Standard Library routines are checked for validity of arguments. User-defined library functions are not checked.	-code-behavior-specifications

Pointers

Check	Default Behavior	Option
Illegally dereferenced pointer	Pointers to a structure field cannot point to another field.	Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)
Illegally dereferenced pointer	Pointers to a structure must have enough buffer for the entire structure.	Allow incomplete or partial allocation of structures (-size-in-bytes)
Illegally dereferenced pointer	Stack pointer dereference outside scope is not detected.	Detect stack pointer dereference outside scope (-detect-pointer-escape)
Correctness condition	Function pointer mismatches are not allowed.	Permissive function pointer calls (-permissive-function-pointer)

Unreachable Code or Dead Code

Checker	Default Behavior	Option
<ul style="list-style-type: none"> Function not called Function not reachable 	Uncalled functions and functions called from unreachable code are not reported.	Detect uncalled functions (-uncalled-function-checks)

See Also

More About


- “Prepare Scripts for Polyspace Analysis” on page 1-2

Configure Comment Import from Previous Results

- “Import Review Information from Previous Polyspace Analysis” on page 11-2
- “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 11-6

Import Review Information from Previous Polyspace Analysis

After you have reviewed analysis results, you can reuse information from the review for subsequent analyses. If you specify a result status or severity or add notes in your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations. For more information on commenting, see Polyspace Code Prover Access documentation.

This topic shows how to import review information from one result file to another. Importing the review information saves you from reviewing already justified results. For instance, after you import the information, on the **Results List** pane (user interface of desktop products), clicking the  icon skips justified results. Using this icon, you can browse through unreviewed results. You can also filter the justified checks from display.

Automatic Import from Last Analysis

By default, in the Polyspace user interface (desktop products only), review information is imported automatically from the most recent analysis on the project module. You can disable this default behavior.

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

If you upload results to the Polyspace Access web interface, review information from the last run of the same project are applied to the current run. You cannot disable the automatic import.

If you run analysis at the command line (and do not upload results to the Polyspace Access web interface), you have to explicitly import from another set of results. See “Command Line” on page 11-3.

Import from Another Analysis Result

You can import review information directly from another Polyspace result to the current result.

If a result is found in both a Bug Finder and Code Prover analysis, you can add review information to the Bug Finder result and import to the Code Prover result. For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add review information to coding rule violations in Bug Finder and import to the same violations in Code Prover.

User Interface (Desktop Products Only)

To import review information from another set of results:

- 1 Open the current analysis results.
- 2 Select **Tools > Import Comments**.
- 3 Navigate to the folder containing your previous results.
- 4 Select the other results file (with extension `.psbf` or `.pscp`) and then click **Open**.

The review information from the previous results are imported into the current results.

Command Line

Use the option `-import-comments` during analysis to import comments from a previous verification.

To import review information from multiple results, use the `polyspace-comments-import` command.

Import Algorithm

You can directly import review information from another set of results into the current results. However, it is possible that part of your review information is not imported to a subsequent analysis because:

- You have changed your source code so that the line with a previous result is not exactly identical to the line in the current run.

The comment import tool accounts for additional code that simply shifts an existing line. For instance, the tool recognizes that line 10 in Run 1 and line 12 in Run 2 have the same statement. If a division by zero occurs on line 10 in Run 1 and you have not fixed the issue in Run 2, the result along with associated review information are imported to line 12 in Run 2.

- Run 1:

```
10 baseLine = min/numRecipients;
11
12
```

- Run 2:

```
10 /* Calculate a baseline per recipient
11    based on minimum available resources */
12 baseLine = min/numRecipients;
```

However, if you change the line content itself, for instance, change `numRecipients` to `numReceiver`, the result and review information are not imported.

- You have changed your source code so that the Code Prover result color has changed.
- You entered new review information for the same result.

If the content of a line does not change and shows the same result as the previous analysis, the review information is imported. In unlikely scenarios, you might get the same result on the same line despite changing previous lines that lead to the result. Your review information from a previous analysis is then imported to the new result. If you justified the previous result with a status such as **Not a defect**, it is likely that you want to continue this justification with the new result. For

instance, if you accepted an overflow previously because you accounted for a wrap-around behavior after the overflow, you are likely to accept the overflow whatever the cause. In a few cases, you might want to review the result again and might not be aware that the result merits another review. To avoid this situation:

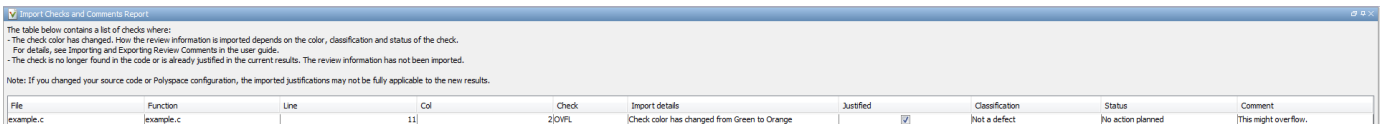
- When justifying nonlocal results that are related to previous events, use careful judgement.
- For critical components, conduct periodic assessments of justified results to see if the justifications still apply. Such assessments are useful specially for the Code Prover run-time checks.

View Imported Review Information That Does Not Apply

In the Polyspace user interface (desktop products only), the Import Checks and Comments Report highlights differences between two analysis results. When you import review information from a previous analysis, you can see this report. If you have closed the report after an import, to review the report again:

- 1 Select **Window > Show/Hide View > Import Comments Report**.

The Import Checks and Comments Report opens, highlighting differences in the two results.



- 2 Review the differences between the two results.

Your review information can differ between two results because of the following reasons:

- In Code Prover, if the check color changes, Polyspace imports the **Comment** field but not the **Status** field. In addition, Polyspace imports the **Severity** and **Justified** fields depending on the color change.

Color Change	Severity	Justified
Orange or red to green	Not imported	Imported
Gray to green	Not imported	Imported, if the Severity was set to High, Medium or Low.
Red to orange or vice versa	Imported	Imported
Green to red/orange/gray	Not imported	Not imported

- If a result no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review information from the previous result.
- If you have already entered different review information for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review information from the previous result.

See Also

-import-comments | polyspace-comments-import

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results

When you check your code for MISRA C: 2012 violations, Polyspace imports justifications of MISRA C: 2004 violations from previous analyses (if they exist). You can upgrade from checking of MISRA C: 2004 rules to MISRA C: 2012 rules while retaining your justifications. For general rules on comment import, see “Import Review Information from Previous Polyspace Analysis” on page 11-2.

The software maps MISRA C: 2004 **Status**, **Severity**, and **Comment** values that you added through the user interface or code annotations to the corresponding MISRA C: 2012 results, if the results exist. For more information about mapping, consult addendum one of the MISRA C: 2012 publication.

Type	Check: (9)	Status	Severity	Comment: (9)
MISRA C:2004	6.3 Typedefs that indicate size and sig...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2004	6.3 Typedefs that indicate size and sig...	To fix	Medium	MISRA2004-6.3
MISRA C:2004	8.1 Functions shall have prototype de...	To fix	Low	MISRA2004-8.1
MISRA C:2004	11.3 A cast should not be performed b...	Justified	Low	MISRA2004-11.3
MISRA C:2004	11.4 A cast should not be performed b...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2004	12.12 The underlying bit representatio...	Unreviewed	Unset	MISRA2004-12.12 comm...
MISRA C:2004	13.2 Tests of a value against zero sho...	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	14.4 The goto statement shall not be ...	Not a defect	Low	MISRA2004-14.4
MISRA C:2004	14.9 An if (expression) construct shall ...	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	19.5 Macros shall not be #define'd an...	Justified	Low	MISRA2004-19.5

If you are transitioning from MISRA C: 2004 to MISRA C: 2012, you do not have to review results that you have already justified.

Type	Check	Status	Severity	Comment: (7)
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	To fix	Medium	MISRA2004-6.3
MISRA C:2012	8.4 A compatible declaration shall be v...	To fix	Low	MISRA2004-8.1
MISRA C:2012	11.3 A cast shall not be performed bet...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2012	11.4 A conversion should not be perfo...	Justified	Low	MISRA2004-11.3
MISRA C:2012	14.4 The controlling expression of an i...	Not a defect	Low	MISRA2004-13.2
MISRA C:2012	15.1 The goto statement should not b...	Not a defect	Low	MISRA2004-14.4
MISRA C:2012	15.6 The body of an iteration-stateme...	Not a defect	Low	MISRA2004-13.2

Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result

When you justify MISRA C: 2004 violations by using code block syntax or multiple line annotation syntax, and multiple violations map to the same MISRA C: 2012 rule, Polyspace does not import each result justification. Instead, the software imports only one set of **Status**, **Severity**, and **Comment** values and applies these values to all the instances of that particular MISRA C: 2012 rule violation.

For example, suppose that you analyze your code and find violations of MISRA C: 2004 rules 16.3 and 16.5. You can justify these results by using the annotation syntax where you enter a different status and explanatory comment for each rule.

```
//polyspace-begin misra2004:16.3 [Status 1] "Explanatory comment 1"
//polyspace-begin misra2004:16.5 [Status 2] "Explanatory comment 2"

code block start;
/* This block of code contains violations of
MISRA C:2004 rules 16.3 and 16.5 */
code block end;

//polyspace-end misra2004:16.3
//polyspace-end misra2004:16.5
```

The previous violations map to MISRA C: 2012 rule 8.2. When you check your annotated code against MISRA C: 2012 rules, Polyspace imports only the first line of annotations (for rule 16.3) and applies it to all rule 8.2 results. The second line of annotations for rule 16.5 is ignored. In the **Results List** pane, all violations of rule 8.2 have the **Status** column set to **Status 1** and the **Comment** column set to **"Explanatory comment 1"**.

Note The **Output Summary** pane displays a warning message for every result where the imported annotation conflicts with the original annotation. After you import your MISRA C: 2004 annotations, check that a justified status has not been assigned to results you intend to investigate or fix.

See Also

Check MISRA C:2004 (-misra2) | Check MISRA C:2012 (-misra3)

Troubleshooting in Polyspace Code Prover Server

- “Read Error Information When Polyspace Analysis Stops” on page 12-3
- “Troubleshoot Compilation and Linking Errors” on page 12-4
- “Reduce Memory Usage and Time Taken by Polyspace Analysis” on page 12-7
- “Contact Technical Support About Issues with Running Polyspace” on page 12-11
- “Polyspace Cannot Find the Server” on page 12-14
- “Job Manager Cannot Write to Database” on page 12-15
- “Compiler Not Supported for Project Creation from Build Systems” on page 12-16
- “Slow Build Process When Polyspace Traces the Build” on page 12-22
- “Check if Polyspace Supports Build Scripts” on page 12-23
- “Troubleshooting Project Creation from MinGW Build” on page 12-25
- “Troubleshooting Project Creation from Visual Studio Build” on page 12-26
- “Error Processing Macro with Semicolon in Build System” on page 12-27
- “Resolve polyspace-autosar Error: Could Not Find Include File” on page 12-28
- “Resolve polyspace-autosar Error: Conflicting Universal Unique Identifiers (UUIDs)” on page 12-30
- “Resolve polyspace-autosar Error: Data Type Not Recognized” on page 12-31
- “Undefined Identifier Error” on page 12-33
- “Unknown Function Prototype Error” on page 12-36
- “Error Related to #error Directive” on page 12-37
- “Large Object Error” on page 12-38
- “Errors Related to Generic Compiler” on page 12-40
- “Errors Related to Keil or IAR Compiler” on page 12-41
- “Errors Related to Diab Compiler” on page 12-42
- “Errors Related to Green Hills Compiler” on page 12-44
- “Errors Related to TASKING Compiler” on page 12-46
- “Errors from In-Class Initialization (C++)” on page 12-48
- “Errors from Double Declarations of Standard Template Library Functions (C++)” on page 12-49
- “Errors Related to GNU Compiler” on page 12-50
- “Errors Related to Visual Compilers” on page 12-51
- “Conflicting Declarations in Different Translation Units” on page 12-53
- “Errors from Conflicts with Polyspace Header Files” on page 12-58
- “C++ Standard Template Library Stubbing Errors” on page 12-59
- “Lib C Stubbing Errors” on page 12-60

- “Errors from Using Namespace std Without Prefix” on page 12-62
- “Errors from Assertion or Memory Allocation Functions” on page 12-63
- “Error or Slow Runs from Disk Defragmentation and Anti-virus Software” on page 12-64
- “SQLite I/O Error” on page 12-66
- “License Error -4,0” on page 12-67

Read Error Information When Polyspace Analysis Stops

When you run a Polyspace analysis on your C/C++ code, if one or more of your files fail to compile, the analysis continues with the remaining files. You can choose to stop the analysis on compilation errors using the option `Stop analysis if a file does not compile (-stop-if-compile-error)`.

However, it is more convenient to let the analysis complete and capture all compilation errors. In a continuous integration process, you can send a notification to the build engineer with a list of compilation errors.

The compilation errors are displayed in the analysis log in addition to the options used and the various stages of analysis. The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.txt`. The lines that indicate errors begin with the `Error:` string and the lines that indicate warnings begin with the `Warning:` string. Find these lines and extract them to another text file for easier scanning.

Troubleshoot Compilation and Linking Errors

Run Polyspace verification on code that builds successfully with your compiler. Once your code builds successfully, set up a Polyspace project in one of these ways:

- Trace your build system with the `polyspace-configure` command.

The software creates an options file from your build scripts. It sets appropriate Polyspace analysis options to emulate your build options.

- If you cannot trace your build system, create a Polyspace options file manually.

Add your sources and includes to the project. Change the default analysis options, if required.

For more information, see “Prepare Scripts for Polyspace Analysis” on page 1-2.

The following issue occurs more often if you manually set up your project.

Issue

Before verification and detection of run-time errors, Polyspace compiles your code and detects compilation and linking errors. Even if your code builds successfully with your compiler, you might still get compilation errors with Polyspace.

Possible Cause: Deviations from ANSI C99 Standard

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

To guarantee absence of certain run-time errors, the default Polyspace compilation strictly follows the standard. Specific compilers allow specific deviations from this standard and follow internal algorithms to compile your code. Without explicit knowledge of your compiler behavior, Polyspace cannot accommodate those deviations. Accommodating these deviations through some arbitrary internal algorithms can compromise the final analysis results, if the Polyspace algorithm does not match your compiler’s algorithm.

Check the error message that caused the compilation failure and see if you can identify some deviation from the standard. The error message shows the line number that caused the compilation failure. If you run verification from the user interface, you can click the error message and navigate to the corresponding line of code.

Solution

Change analysis options to emulate your compiler more closely. To get past compilation issues, use these options.

Option	Purpose
"Target and Compiler" options	<p>Using these predefined options, you can specify your compiler behavior directly and work around known deviations from the standard.</p> <p>Often, setting <code>Compiler</code> (<code>-compiler</code>) appropriately is enough to emulate your compiler.</p>
<ul style="list-style-type: none"> • Preprocessor definitions (<code>-D</code>) • Command/script to apply to preprocessed files (<code>-post-preprocessing-command</code>) 	<p>Using these options, you can sometimes work around unknown deviations from the standard.</p> <p>For instance, you can use these options to replace unrecognized keywords from your preprocessed code with closely matching recognized keywords, or remove them completely. Because you do not change your source code, the options allow you to work around compilation errors while keeping your source code intact.</p>

For specific types of compilation errors, see "Troubleshoot Compilation Errors".

If you cannot solve your compilation error, contact MathWorks Technical Support and provide your compiler name for better support. See "Contact Technical Support About Issues with Running Polyspace" on page 12-11.

Possible Cause: Linking Errors

Even if a single compilation unit compiles successfully, you get a linking error because of mismatch between two compilation units. For instance, you define the same function in two `.c` files with different argument or return types.

Common compilation toolchains do not store information about function prototypes during the linking process. Therefore, despite these types of linking errors, the build does not fail. To guarantee absence of certain run-time errors, Polyspace does not continue analysis when such linking errors occur.

Solution

Fix the linking errors that Polyspace detects. Even if your build process allows these errors, you can have unexpected results during run time. For instance, if two function definitions with the same name but conflicting prototypes exist in your code, when you call the function, the result can be unexpected.

When a linking error occurs, the error message shows the location in your file where Polyspace compilation fails. Previous warning messages show the location of the conflicts that lead to the linking error. Using the line numbers in those messages (or by clicking the messages if you run analysis from the user interface), you can navigate to the location of the conflicts in your code.

For specific types of linking errors, see "Troubleshoot Compilation Errors".

Possible Cause: Conflicts with Polyspace Function Stubs

Polyspace uses its own implementation of standard library functions for more efficient verification. If your compiler redeclares and redefines a standard library function, you can get a warning or error when you invoke the function.

The error implies that Polyspace found the redeclaration but cannot find the body of your redefined library function. The verification continues to use the Polyspace implementation of the function but provides a warning. If your redefined function has a different signature from the normal signature of the function, the verification stops with an error.

Warnings and errors of this type often refer to the file `__polyspace__stdstubs.c`. This file contains prototypes for the Polyspace implementation of standard library functions. The file is located in `polyspaceroot\polyspace\verifier\cxx\polyspace_stubs\`. `polyspaceroot` is the Polyspace installation folder.

Solution

If you know the location of the file that contains the body of your redefined standard library function, add the file to your verification. For more information, see “Errors from Conflicts with Polyspace Header Files” on page 12-58.

If you do not have the function body available:

- If you see a warning of this type, you can ignore the warning. The verification results are based on Polyspace implementations of standard library functions. If your compiler redefinition closely matches the standard library function specifications, the verification results are still applicable for code compiled with your compiler.
- If you see an error:
 - 1** Define the macro `__polyspace_no_function_name` in your project. For instance, if an error occurs because of a conflict with the definition of the `sprintf` function, define the macro `__polyspace_no_sprintf`. For information on how to define macros, see *Preprocessor definitions (-D)*.

The macro disables the use of Polyspace implementations of the standard library function. The software stubs the standard library function like any other undefined function. You do not have an error because of signature mismatch with the Polyspace implementations.

- 2** Contact MathWorks Technical Support and provide information about your compiler.

For some standard library functions, such as `assert`, and memory allocation functions such as `malloc` and `calloc`, Polyspace continues to use its own implementations, even if you redefine the function and provide the function body. For more information, see “Errors from Assertion or Memory Allocation Functions” on page 12-63.

Reduce Memory Usage and Time Taken by Polyspace Analysis

In this section...

“Issue” on page 12-7

“Possible Cause: Anti-Virus Software” on page 12-7

“Possible Cause: Large and Complex Application” on page 12-7

“Possible Cause: Too Many Entry Points for Multitasking Applications” on page 12-9

Issue

The verification is stuck at a certain point for a long time. Sometimes, after the period of inactivity exceeds an internal threshold, the verification stops or you get an error message:

```
The analysis has been stopped by timeout.
```

For large projects with several hundreds of source files or millions of lines of code, you might run into the same issue in another way. The verification stops with the error message:

```
Fatal error: Not enough memory
```

If you have a multicore system with more than four processors, try increasing the number of processors by using the option `-max-processes`. By default, the verification uses up to four processors. If you have fewer than four processors, the verification uses the maximum available number. You must have at least 4 GB of RAM per processor for analysis. For instance, if your machine has 16 GB of RAM, do not use this option to specify more than four processors.

If the verification still takes too long, to improve the speed and make the verification faster, try one of the solutions below.

Possible Cause: Anti-Virus Software

In some cases, anti-virus software checks can noticeably slow down a Polyspace analysis. This reduction occurs because the software checks the temporary files produced by the Polyspace analysis.

Configure Exceptions for Polyspace Processes

Check the processes running and see if an anti-virus is consuming large amount of memory.

See “Error or Slow Runs from Disk Defragmentation and Anti-virus Software” on page 12-64.

Possible Cause: Large and Complex Application

The verification time depends on the size and complexity of your code.

If the application contains greater than 100,000 lines of code, the verification can sometimes take a long time. Even for smaller applications, the verification can take long if it involves complexities such as structures with many levels of nesting or several levels of aliasing through pointers. You can see the number of lines excluding comments towards the beginning of the analysis log in your results folder. Search for the string:

Number of lines without comments

However, if verification with the default options takes unreasonably long or stops altogether, there are multiple strategies to reduce the verification time. Each strategy involves reducing the complexity of verification in some way.

Solution: Use Polyspace Bug Finder First

Use Polyspace Bug Finder first to find defects in your code. Some defects that Polyspace Bug Finder finds can translate to a red error in Polyspace Code Prover. Once you fix these defects, use Polyspace Code Prover for a more rigorous verification.

Solution: Modularize Application

You can divide the application into multiple modules. Verify each module independently of the other modules. You can review the complete results for one module, while the verification of the other modules are still running.

- You can let the software modularize your application. The software divides your source files into multiple modules such that the interdependence between the modules is as little as possible. Use the `polyspace-modularize` command in `polyspaceroot\polyspace\bin` to create an initial. For more information on the command, use the `-h` option.
- You can perform a file-by-file verification. Each file constitutes a module by itself. See `Verify files independently (-unit-by-unit)`.

When you divide your complete application into modules, each module has some information missing. For instance, one module can contain a call to a function that is defined in another module. The software makes certain assumptions about the undefined functions. If the assumptions are broader than an actual representation of the function, you see an increase in orange checks from overapproximation. For instance, an error management function might return an `int` value that is either 0 or 1. However, when Polyspace cannot find the function definition, it assumes that the function returns all possible values allowed for an `int` variable. You can narrow down the assumptions by specifying external constraints. See `Constraint setup (-data-range-specifications)`.

When modularizing an application manually, you can follow your own modularization approach. For instance, you can copy only the critical files that you are concerned about into one module, and verify them. You can represent the remaining files through external constraints, provided you are confident that the constraints represent the missing code faithfully. For instance, the constraints on an undefined function represent the function faithfully if they represent the function return value and also reproduce other relevant side effects of the function. To specify external constraints, use the option `Constraint setup (-data-range-specifications)`.

Solution: Choose Lower Precision Level or Verification Level

If your verification takes too long, use a lower precision level or a lower verification level. Fix the red errors found at that level and rerun verification.

- The precision level determines the algorithm used for verification. Higher precision leads to greater number of proven results but also requires more verification time. For more information, see `Precision level (-O)`.
- The verification level determines the number of times Polyspace runs on your source code. For more information, see `Verification level (-to)`.

The verification results from lower precision can contain more orange checks. An orange check indicates that the analysis considers an operation suspect but cannot prove the presence or absence of a run-time error. You have to review an orange check thoroughly to determine if you can retain the operation. By increasing the number of orange checks, you are effectively increasing the time you spend reviewing the verification results. Therefore, use these strategies only if the verification is taking too long.

Solution: Reduce Code Complexity

Both for better readability of your code and for shorter verification time, you can reduce the complexity of your code. Polyspace calculates code complexity metrics from your application, and allows you to limit those metrics below predefined values.

For more information, see:

- “Code Metrics” (Polyspace Code Prover Access): List of code complexity metrics and their recommended upper limits
- “Compute Code Complexity Metrics” on page 8-44: How to set limits on code complexity metrics

Solution: Enable Approximations

Depending on your situation, you can choose scaling options to enable certain approximations. Often, warning messages indicate that you must use those options to reduce verification.

Situation	Option
Your code contains structures that are many levels deep.	Depth of verification inside structures (-k-limiting)
The verification log contains suggestions to inline certain functions.	Inline (-inline)

Solution: Remove Parts of Code

You can try to remove code from third-party libraries. The software uses stubs for functions that are not defined in files specified for the Polyspace analysis.

Although the analysis time is reduced, you can see an increase in orange checks because of Polyspace assumptions about stubbed functions. You can constrain stubbed functions using the option `Constraint setup (-data-range-specifications)`.

Possible Cause: Too Many Entry Points for Multitasking Applications

If your code is intended for multitasking and you provide many Tasks (Polyspace Code Prover), verification can take a long time. The following warning can appear:

```
Warning: Important use of shared variables have been detected,
|         verification carry on but to avoid scaling issues
|         it roughly approximates shared variables values.
|         You may consider adding -force-refined-shared-variables-analysis
|                                     option to improve results
```

If you receive this warning, it means that Polyspace is switching to a less precise analysis mode to complete the verification in a reasonable amount of time. In this less precise mode, the verification can consider some shared variables as full-range and cause orange checks from overapproximation.

Solution

Instead of using the option `-force-refined-shared-variables-analysis` to retain the precise analysis, you can reduce the number of entry points that you specify. If you know that some of your entry point functions do not execute concurrently, you do not have to specify them as separate entry points. You can call those functions sequentially in a wrapper function, and then specify the wrapper function as your entry point.

For instance, if you know that the entry point functions `task1`, `task2`, and `task3` do not execute concurrently:

- 1 Define a wrapper function `task` that calls `task1`, `task2`, and `task3` in all possible sequences.

```
void task() {
    volatile int random = 0;
    if (random) {
        task1();
        task2();
        task3();
    } else if (random) {
        task1();
        task3();
        task2();
    } else if (random) {
        task2();
        task1();
        task3();
    } else if (random) {
        task2();
        task3();
        task1();
    } else if (random) {
        task3();
        task1();
        task2();
    } else {
        task3();
        task2();
        task1();
    }
}
```

- 2 Instead of `task1`, `task2`, and `task3`, specify `task` for the option `Tasks (-entry-points)`.

For an example of using a wrapper function as an entry point, see “Configuring Polyspace Multitasking Analysis Manually” on page 7-16.

See Also

External Websites

- Resolving Scaling Problems in Code Prover

Contact Technical Support About Issues with Running Polyspace

To contact MathWorks Technical Support, use this page. You 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 *polyspaceroot* with your Polyspace installation folder:
 - UNIX — `polyspaceroot/polyspace/bin/polyspace-code-prover -ver`
 - Windows — `polyspaceroot\polyspace\bin\polyspace-code-prover -ver`

Provide Information About the Issue

Depending on the issue, provide appropriate artifacts to help Technical Support understand and reproduce the issue.

Compilation Errors

If you face compilation issues with your project, see “Troubleshoot Compilation Errors”. 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.log`. It contains the error message, the options used for the analysis and other relevant information.

- The source files related to the compilation error or the complete results folder 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.

Errors in Project Creation from Build Systems

If you face errors in creating a project from your build system, see “Troubleshoot Project Creation”.

If you are still having issues, contact technical support with debug information. To provide the debug information:

- 1 Run `polyspace-configure` at the command line with the option `-easy-debug`. For instance:
`polyspace-configure options -easy-debug pathToFolder buildCommand`

Here:

- `options` is the list of `polyspace-configure` options that you typically use.
- `buildCommand` is the build command that you use, for instance, `make`.
- `pathToFolder` is the folder where you want to store debug information, for instance, `C:\Temp\BuildLogs`. After a `polyspace-configure` run, the path provided contains a zipped file ending with `pscfg-output.zip`. The zipped file contains debug information only and does not contain source files traced in the build.

Make sure that you do not use the option `-verbose` or `-silent` after `-easy-debug`. These options reduce or modify the information logged and might make debugging difficult.

- 2 Send this zipped file ending with `pscfg-output.zip` to MathWorks Technical Support for further debugging.

You can also create the zipped file with debug information during every `polyspace-configure` run by creating an environment variable `PS_CONFIGURE_OPTIONS` and setting its value to:

```
-easy-debug pathToFolder
```

where `pathToFolder` is the folder where you want to store debug information.

Verification Result

If you are having trouble understanding a result, see “Polyspace Code Prover Access Results” (Polyspace Code Prover Access).

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.log`. It contains the options used for the analysis and other relevant information.

- The source files related to the result or the complete results folder 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 the preferences of a Polyspace desktop product to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Open the user interface of the Polyspace desktop product. Check if the server information provided is correct.

- 1 Select **Tools > Preferences**.
- 2 Select the **Server Configuration** tab. Check your server information.

For instance, the entry in **Job scheduler host name** must match the host name of the computer that forms the head node of the MATLAB Parallel Server cluster. For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Job Manager Cannot Write to Database

Message

Unable to write data to the job manager database

Possible Cause

If the computer that forms the head node of the MATLAB Parallel Server cluster cannot send data to the client computer, you see this error. The most likely reasons for the remote computer being unable to connect to the client computer are:

- Firewalls do not allow traffic from the MATLAB Job Scheduler to the client.
- The MATLAB Job Scheduler cannot resolve the short hostname of the client computer.

Workaround

Add localhost IP to configuration.

- 1 In the user interface of the Polyspace desktop products, select **Tools > Preferences**.
- 2 On the **Server Configuration** tab, in the **Localhost IP address** field, enter the IP address of your local computer.

To retrieve your IP address:

- Windows
 - 1 Open **Control Panel > Network and Sharing Center**.
 - 2 Select your active network.
 - 3 In the Status window, click **Details**. Your IP address is listed under **IPv4 address**.
- Linux — Run the `ifconfig` command and find the `inet addr` corresponding to your network connection.
- Mac — Open **System Preferences > Network**.

See Also

Related Examples

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Connection Problems Between the Client and MATLAB Job Scheduler” (Parallel Computing Toolbox)

Compiler Not Supported for Project Creation from Build Systems

Issue

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

Cause

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

For information on which compilers are supported, see “Requirements for Project Creation from Build Systems” on page 5-20.

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 *polyspaceroot\polyspace\configure\compiler_configuration*. Select the configuration that most closely corresponds to your compiler using the mapping between the configuration files and compiler names on page 12-20.
- 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 *polyspaceroot\polyspace\configure\compiler_configuration*.

- 3 Edit the contents of the file to represent your compiler. Replace the entries between the XML elements with appropriate content.
- 4 After saving the edited XML file to *polyspaceroot\polyspace\configure\compiler_configuration*, create a project automatically using your build command.

If you see errors, for instance, compilation errors, contact MathWorks Technical Support. After tracing your build command, the software compiles certain files using the compiler specifications detected from your configuration file and build command. Compilation errors might indicate issues in the configuration file.

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

Elements of Compiler Configuration File

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

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

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

XML Element	Content Description	Content Example for GNU C Compiler
<pre><compiler> ... </compiler></pre>	<p>The Polyspace compiler option that corresponds to or closely matches your compiler. The content of this element directly translates to the option Compiler in your Polyspace project or options file.</p> <p>For the complete list of compilers available, see <code>Compiler (-compiler)</code>.</p>	<p>gnu4.7</p>
<pre><preprocess_options_list> <opt> ... </opt> </preprocess_options_list></pre>	<p>The options that specify how your compiler generates a preprocessed file.</p> <p>You can use the macro <code>\$(OUTPUT_FILE)</code> if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.</p>	<p>-E</p> <p>For an example of the <code>\$(OUTPUT_FILE)</code> macro, see the existing compiler configuration file <code>cl2000.xml</code>.</p>
<pre><preprocessed_output_file> ... </preprocessed_output_file></pre>	<p>The name of file where the preprocessed output is stored.</p> <p>You can use the following macros when the name of the preprocessed output file is adapted from the source file:</p> <ul style="list-style-type: none"> • <code>\$(SOURCE_FILE)</code>: Source file name • <code>\$(SOURCE_FILE_EXT)</code>: Source file extension • <code>\$(SOURCE_FILE_NO_EXT)</code>: Source file name without extension <p>For instance, use <code>\$(SOURCE_FILE_NO_EXT).pre</code> when the preprocessor file name has the same name as the source file, but with extension <code>.pre</code>.</p>	<p>For an example of this element, see the existing compiler configuration file <code>xc8.xml</code>.</p>
<pre><src_extensions><ext> ... </ext></src_extensions></pre>	<p>The file extensions for source files.</p>	<ul style="list-style-type: none"> • c • cpp • c++

XML Element	Content Description	Content Example for GNU C Compiler
<pre><obj_extensions><ext> ... </ext></obj_extensions></pre>	The file extensions for object files.	
<pre><precompiled_header_extensions> ... </precompiled_header_extensions></pre>	The file extensions for precompiled headers (if available).	
<pre><polyspace_extra_options_list> <opt> ... </opt> <opt> ... </opt> </polyspace_extra_options_list></pre>	<p>Additional options that are used for the subsequent analysis.</p> <p>For instance, to avoid compilation errors in the subsequent analysis due to non-ANSI extension keywords, enter <code>-D keyword=value</code>, for example:</p> <pre><polyspace_extra_options_list> <opt>-D MACR01</opt> <opt>-D MACR02=VALUE</opt> </polyspace_extra_options_list></pre> <p>For more information, see Preprocessor definitions (-D).</p>	

Mapping Between Existing Configuration Files and Compiler Names

Select the configuration file in `polyspaceroot\polyspace\configure\compiler_configuration\` that most closely resembles the configuration of your compiler. Use the following table to map compilers to their configuration files.

Compiler Name	Vendor	XML File
ARM®	ARM Keil	armcc.xml
		armclang.xml
Visual C++	Microsoft	cl.xml
Clang	Not applicable	clang.xml
CodeWarrior	NXP	cw_ppc.xml
		cw_s12z.xml
cx6808	Cosmic	cx6808.xml
Diab	Wind River	diab.xml
gcc	Not applicable	gcc.xml
Green Hills	Green Hills Software	ghs_arm.xml
		ghs_arm64.xml
		ghs_i386.xml
		ghs_ppc.xml

Compiler Name	Vendor	XML File
		ghs_rh850.xml
		ghs_tricore.xml
IAR Embedded Workbench	IAR	iar.xml
		iar-arm.xml
		iar-avr.xml
		iar-msp430.xml
		iar-rh850.xml
		iar-rl78.xml
Renesas	Renesas	renesas-rh850.xml
		renesas-rl78.xml
		renesas-rx.xml
TASKING®	Altium	tasking.xml
		tasking-166.xml
		tasking-850.xml
		tasking-arm.xml
Tiny C	Not applicable	tcc.xml
TM320 and its derivatives	Texas Instruments	ti_arm.xml
		ti_c28x.xml
		ti_c6000.xml
		ti_msp430.xml
xc8 (PIC)	Microchip	xc8.xml

Slow Build Process When Polyspace Traces the Build

Issue

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

Cause

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

Solution

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

- If you trace your build from the Polyspace user interface, enter this flag in the field **Add advanced configure options**.
- If you trace your build from the DOS/ UNIX or MATLAB command line, use this flag with the `polyspace-configure` command.

For more information, see `polyspace-configure`.

Check if Polyspace Supports Build Scripts

Issue

This topic is relevant only if you are creating a Polyspace project in Windows from your build scripts.

When Polyspace traces your build script in a Windows console application other than `cmd.exe`, the command fails. However, the build command by itself executes to completion.

For instance, your build script executes to completion from the Cygwin shell. However, when Polyspace traces the build, the build script throws an error.

Possible Cause

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

Polyspace attempts to trace your build script with the assumption that the script runs to completion in `cmd.exe`. Therefore, even if your script runs to completion in the alternate console application, when Polyspace traces the build, the script can fail.

Solution

Make sure that your build script executes to completion in the `cmd.exe` interface. If the build executes successfully, create a wrapper `.bat` file around your script and trace this file.

For instance, before you trace a build command that executes to completion in the Cygwin shell, do one of the following:

- Launch the Cygwin shell from `cmd.exe` and then run your build script. For instance, if you use a script `build.sh` to build your code, enter the following command at the DOS command line:

```
cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
```

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

For instance, enter the following command at the DOS command line:

```
cmd.exe /C path_to_script
```

`path_to_script` is the full path to your build script. For instance, `C:\my_scripts\build.sh`.

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

If the steps complete successfully, trace the build command after launching it from `cmd.exe`. For instance, on the command-line, do the following to create a Polyspace options file.

- 1 Enter your build commands in a `.bat` file.

```
rem @echo off
cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
```

Name the file, for instance, `launching.bat`.

- 2 Trace the build commands in the .bat file and create a Polyspace options file.

```
"C:\Program Files\MATLAB\R2017b\polyspace\bin\polyspace-configure.exe"  
-output-options-file myOptions.txt launching.bat
```

You can now run `polyspace-code-prover-server` on the options file.

Troubleshooting Project Creation from MinGW Build

Issue

You create a project from a MinGW build, but get an error when running an analysis on the project. The error message comes from using one of these keywords: `__declspec`, `__cdecl`, `__fastcall`, `__thiscall` or `__stdcall`.

Cause

When you create a project from a MinGW build, the project uses a GNU compiler. Polyspace does not recognize these keywords for the GNU compilers.

Solution

Replace these keywords with equivalent keywords just for the purposes of analysis.

Before analysis, for the option Preprocessor definitions (-D), enter:

- `__declspec(x)=__attribute__((x))`
- `__cdecl=__attribute__((__cdecl__))`
- `__fastcall=__attribute__((__fastcall__))`
- `__thiscall=__attribute__((__thiscall__))`
- `__stdcall=__attribute__((__stdcall__))`

If you are running Polyspace on the command line in a UNIX shell, add double quotes around the -D option. For instance, use:

```
"-D __cdecl=__attribute__((__cdecl__))"
```

Troubleshooting Project Creation from Visual Studio Build

You can run `polyspace-configure` on a Visual Studio build and extract information from the build to create a Polyspace project or options file.

You can trace your Visual Studio build in one of the following ways:

- Build your Visual Studio project completely at the command line with `msbuild` while tracing this build with `polyspace-configure`.

In this workflow, you run `polyspace-configure` on an `msbuild` command with a Visual Studio project (`.vcxproj`) file. For instance, in a Visual Studio 2019 developer prompt, enter the following:

```
polyspace-configure msbuild TestProject.vcxproj /t:Rebuild
```

- Build your Visual Studio project in the Visual Studio IDE while tracing this build with `polyspace-configure`.

Run `polyspace-configure` on the `devenv.exe` executable to open the Visual Studio IDE, build your project or solution within the IDE, and then close the IDE.

If running `polyspace-configure` on the `msbuild` command does not work properly, do the following:

- 1 Stop the `msbuild` process.
- 2 Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
- 3 Restart `polyspace-configure` on `msbuild`, this time using the `/nodereuse:false` option. For instance:

```
polyspace-configure msbuild TestProject.vcxproj /t:Rebuild /nodereuse:false
```

See Also

`polyspace-configure`

Error Processing Macro with Semicolon in Build System

Issue

You see this error when creating a Polyspace project or options file from your build system:

```
Could not process macro containing a semicolon
```

Cause

Some options in your build system use semicolons in the replacement list of a macro. Automatic project creation from build systems does not support this usage. For instance, a macro `OK` with this replacement list can cause issues:

```
{printf("OK");flush();}
```

The use of semicolons in replacement lists is not supported because a Polyspace project or options file created from your build system itself uses semicolon separators to separate macro definitions. For details on the Polyspace options that define macros, see:

- `Preprocessor definitions (-D)`: This option defines macros.
- `-options-for-sources`: This option collects several macro definitions, separated by semicolon.

Solution

Define the macro in a header file instead of in the build system. For instance, define the macro `OK` like this in a header file:

```
#ifndef OK_DEFINED
#define OK_DEFINED
#define OK {printf("OK");flush();}
#endif
```

Provide the header file only for the purposes of Polyspace analysis using the option `Include (-include)`.

Resolve polyspace-autosar Error: Could Not Find Include File

Issue

When creating a Polyspace project from an AUTOSAR description, by default, the software searches only in the source folder for `#include-d` files. If an include file is not present directly in the source folder, Polyspace cannot find it. For instance, the missing include file can be in a subfolder of the source folder.

You see a warning like this when creating a Polyspace project from AUTOSAR XML and source files:

```
Could not find include file "MemMap.h"
```


If you use variables or functions declared in the missing include file, you can also see errors later.

Possible Solutions

If you want to expand the search path for include files, explicitly add new folders.

- In the Polyspace user interface, use the field **Specify additional include folders**.
See “Create Polyspace Analysis Configuration from AUTOSAR Specifications” on page 2-12.
- At the command-line, use the option `-I`.
See `polyspace-autosar`.

You can find the possible include folders to add in several ways:

- If an include file is in a subfolder of the source code folder, the analysis proposes a resolution hint with one or more include folders that might contain the missing include file. To see the resolution hints, in the file `psar_project.xhtml`, click the  button on the upper left, then click **Behaviors**. On the **Behaviors** tab, below the errors in the code extraction phase, click the link to see a summary of code-extraction diagnostics with possible resolution hints.

Extract implementation code for **89** AUTOSAR behaviors with proof artifacts:

- [noRunnableImplementation \(30\)](#)
- [error_noRunnableImplementationTopFileError \(3\)](#)
- [error_atLeastOneRunnableInFileThatDoesNotCompile \(23\)](#)
- [subsetOfRunnablesImplementation \(3\)](#)
- [allRunnablesImplementation \(30\)](#)

[See summary of code-extraction diagnostics with possible resolution hints](#)

Execution reported errors and warnings. [Reported errors](#) [See detailed log messages](#)

You can see resolution hints, that is, possible include folders to add, that would resolve some of the missing include files.

Instead of fixing individual code extraction errors using the resolution hints, you can also download a file with all options that implement the hints. On the summary page, click the link **Download polyspace-autosar options**.

Summary of polyspace-autosar code-extraction diagnostics

Lists diagnostics that are reported when extracting the implementation-code of one or more AUTOSAR behaviors. Each diagnostic may have "resolution-hints" which are specific to the class of error. Resolution-hints can translate to polyspace-autosar options that you may add to your project [Download polyspace-autosar options](#)

You can use the downloaded text file with the `polyspace-autosar` option `-options-file` to implement the resolution hints in one shot.

- If you use a build command for compilation, you can extract compilation options such as path to includes from your build command. See "Create Polyspace Analysis Configuration from AUTOSAR Specifications" on page 2-12.

You might also simply know the architecture of the system to locate the missing include folders.

See Also

`polyspace-autosar`

Related Examples

- "Create Polyspace Analysis Configuration from AUTOSAR Specifications" on page 2-12

Resolve polyspace-autosar Error: Conflicting Universal Unique Identifiers (UUIDs)

Issue

If multiple elements in an AUTOSAR description contain the same Universal Unique Identifier (UUID) or a single element contains multiple UUIDs, one of these errors can occur when creating a Polyspace project from the AUTOSAR XML files:

- Elements `"/pkg/swc002/bhv/twosec"` and `"/pkg/swc002/bhv/step"` in file `$file{C:/AUTOSAR/arxml/mSwc002_component.arxml}{332}` have the same UUID `"5bdd54d5-50ae-4ad3-bdea-e0b0ab2bcab6"`. Each of these elements should have its own unique UUID.
- Element `"/AUTOSAR"` has both UUID `"ECUS:6b411924-70da-40a5-85f5-65d5630ea0cb"` and `"ECUS:48ea040a-c40d-4ee0-ae61-8a6ccc9cb18d"`. You should specify only one UUID.

Possible Solutions

Investigate why multiple elements have the same UUID, or the same element has two different UUID-s. Fix the issue if possible.

If you do not own the AUTOSAR XML with the conflicting UUID-s or do not want to fix the issue because it represents work in progress, use the options `-Eno-autosar-xmlReaderSameUuidForDifferentElements` and `-Eno-autosar-xmlReaderTooManyUuids`. The analysis ignores the issue of conflicting UUID-s and continues with a warning. For conflicting UUID-s, the analysis stores the last element read.

The subsequent analyses continue to use the warning mode. To revert back to the error mode, use the option `-Eautosar-xmlReaderSameUuidForDifferentElements` and `-Eautosar-xmlReaderTooManyUuids`.

See Also

`polyspace-autosar`

Related Examples

- “Create Polyspace Analysis Configuration from AUTOSAR Specifications” on page 2-12

Resolve polyspace-autosar Error: Data Type Not Recognized

Issue

When creating a Polyspace project from an AUTOSAR description, the software parses your AUTOSAR XML specifications and imports the data types that are required by the Software Components in the scope of verification. If your code uses a data type that is not in the Software Component specification, the analysis does not recognize this data type.

You see an error such as:

```
Identifier "LaneDetectionVar" is undefined
```

when creating a Polyspace project from AUTOSAR XML and source files. The error suggests that a data type used in your source code is not recognized.

Possible Solutions

You can force import of data types that are not defined for Software Components that you are verifying. Use the option `-autosar-datatype`. See `polyspace-autosar`.

You can find the already imported data types using the file `autosar_model_key_elements.html` in the AUTOSAR subfolder of your project folder. In the `DataTypes` section of the HTML, the file shows:

- Automatically imported data types using this format:

<code>indirect</code>	<code>pkg.types.app.Array_2_n320to320</code>
<code>indirect</code>	<code>pkg.types.app.Boolean</code>


The text `indirect` in the first column indicates that the data types are automatically imported.

- Explicitly imported data types using this format:

<code>name</code>	<code>tst003.typ.app.Boolean</code>
-------------------	-------------------------------------

The text `name` in the first column indicates that the data type `tst003.typ.app.Boolean` is explicitly imported for the analysis.

In some cases, the analysis proposes a resolution hint using additional data types imported from the ARXML as a possible match for the unrecognized data type. To see the resolution hints, in the file

`psar_project.xhtml`, click the  button on the upper left, then click **Behaviors**. On the **Behaviors** tab, below the errors in the code extraction phase, click the link to see a summary of code-extraction diagnostics with possible resolution hints.

Extract implementation code for **89** AUTOSAR behaviors with proof artifacts:

- [noRunnableImplementation \(30\)](#)
- [error_noRunnableImplementationTopFileError \(3\)](#)
- [error_atLeastOneRunnableInFileThatDoesNotCompile \(23\)](#)
- [subsetOfRunnablesImplementation \(3\)](#)
- [allRunnablesImplementation \(30\)](#)

[🔗 See summary of code-extraction diagnostics with possible resolution hints](#)

Execution reported errors and warnings. [🔗 Reported errors](#) [🔗 See detailed log messages](#)

You can see resolution hints, that is, possible data types to add, that would resolve some of the issues related to unrecognized data types.

Instead of fixing individual code extraction errors using the resolution hints, you can also download a file with all options that implement the hints. On the summary page, click the link **Download polyspace-autosar options**.

Summary of polyspace-autosar code-extraction diagnostics

Lists diagnostics that are reported when extracting the implementation-code of one or more AUTOSAR behaviors. Each diagnostic may have "resolution-hints" which are specific to the class of error.

Resolution-hints can translate to polyspace-autosar options that you may add to your project [🔗 Download polyspace-autosar options](#)

You can use the downloaded text file with the polyspace-autosar option `-options-file` to implement the resolution hints in one shot.

See Also

`polyspace-autosar`

Related Examples

- “Create Polyspace Analysis Configuration from AUTOSAR Specifications” on page 2-12

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 of the Polyspace desktop products, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Code Prover).

- At the command line, use the flag `-I` with the `polyspace-code-prover-server` 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)`. For a sample header file, see “Gather Compilation Options Efficiently” on page 5-28.

Possible Cause: Declaration Embedded in #ifdef Statements

The variable is declared in a branch of an `#ifdef macro_name` preprocessor directive. For instance, the declaration of a variable `max_power` occurs as follows:

```
#ifdef _WIN32
    #define max_power 31
#endif
```

Your compilation toolchain might consider the macro `macro_name` as implicitly defined and execute the `#ifdef` branch. However, the Polyspace compilation might not consider the macro as defined. Therefore, the `#ifdef` branch is not executed and the variable `max_power` is not declared.

Solution

To work around the compilation error, do one of the following:

- Use **Target & Compiler** options to directly specify your compiler. For instance, to emulate a Visual C++ compiler, set the **Compiler** to `visual12.0`. See “Target and Compiler”.
- Define the macro explicitly using the option `Preprocessor definitions (-D)`.

Note If you create a Polyspace by tracing your build commands, most **Target & Compiler** options are automatically set.

Possible Cause: Project Created from Non-Debug Build

This can be a possible cause only if the undefined identifier occurs in an `assert` statement (or an equivalent Visual C++ macro such as `ASSERT` or `VERIFY`).

Typically, you come across this error in the following way. You create a Polyspace project from a build system in non-debug mode. When you run an analysis on the project, you face a compilation error from an undefined identifier in an `assert` statement. You find that the identifier `my_identifier` is defined in a `#ifndef NDEBUG` statement, for instance as follows:

```
#ifndef NDEBUG
int my_identifier;
#endif
```

The C standard states that when the `NDEBUG` macro is defined, all `assert` statements must be disabled.

Most IDEs define the `NDEBUG` macro in their build systems. When you build your source code in your IDE in non-debug mode, code in a `#ifndef NDEBUG` statement is removed during preprocessing. For instance, in the preceding example, `my_identifier` is not defined. If `my_identifier` occurs only in `assert` statements, it is not used either, because `NDEBUG` disables `assert` statements. You do not have compilation errors from undefined identifiers and your build system executes successfully.

Polyspace does not disable `assert` statements even if `NDEBUG` macro is defined because the software uses `assert` statements internally to enhance verification.

When you create a Polyspace project from your build system, if your build system defines the `NDEBUG` macro, it is also defined for your Polyspace project. Polyspace removes code in a `#ifndef NDEBUG` statement during preprocessing, but does not disable `assert` statements. If `assert` statements in your code rely on the code in a `#ifndef NDEBUG` statement, compilation errors can occur.

In the preceding example:

- The definition of `my_identifier` is removed during preprocessing.
- `assert` statements are not disabled. When `my_identifier` is used in an `assert` statement, you get an error because of undefined identifier `my_identifier`.

Solution

To work around this issue, create a Polyspace project from your build system in debug mode. When you execute your build system in debug mode, `NDEBUG` is not defined. When you create a Polyspace project from this build, `NDEBUG` is not defined for your Polyspace project.

Depending on your project settings, use the option that enables building in debug mode. For instance, if your build system is gcc-based, you can define the `DEBUG` macro and undefine `NDEBUG`:

```
gcc -DDEBUG=1 -UNDEBUG *.c
```

Alternatively, you can disable the `assert` statements in your preprocessed code using the option `Preprocessor definitions (-D)`. However, Polyspace will not be able to emulate the `assert` statements.

Unknown Function Prototype Error

Issue

During the compilation phase, the software displays a warning or error message about unknown function prototype.

```
the prototype for function 'myfunc' is unknown
```

The message indicates that Polyspace cannot find a function prototype. Therefore, it cannot identify the data types of the function argument and return value, and has to infer them from the calls to the function.

To determine the data types for such functions, Polyspace follows the C99 Standard (ISO/IEC 9899:1999, Chapter 6.5.2.2: Function calls).

- The return type is assumed to be `int`.
- The number and type of arguments are determined by the first call to the function. For instance, if the function takes one `double` argument in the first call, for subsequent calls, the software assumes that it takes one `double` argument. If you pass an `int` argument in a subsequent call, a conversion from `int` to `double` takes place.

During the linking phase, if a mismatch occurs between the number or type of arguments or the return type in different compilation units, the verification stops. For more information, see “Conflicting Declarations in Different Translation Units” on page 12-53.

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 of the Polyspace desktop products, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Code Prover).

- At the command line, use the flag `-I` with the `polyspace-code-prover-server` command.

For more information, see `-I`.

Error Related to #error Directive

Issue

The analysis stops with a message containing a #error directive. For instance, the following message appears: #error directive: !Unsupported platform; stopping!.

Cause

You typically use the #error directive in your code to trigger a fatal error in case certain macros are not defined. Your compiler implicitly defines the macros, therefore the error is not triggered when you compile your code. However, the default Polyspace compilation does not consider the macros as defined, therefore, the error occurs.

For instance, in the following example, the #error directive is reached only if the macros `__BORLANDC__`, `__VISUALC32__` or `__GNUC__` are not defined. If you use a GNU C compiler, for instance, the compiler considers the macro `__GNUC__` as defined and the error does not occur. However, if you use the default Polyspace compilation, it does not consider the macros as defined.

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#define MYINT int
#elif defined(__GNUC__)
#define MYINT long
#else
#error !Unsupported platform; stopping!
#endif
```

Solution

For successful compilation, do one of the following:

- Specify a compiler such as `visual12.0` or `gnu4.9`. Specifying a compiler defines some of the compilation flags for the analysis.

For more information, see `Compiler (-compiler)`.

- If the available compiler options do not match your compiler, explicitly define one of the compilation flags `__BORLANDC__`, `__VISUALC32__`, or `__GNUC__`.

For more information, see `Preprocessor definitions (-D)`.

Large Object Error

Issue

The analysis stops during compilation with a message indicating that an object is too large.

Cause

The error happens when the software detects an object such as an array, union, structure, or class, that is too big for the pointer size of the selected target.

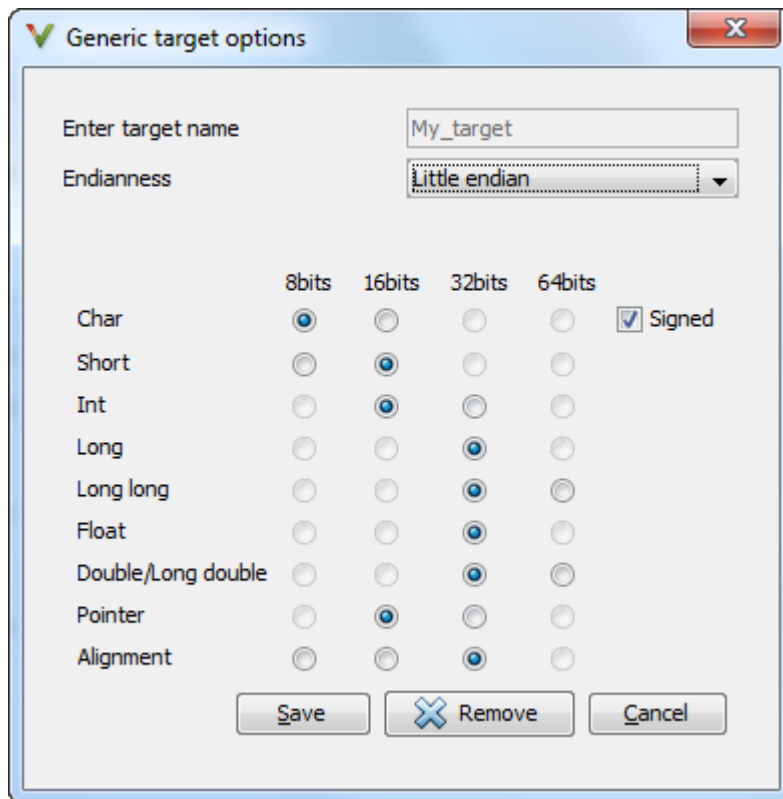
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be $2^{16}-1$ bytes. However, you declare a structure as follows:

- ```
struct S
{
 char tab[65536];
}s;
```
- ```
struct S
{
    char tab[65534];
    int val;
}s;
```

Solution

- 1 Check the pointer size that you specified through your target processor type. For more information, see `Target processor type (-target)`.

For instance, in the following, the pointer size for a custom target `My_target` is 16 bits.



- 2 Change your code or specify a different pointer size.

For instance, you can:

- Declare an array of smaller size in the structure.

If you are using a predefined target processor type, the pointer size is likely to be the same as the pointer size on your target architecture. Therefore, your declaration might cause errors on your target architecture.

- Change the pointer size of the target processor type that you specified, if possible.

Otherwise, specify another target processor type with larger pointer size or define your own target processor type. For more information on defining your own processor type, see `Generic target options`.

Note Polyspace imposes an internal limit of 128 MB on the size of data structures. Even if your target processor type specification allows data structures of larger size, this internal limit constrains the data structure sizes.

Errors Related to Generic Compiler

If you use a generic compiler, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Issue

The analysis stops with an error message related to a non-ANSI C keyword, for instance, `data` or attributes such as `__attribute__((weak))`.

Depending on the location of the keyword, the error message can vary. For instance, this line causes the error message: `expected a ";"`.

```
data int tab[10];
```

Cause

The generic Polyspace compiler supports only ANSI C keywords. If you use a language extension, the generic compiler does not recognize it and treats the keyword as a regular identifier.

Solution

Specify your compiler by using the option `Compiler (-compiler)`.

If your compiler is not directly supported or is not based on a supported compiler, you can use the generic compiler. To work around the compilation errors:

- If the keyword is related to memory modelling, remove it from the preprocessed code. For instance, to remove the `data` keyword, enter `data=` for the option `Preprocessor definitions (-D)`.
- If the keyword is related to an attribute, remove attributes from the preprocessed code. Enter `__attribute__(x)=` for the option `Preprocessor definitions (-D)`.

If your code has this line:

```
void __attribute__((weak)) func(void);
```

And you remove attributes, the analysis reads the line as:

```
void func(void);
```

When you use these workarounds, your source code is not altered.

Errors Related to Keil or IAR Compiler

If you use the compiler, Keil or IAR, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Missing Identifiers

Issue

The analysis stops with the error message, `expected an identifier`, as if an identifier is missing. However, in your source code, you can see the identifier.

Cause

If you select Keil or IAR as your compiler, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs.

For a list of keywords that are removed, see “Supported Keil or IAR Language Extensions” on page 5-23.

Solution

Specify that Polyspace must not remove the keywords during preprocessing. Define the macros `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__`.

For more information, see `Preprocessor definitions (-D)`.

Errors Related to Diab Compiler

If you choose `diab` for the option `Compiler` (`-compiler`), you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a keyword specific to the Diab compiler. For instance, you see an error related to the `restrict` keyword.

Cause

You typically use a compiler flag to enable the keyword. The Polyspace analysis does not enable these keywords by default. You have to make Polyspace aware of your compiler flags.

The Polyspace analysis does not enable these keywords by default to prevent compilation errors. Another user might not enable the keyword and instead use the keyword name as a regular identifier. If Polyspace treats the identifier as a keyword, a compilation error will occur.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface of the Polyspace desktop products, you can enter the command-line option in the field `Other` (Polyspace Code Prover). 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 Green Hills Compiler

If you choose `greenhills` for the option `Compiler (-compiler)`, you encounter this issue.

Issue

During Polyspace analysis, you see an error related to vector data types specific to Green Hills target `rh850`. For instance, you see an error related to identifier `__ev64_u16__`.

Cause

When compiling code using the Green Hills compiler with target `rh850`, to enable single instruction multiple data (SIMD) vector instructions, you specify two flags:

- `-rh850_simd`: You enable intrinsic functions that support SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev64_u16__`
 - `__ev64_s16__`
 - `__ev64_u32__`
 - `__ev64_s32__`
 - `__ev64_u64__`
 - `__ev64_s64__`
 - `__ev64_opaque__`
 - `__ev128_opaque__`
- `-rh850_fpsimd`: You enable intrinsic functions that support floating-point SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev128_f32__`
 - `__ev256_f32__`

The Polyspace analysis does not enable SIMD support by default. You must identify your compiler flags to Polyspace.

Solution

In your Polyspace analysis, use the command-line option `-compiler-parameter`. In the user interface, you can enter the command-line option in the `Other (Polyspace Code Prover)` field, under the **Advanced Settings** in the **Configuration** pane.

- `-rh850_simd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_simd`
- `-rh850_fpsimd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_fpsimd`

Note

- `__ev128_opaque__` is 16 bytes aligned in Polyspace.
 - `__ev256_f32__` is 32 bytes aligned in Polyspace.
-

Errors Related to TASKING Compiler

If you choose `tasking` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a Special Function Register data type.

Cause

When compiling with the TASKING compiler, you typically use the following compiler flags to specify where Special Function Register (SFR) data types are declared:

- `--cpu=xxx`: The compiler implicitly `#includes` the file `sfr/regxxx.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` (Polyspace Code Prover). 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 5-20.

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
<pre>class Test { public: static int m_number; };</pre>	<pre>int Test::m_number = 0;</pre>

Errors from Double Declarations of Standard Template Library Functions (C++)

Consider the following code.

```
#include <list>

void f(const std::list<int*>::const_iterator it) {}
void f(const std::list<int*>::iterator it) {}
void g(const std::list<int*>::const_reverse_iterator it) {}
void g(const std::list<int*>::reverse_iterator it) {}
```

The declared functions belong to `list` container classes with different iterators. However, the software generates the following compilation errors:

```
error: function "f" has already been defined
error: function "g" has already been defined
```

You would also see the same error if, instead of `list`, the specified container was `vector`, `set`, `map`, or `deque`.

To avoid the double declaration errors, do one of the following:

- Deactivate automatic stubbing of standard template library functions. For more information, see `No STL stubs (-no-stl-stubs)`.
- Define the following Polyspace preprocessing directives:
 - `__PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_VECTOR_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_SET_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_MAP_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_DEQUE_CONST_ITERATOR_DIFFER_ITERATOR__`

For example, for the given code, run analysis at the command line with the following flag. The flag defines the appropriate directive for the `list` container.

```
-D __PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__
```

For more information on defining preprocessor directives, see `Preprocessor definitions (-D)`.

Errors Related to GNU Compiler

If you choose `gnu` for the option `Compiler` (`-compiler`), you can encounter this issue.

Issue

The Polyspace analysis stops with a compilation error.

Cause

You are using certain advanced compiler-specific extensions that Polyspace does not support. See “Limitations”.

Solution

For easier portability of your code, avoid using the extensions.

If you want to use the extensions and still analyze your code, wrap the unsupported extensions in a preprocessor directive. For instance:

```
#ifdef POLYSPACE
    // Supported syntax
#else
    // Unsupported syntax
#endif
```

For regular compilation, do not define the macro `POLYSPACE`. For Polyspace analysis, enter `POLYSPACE` for the option `Preprocessor definitions` (`-D`).

If the compilation error is related to assembly language code, use the option `-asm-begin -asm-end`.

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
<pre>#pragma pack(4) #include "type.h"</pre>	<pre>struct A { char c ; int i ; } ;</pre>	<pre>#pragma pack(2) #include "type.h"</pre>

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
    struct A
```

```
^
  detected during compilation of secondary translation unit
"test2.cpp"
```

To continue the analysis, use the option `Ignore pragma pack directives (-ignore-pragma-pack)`.

C++/CLI

Polyspace does not support Microsoft C++/CLI, a set of language extensions for .NET programming.

You can get errors such as:

```
error: name must be a namespace name
|         using namespace System;
```

Or:

```
error: expected a declaration
|         public ref class Form1 : public System::Windows::Forms::Form
```

Conflicting Declarations in Different Translation Units

Issue

The analysis shows an error or warning similar to one of these error messages:

- Declaration of [...] is incompatible with a declaration in another translation unit ([...])

This message appears when the conflicting declarations do not come from the same header file.

- When one of the conflicting declarations is in a header file.

Declaration of [...] had a different meaning during compilation of [...] ([...])

This message appears when the conflicting declarations come from the same header file included in different source files.

The error indicates that the same variable or function or data type is declared differently in different translation units. The conflicting declarations violate the One Definition Rule (cf. C++Standard, ISO/IEC 14882:2003, Section 3.2). When conflicting declarations occur, Polyspace Code Prover does not choose a declaration and continue analysis.

Common compilation toolchains often do not store data type information during the linking process. The conflicting declarations do not cause errors with your compiler. Polyspace Code Prover follows stricter standards for linking to guarantee the absence of certain run-time errors.

To identify the root cause of the error:

- 1 From the error message, identify the two source files with the conflicting declarations.

For instance, an error message looks like this message:

```
C:\field.h, line 1: declaration of class "a_struct" had
    a different meaning during compilation of "file1.cpp"
| struct a_struct {
|
| Detected during compilation of secondary translation unit "file2.cpp"
```

The message shows that the structure `a_struct` has a conflicting declaration in `file1.cpp` and `file2.cpp`, both of which include the header file `field.h`.

An alternative error message can look like this:

```
C:\field2.h, line 1: declaration of class "a_struct" had
    is incompatible with a declaration in another translation unit
| the other declaration is at line 1 of field1.h"
| struct a_struct {
|
| Detected during compilation of secondary translation unit "file2.cpp"
```

The message shows that the structure `a_struct` has a conflicting declaration in `field2.h` and `field.h`. The header file `field2.h` is included in the source file `file2.cpp`.

- 2 Try to identify the conflicting declarations in the source files.

Otherwise, open the translation units containing these files. Sometimes, the translation units or preprocessed files show the conflicting declarations more clearly than the source files because

the preprocessor directives, such as `#include` and `#define` statements, are replaced appropriately and the macros are expanded.

- a Rerun the analysis with the flag `-keep-relaunch-files` so that all translation units are saved. In the user interface, enter the flag for the option `Other` (Polyspace Code Prover).

The analysis stops after compilation. The translation units or preprocessed files are stored in a zipped file `ci.zip` in a subfolder `.relaunch` of the results folder.

- b Unzip the contents of `ci.zip`.

The preprocessed files have the same name as the source files. For instance, the preprocessed file with `file1.cpp` is named `file1.ci`.

When you open the preprocessed files at the line numbers stated in the error message, you can spot the conflicting declarations.

Possible Cause: Variable Declaration and Definition Mismatch

A variable declaration does not match its definition. For instance:

- The declaration and definition use different data types.
- The variable is declared as signed, but defined as unsigned.
- The declaration and definition uses different type qualifiers.
- The variable is declared as an array, but defined as a non-array variable.
- For an array variable, the declaration and definition use different array sizes.

In this example, the code shows a linking error because of a mismatch in type qualifiers. The declaration in `file1.c` does not use type qualifiers, but the definition in `file2.c` uses the `volatile` qualifier.

<code>file1.c</code>	<code>file2.c</code>
<pre>extern int x; void main(void) { /* Variable x used */ }</pre>	<pre>volatile int x;</pre>

In these cases, you can typically spot the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the variable declaration matches its definition.

Possible Cause: Function Declaration and Definition Mismatch

A function declaration does not match its definition. For instance:

- The declaration and definition use different data types for arguments or return values.
- The declaration and definition use a different number of arguments.
- A variable-argument or `varargs` function is declared in one function, but it is called in another function without a previous declaration.

In this case, the error message states that the required prototype for the function is missing.

In this example, the code shows a linking error because of a mismatch in the return type. The declaration in `file1.c` has return type `int`, but the definition in `file2.c` has return type `float`.

<code>file1.c</code>	<code>file2.c</code>
<pre>int input(void); void main() { int val = input(); }</pre>	<pre>float input(void) { float x = 1.0; return x; }</pre>

In these cases, you can typically find the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the function declaration matches its definition.

Even if your build process allows these errors, you can have unexpected results during run time. If a function declaration and definition with conflicting prototypes exist in your code, when you call the function, the result can be unexpected.

For a variable-argument or `varargs` function, declare the function before you call it. If you do not want to change your source code, you can work around this linking error.

- 1 Add the function declaration in a separate file.
- 2 Only for the purposes of verification, `#include` this file in every source file by using the option `Include (-include)`.

Possible Cause: Conflicts from Unrelated Declarations

You use the same identifier name for two unrelated objects. These are some common reasons for unrelated objects in the same Polyspace project:

- You intended to declare the objects `static` so that they do not have external linkage, but omitted the `static` specifier.
- You declared the same object in several source files instead of putting the declaration in a header file and including in the source files.
- You created a Polyspace project from a build command using the `polyspace-configure` command. The build command created several independent binaries, but files involved in all the binaries were collected in one Polyspace project.

Solution

Depending on the root cause for unrelated objects using the same name, use an appropriate solution.

If your Polyspace project was created from a build command and source files for independent binaries were clubbed together, split the project into modules when tracing your build command. See `polyspace-configure`.

Possible Cause: Macro-dependent Definitions

A variable definition is dependent on a macro being defined earlier. One source file defines the macro while another does not, causing conflicts in variable definitions.

In this example, `file1.cpp` and `file2.cpp` include a header file `field.h`. The header file defines a structure `a_struct` that is dependent on a macro definition. Only one of the two files, `file2.cpp`, defines the macro `DEBUG`. The definition of `a_struct` in the translation unit with `file1.cpp` differs from the definition in the unit with `file2.cpp`.

<code>file1.cpp</code>	<code>file2.cpp</code>
<pre>#include "field.h" int main() { a_struct s; init_a_struct(&s); return 0; }</pre>	<pre>#define DEBUG #include <string.h> #include "field.h" void init_a_struct(a_struct* s) { memset(s, 0, sizeof(*s)); }</pre>
<p>field.h:</p> <pre>struct a_struct { int n; #ifdef DEBUG int debug; #endif };</pre>	

When you open the preprocessed files `file1.ci` and `file2.ci`, you see the conflicting declarations.

<code>file1.ci</code>	<code>file2.ci</code>
<pre>struct a_struct { int n; };</pre>	<pre>struct a_struct { int n; int debug; };</pre>

Solution

Avoid macro-dependent definitions. Otherwise, fix the linking errors. Make sure that the macro is either defined or undefined on all paths that contain the variable definition.

Possible Cause: Keyword Redefined as Macro

A keyword is redefined as a macro, but not in all files.

In this example, `bool` is a keyword in `file1.cpp`, but it is redefined as a macro in `file2.cpp`.

file1.cpp	file2.cpp
<pre>#include "bool.h" int main() { return 0; }</pre>	<pre>#define false 0 #define true (!false) #include "bool.h"</pre>
<p>bool.h:</p> <pre>template <class T> struct a_struct { bool flag; T t; a_struct() { flag = true; } };</pre>	

Solution

Be consistent with your keyword usage throughout the program. Use the keyword defined in a standard library header or use your redefined version.

Possible Cause: Differences in Structure Packing

A `#pragma pack(n)` statement changes the structure packing alignment, but not in all files. See also “`#pragma Directives`” (Polyspace Code Prover).

In this example, the default packing alignment is used in `file1.cpp`, but a `#pragma pack(1)` statement enforces a packing alignment of 1 byte in `file2.cpp`.

file1.cpp	file2.cpp
<pre>int main() { return 0; }</pre>	<pre>#pragma pack(1) #include "pack.h"</pre>
<p>pack.h:</p> <pre>struct a_struct { char ch; short sh; };</pre>	

Solution

Enter the `#pragma pack(n)` statement in the header file so that it applies to all source files that include the header.

Errors from Conflicts with Polyspace Header Files

Issue

You see compilation errors from header files included by Polyspace.

For instance, the error message refers to one of the subfolders of *polyspaceroot*\polyspace\verifier\cxx\include.

Typically, the error message is related to a standard library function.

Cause

If your compiler defines a standard library function or another construct and you do not provide the path to your compiler header files, Polyspace uses its own implementation of the function.

If your compiler definitions differ from the corresponding Polyspace definitions, the verification stops with an error.

Solution

Specify the folder containing your compiler header files.

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

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Code Prover).

- At the command line, use the flag `-I` with the `polyspace-code-prover-server` command.

For more information, see `-I`.

For compilation with GNU C on UNIX-based platforms, use `/usr/include`. On embedded compilers, the header files are typically in a subfolder of the compiler installation folder. Examples of include folders are given for some compilers.

- Wind River Diab: For instance, `/apps/WindRiver/Diab/5.9.4/diab/5.9.4.8/include/`.
- IAR Embedded Workbench: For instance, `C:\Program Files\IAR Systems\Embedded Workbench 7.5\arm\inc`.
- Microsoft Visual Studio: For instance, `C:\Program Files\Microsoft Visual Studio 14.0\VC\include`.

Consult your compiler documentation for the path to your compiler header files. Alternatively, see “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.

C++ Standard Template Library Stubbing Errors

Issue

The analysis stops with an error message that refers to class templates such as `map` and `vector` from the Standard Template Library.

Often, the error message states that either an operator cannot be found or more than one operator matches the given operands.

Cause

Polyspace software provides an efficient implementation of all class templates from the Standard Template Library (STL). If your source code redeclares the templates, the analysis can stop with an error message.

Solution

To use your own implementations of templates from the Standard Template Library:

- 1 Disable the Polyspace implementations using the option `No STL stubs (-no-stl-stubs)`.
- 2 Add the folders containing your implementations to the verification.
 - In the user interface, add the folder to your project.
For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Code Prover).
 - At the command line, use the flag `-I` with the `polyspace-code-prover-server` command.
For more information, see `-I`.

Note Using your own template definitions can cause other compilation and linking errors.

Lib C Stubbing Errors

Extern C Functions

Some functions may be declared inside an extern "C" { } block in some files, but not in others. In this case, the linkage is different which causes a link error, because it is forbidden by the ANSI standard.

Original code:

```
extern "C" {
    void* memcpy(void*, void*, int);
}
class Copy
{
public:
    Copy() {};
    static void* make(char*, char*, int);
};
void* Copy::make(char* dest, char* src, int size)
{
    return memcpy(dest, src, size);
}
```

Error message:

Pre-linking C++ sources ...

```
<results_dir>/test.cpp, line 2: error: declaration of function "memcpy"
is incompatible with a declaration in another translation unit
(parameters do not match)
|         the other declaration is at line 4096 of "__polyspace__stdstubs.c"
|         void* memcpy(void*, void*, int);
|         ^
|         detected during compilation of secondary translation unit "test.cpp"
```

The function `memcpy` is declared as an external "C" function and as a C++ function. It causes a link problem. Indeed, function management behavior differs whether it relates to a C or a C++ function.

When such error happens, the solution is to homogenize declarations, i.e. add extern "C" { } around previous listed C functions.

Another solution consists in using the permissive option `-no-extern-C`. It removes all extern "C" declarations.

Functional Limitations on Some Stubbed Standard ANSI Functions

- `signal.h` is stubbed with functional limitations: `signal` and `raise` functions do not follow the associated functional model. Even if the function `raise` is called, the stored function pointer associated to the signal number is not called.
- No jump is performed even if the `setjmp` and `longjmp` functions are called.
- `errno.h` is partially stubbed. Some math functions do not set `errno`, but instead, generate a red error when a range or domain error occurs with **ASRT** checks.

You can also use the compile option `POLYSPACE_STRICT_ANSI_STANDARD_STUBS` (-D flag). This option only deactivates extensions to ANSI C standard libC, including the functions `bzero`, `bcopy`,

bcmp, chdir, chown, close, fchown, fork, fsync, getlogin, getuid, geteuid, getgid, lchown, link, pipe, read, pread, resolvepath, setuid, setegid, seteuid, setgid, sleep, sync, symlink, ttyname, unlink, vfork, write, pwrite, open, creat, sigsetjmp, __sigsetjmp, and siglongjmpare.

Errors from Using Namespace `std` Without Prefix

Issue

The Polyspace analysis stops with an error message such as:

```
error: the global scope has no "modfl"
```

The line highlighted in the error uses a function from the standard library without the `std::` prefix.

Cause

Some compilers allow using members of the standard library namespace without explicitly specifying the `std::` prefix. For such compilers, your code can contain lines like this:

```
using ::mblen;
```

where `mblen` is a member of the C++ standard library. Polyspace compilation considers the members as part of the global namespace and shows an error.

Solution

It is a good practice to qualify members of the standard library with the `std::` prefix. For instance, to use the `mblen` function in the preceding example, rewrite the line as:

```
using std::mblen;
```

To continue to retain the current code and work around the Polyspace error, use the analysis option - `using-std`. If you are running the analysis in the Polyspace user interface, enter the option in the **Other** field. See **Other**.

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 or Slow Runs from Disk Defragmentation and Anti-virus Software

Issue

In some cases, anti-virus software checks can noticeably slow down a Polyspace analysis. This reduction occurs because the software checks the temporary files produced by the Polyspace analysis.

You see noticeably slow analysis for a simple project or 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 anti-virus software is running on your machine.

After starting an analysis, check the processes running and see if an anti-virus process is causing large amount of CPU usage (and possibly memory usage).

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the anti-virus software. Or, configuring exception rules for the anti-virus software to allow Polyspace to run without a failure.

For instance, you can try the following:

- Configure the anti-virus software to whitelist the Polyspace executables.

For instance, in Windows, with the anti-virus software Windows Defender, you can add an exclusion for the Polyspace installation folder C:\Program Files\Polyspace\R2019a, in particular, the .exe files in the subfolder polyspace\bin and the .exe files starting with ps_ in the subfolder bin\win64.

- Configure the anti-virus software to exclude your temporary folder, for example, C:\Temp, from the checking process.

SQLite I/O Error

Issue

When you try to run Polyspace, you get this error message:

Cause

Polyspace uses an SQLite database for storing results. This error can appear when SQLite databases are saved on NFS (Network File System) folders.

Solution

Check the folder where you save Polyspace results. For instance, if you run Polyspace at the command line, check the option `-results-dir`.

If the folder is an NFS folder, use a local folder instead.

License Error -4,0

Issue

When you try to run Polyspace, you get this error message:

```
License Error -4,0
```

Possible Cause: Another Polyspace Instance Running

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.

Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder

If you run Polyspace on generated code in the Simulink user interface or in the MATLAB Coder app, you can get a license error if you try to run a subsequent analysis in the Polyspace user interface. You get the error even if the previous run is over.

Solution

Run the subsequent analysis using the method that you used before, that is, in the Simulink user interface or MATLAB Coder app.

If you want to run the analysis in the Polyspace user interface, close Simulink or MATLAB Coder and then rerun the analysis.

