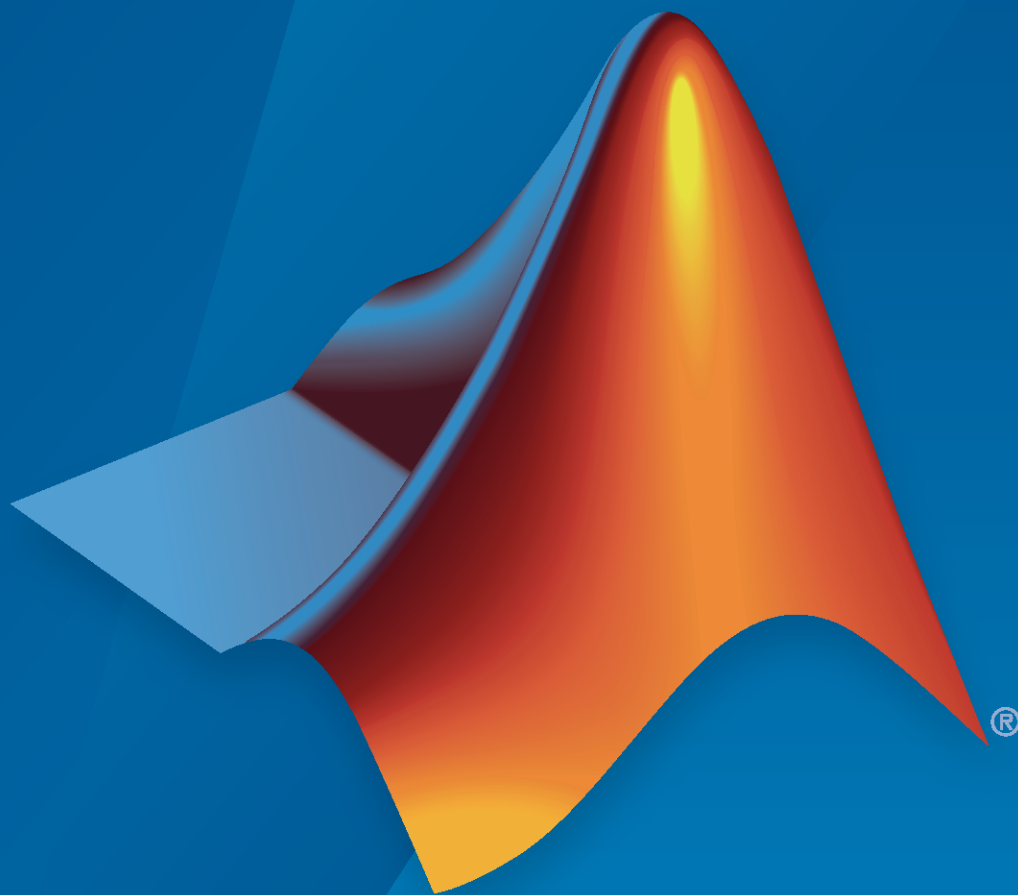


Polyspace[®] Code Prover[™] Server[™]

Reference



R2020a

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™ Reference

© 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)

1	Commands
2	Analysis Options
3	Analysis Options, Command-Line Only

Commands

polyspace-access

(DOS/UNIX) Manage upload or export of Polyspace results from the Polyspace Access web interface

Syntax

```
polyspace-access -host hostname [configuration options] -create-project  
projectFolder
```

```
polyspace-access -host hostname [configuration options] -upload  
pathToFolderOrZipFile [upload options]
```

```
polyspace-access -host hostname [configuration options] -export  
findingsToExport -output filePath [export options]
```

```
polyspace-access -host hostname [configuration options] -set-unassigned-  
findings findingsToAssign -owner userToAssign -source-contains pattern [set  
unassigned findings options]
```

```
polyspace-access -host hostname [configuration options] -list-project [  
findingsPath]
```

```
polyspace-access -host hostname [configuration options] -set-role role -user  
username -project-path projectFolderOrFindingsPath
```

```
polyspace-access -host hostname [configuration options] -unset-role -user  
username -project-path projectFolderOrFindingsPath
```

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

```
polyspace-access -generate-migration-commands metrics_dir -output-folder-path  
dir [generate migration commands options]
```

```
polyspace-access -host hostname [configuration options] -migrate -option-  
file-path dir [-dryrun]
```

Description

`polyspace-access -host hostname [configuration options] -create-project projectFolder` creates a project folder in the Polyspace Access web interface. The folder can be at the top of the project hierarchy or a subfolder under an existing project folder.

`polyspace-access -host hostname [configuration options] -upload pathToFolderOrZipFile [upload options]` uploads Polyspace results from a folder or a zipped file to the Polyspace Access database. Use the `upload options` to specify a project folder other than `public`.

`polyspace-access -host hostname [configuration options] -export findingsToExport -output filePath [export options]` exports project results from a project in the Polyspace Access database to a text file whose location you specify with `filePath`. You specify the project using either the full path in Polyspace Access or the run ID. Use this command to export findings to other tools that you use for custom reports. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -host hostname [configuration options] -set-unassigned-findings findingsToAssign -owner userToAssign -source-contains pattern [set unassigned findings options]` assigns owners to unassigned results in a project in the Polyspace Access database. You specify the project using either the full path in Polyspace Access or the run ID. Use the `set unassigned findings options` to assign findings from different source files or different groups of source files to different owners. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -host hostname [configuration options] -list-project [findingsPath]` without the optional argument `findingsPath` lists the paths to all projects in the Polyspace Access database and their last run IDs. If you specify the full path to a project with the argument `findingsPath`, the command lists the last run ID.

`polyspace-access -host hostname [configuration options] -set-role role -user username -project-path projectFolderOrFindingsPath` assigns a role `role` to the user specified by `username` for the specified project or project folder. A user role set for a project folder applies to all project findings under that folder. You specify the project using either the full path in Polyspace Access or the last run ID. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -host hostname [configuration options] -unset-role -user username -project-path projectFolderOrFindingsPath` removes any role previously assigned to `username` for the specified project or project folder. You specify the project using either the full path in Polyspace Access or the last run ID. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -encrypt-password` encrypts the password you use to log into Polyspace Access. Use the output of this command as the argument of the `-encrypted-password` option when you write automation scripts to interact with Polyspace Access.

`polyspace-access -generate-migration-commands metrics_dir -output-folder-path dir [generate migration commands options]` generates scripts to migrate projects from the path `metrics_dir` in Polyspace Metrics to Polyspace Access. The command stores the scripts in `dir`. To specify which project findings to migrate, use `generate migration commands options`.

`polyspace-access -host hostname [configuration options] -migrate -option-file-path dir [-dryrun]` migrates projects from Polyspace Metrics to Polyspace Access using the scripts generated with the `-generate-migration-commands` command. To view which projects are migrated without actually migrating the projects, use the `-dryrun` option.

Examples

Encrypt Password and Set Configuration Options

Polyspace Access requires login credentials. You can enter them at the command line when you execute a command, or you can generate an encrypted password that you use in automation scripts.

To encrypt your password, use the `-encrypt-password` command and enter your Polyspace Access credentials.

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

The command uses the user name and password you enter to generate an encrypted password.

```
login: jsmith
password:
CRYPTED_PASSWORD KEAGKAMJMCOPLFKPKOHOJNDJCBACFJBL
Command Completed
```

If you manage your analysis findings through automated scripts, create a variable to store the connection configuration and login credentials. Use this variable in your script, or at the command line to avoid entering your credentials when you execute a command.

```
set LOGIN=-host my-company-server -port 1234 ^
-protocol https -login jsmith ^
-encrypted-password KEAGKAMJMCOPLFKPKOHOJNDJCBACFJBL
polyspace-access %LOGIN% -create-project myProject
```

Create a Project Folder with Restricted Access and Upload to Folder

Suppose that you want to upload a set of findings to Polyspace Access and authorize only some team members to view these findings.

Create a project folder `Restricted` at the top of the project hierarchy.

```
polyspace-access -host my-company-server -port 1234 ^
-create-project Restricted
```

Set user roles for users `aUser` and `bUser`, authorizing them to access the project folder as contributors.

```
polyspace-access -host my-company-server ^
-port 1234 -set-role contributor ^
-user aUser -user bUser -project Restricted
```

Aside from the creator of the project folder and the previous two users, no other user can view or access any findings uploaded to `Restricted`.

Upload project findings under `Restricted`.

```
polyspace-access -host my-company-server -port 1234 ^
-upload C:\Polyspace_Workspace\projectName\Module_1 ^
-parent-project Restricted
```

The uploaded findings are stored under `Restricted/projectName`.

Assign Results to Component Owners and Export Assigned Results

If you follow a component-based development approach, you can assign analysis findings by component to their respective owners.

Get a list of projects currently stored on the Polyspace Access database.

```
polyspace-access -host my-company-server ^
-list-project
```


The command outputs a list of project findings paths and their last run ID.

```
Connecting to https://my-company-server:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
multimodule/vxWorks_demo (Code Prover) RUN_ID 16
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 8
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

Assign all red and orange run-time error findings to the owner of all the files in Component_A of project vxWorks_demo. Perform the same assignment for the owner of Component_B. To specify the vxWorks_demo project, use the run ID.

```
polyspace-access -host my-company-server ^
-set-unassigned-findings 16 ^
-owner A_owner -source-contains Component_A ^
-owner B_owner -source-contains Component_B ^
-rte Red -rte Orange
```

-source-contains Component_A matches all files with a file path that contains Component_A.

-source-contains Component_B matches all files with a file path that contains Component_B, but excludes files with a file path that contains Component_A.

After you assign findings, export the findings and generate .csv files for each owner containing the findings assigned to them.

```
polyspace-access -host my-company-server ^
-export 16 ^
-output C:\Polyspace_Workspace\myResults.csv ^
-output-per-owner
```

The command generates file myResults.csv containing all findings from the project with run ID 16. The command also generates files myResults.csv.A_owner.csv and myResults.csv.B_owner.csv on the same file path.

Migrate Projects from Metrics to Polyspace Access

If you have projects stored on a Polyspace Metrics server, you can migrate them to the Polyspace Access database. Log in to your Metrics server to complete this operation.

Generate migration scripts for the projects you want to migrate. Specify the folder path of the location where the projects are stored, for example C:\Users\jsmith\AppData\Roaming\Polyspace_RLDatas\results-repository

```
polyspace-access -generate-migration-commands ^
C:\Users\jsmith\AppData\Roaming\Polyspace_RLDatas\results-repository ^
-output-folder-path C:\Polyspace_Workspace\toMigrate -project-date-after 2017-06
```

The command generates migration scripts for all projects in the specified metrics folder that were uploaded on or after June 2017. The scripts are stored in folder C:\Polyspace_Workspace\toMigrate.

Use the `-dryrun` option to check which projects will be migrated.

```
polyspace-access -host my-company-server ^
-migrate -option-file-path ^
C:\Polyspace_Workspace\toMigrate -dryrun
```

The command output contains a list of projects. Inspect it to ensure that you are migrating the correct projects.

To perform the migration, rerun the last command without the `-dryrun` option.

Input Arguments

Connect and Login

hostname — Machine host name

string

Fully qualified host name of the machine hosting the Polyspace Access **Gateway** service. You must specify a host name with all `polyspace-access` commands, except the `-generate-migration-commands` and `-encrypt-password` commands.

Example: `-host my-company-server`

configuration options — Options to configure connection to Polyspace Access

string

Options to specify connection configuration and login credentials.

Configuration Options

Option	Description
<code>-port portNumber</code>	Port number of the Polyspace Access Gateway service. The default port number is 9443.
<code>-protocol http https</code>	HTTP protocol used to access Polyspace Access. The default protocol is https.
<code>-login username</code> <code>-encrypted-password ENCRYPTED_PASSWD</code>	Login credentials you use to interact with Polyspace Access. The argument of <code>-encrypted-password</code> is the output of the <code>-encrypt-password</code> command. If you do not use these two options, you are prompted to enter your credentials at the command line.

Miscellaneous Options

Option	Description
<code>-output file_path</code>	File path where you store command outputs
<code>-tmp-dir folder_path</code>	Folder path where you store temporary files generated by the <code>polyspace-access</code> commands. The default folder path is <code>tmp/ps_results_server</code> on Linux and <code>C:/Users/%username%/AppData/Local/Temp/ps_results_server</code> on Windows.

Option	Description
-log	File path where you store the command output log. By default the command does not generate a log file.
-h	Display the help information for polyspace-access or one of its commands.

Create New Folder

projectFolder — Name of project folder

string

Project folder path specified as a string. If the name includes spaces, use double quotes. Specify the full path to folders nested under a parent folder.

If your folder path involves a folder that does not already exist, the folder is created.

Example: `-create-project topFolder`

Example: `-create-project "topFolder/subFolder/subSubFolder"`

Upload Results

pathToFolderOrZipFile — Path to folder or zipped file containing analysis results

string

Folder or zipped file path specified as a string. The folder or zipped file contains analysis results you want to upload to Polyspace Access. Specify the path of the folder containing the *.psbf, *.pscp, or *.rte file, or the path of the parent of this folder to upload multiple analysis runs.

For instance, for the Bug Finder results stored in `C:\Polyspace_Workspace\myProject\Module_1\BF_results\ps_results.psbf`, specify the path to `BF_results` or to `Module_1`. If the path name includes spaces, use double quotes.

Example: `-upload C:\Polyspace_Workspace\myProject\Module_1\BF_results`

Example: `-upload C:\Polyspace_Workspace\myProject\Module_1\ -project projectFolder`

upload options — Options to specify where to upload results

string

Options to specify path to project folder where you upload results.

Option	Description
<code>-parent-project projectFolder</code>	Path of the parent project folder under which you upload project findings. If you do not specify a parent project folder, projects are upload to the public folder.

Option	Description
<code>-project</code> <i>projectFolderOrFindingsPath</i>	<p>If the FOLDER you specify for <code>-upload</code> contains only one analysis run, for instance <code>ps_results.psbf</code>, this option is optional. Use <code>-project</code> to rename project findings, or omit it to use the project name from your Polyspace analysis.</p> <p>If the FOLDER you specify for <code>-upload</code> contains more than one analysis run, or if you specify the parent folder of the results folder, this option is mandatory. Use <code>-project</code> to create a project folder under which all the analysis runs are stored.</p>

Export Results

findingsToExport – Project findings path or run ID

string

Path or run ID of the project findings that you export. Polyspace assigns a unique run ID to each analysis run you upload. If the path name includes spaces, use double quotes. To get the project findings path or last run ID, use `-list-project`.

Example: `-export "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-export 4`

filePath – Path to file containing command output

string

Path to the file that stores the output of the command when you specify the `-output` option. This option is mandatory with the `-export` command.

Example: `-output C:\Polyspace_Workspace\myResults.txt`

export options – Options to specify which findings to export

string

Options to specify where to export findings, and which subset of findings you export. Use these options to export findings to other tools you use to create custom reports or other custom review templates.

Option	Description
<code>-output</code> <i>file_path</i>	File path where you export the findings. This option is mandatory with the <code>-export</code> command.
<code>-new-findings</code>	Export only new findings compared to the previous analysis (previous upload with the same project name).
<code>-output-per-owner</code>	Use this option to generate files that only contain findings assigned to a particular user. The files are stored on the path you specify with <code>-output</code> .
<code>-rte</code> <i>color</i>	<p>Type of RTE finding to export. Specify <code>All</code>, <code>Red</code>, <code>Gray</code>, <code>Orange</code>, or <code>Green</code>.</p> <p>To specify more than one argument, call the option for each argument. For example, <code>-rte Red -rte Orange</code>.</p>

Option	Description
<code>-defects impact</code>	Impact of DEFECTS findings to export. Specify All, High, Medium, or Low. To specify more than one argument, call the option for each argument. For example, <code>-defects Medium -defects Low</code> .
<code>-custom-coding-rules</code>	Export all custom coding rules findings.
<code>-coding-rules</code>	Export all coding rules findings.
<code>-code-metrics</code>	Export all code metrics findings.
<code>-global-variables</code>	Export all global variables findings.
<code>-review-status status</code>	Review status of the findings to export. Specify New, Unreviewed, Unassigned, Toinvestigate, Tofix, Justified, Noactionplanned, Notadefect, Other, or Annotated. To specify more than one argument, call the option for each argument. For example, <code>-review-status Tofix -review-status Toinvestigate</code> .
<code>-severity severity</code>	Severity of the findings to export. Specify All, High, Medium, or Low. To specify more than one argument, call the option for each argument. For example, <code>-severity High -severity Low</code> .
<code>-open-findings-for-sqo sqo_level</code>	Software quality objective or SQO level that must be satisfied. Specify a number from 1 to 6 for <code>sqo_level</code> . If you specify an SQO level, the <code>polyspace-access</code> command exports only open findings that must be fixed or justified to satisfy the requirements of this level. For more information on the SQO levels, see “Software Quality Objectives” (Polyspace Code Prover Access). The SQO levels 1 to 6 specify an increasingly stricter set of requirements defined in terms of Polyspace results. The requirements are predefined but you can customize them in the Polyspace Access web interface. For instance, SQO level 2 in Code Prover requires that you must not have unjustified red checks. This specification means that if you use <code>-open-findings-for-sqo</code> with a level higher than 2, all red checks are exported and must be subsequently fixed or justified. If you want to impose this requirement in the earlier SQO level 1, you can customize level 1 in the Polyspace Access web interface.

You can also use a combination of options. For instance, `-coding-rules -severity High` exports coding rule violations that have been assigned a status of High in the Polyspace Access web interface.

Assign Findings

findingsToAssign — Project findings path or run ID

string

Path or run ID of the project findings that you assign to a user. Polyspace assigns a unique run ID to each analysis run you upload. If the path name includes spaces, use double quotes. To get the project findings path or last run ID, use `-list-project`.

Example: `-set-unassigned-findings "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-set-unassigned-findings 4`

userToAssign — Polyspace Access user name

string

User name of user you assign as owner of unassigned findings. To assign multiple owners, call the option for each user.

Each call to `-owner` must be paired with a call to `-source-contains`.

Example: `-user jsmith`

pattern — Pattern to match against file path

string

Pattern to match against file path of project source files. To match file paths for all source files, use `-source-contains /`.

Enter a substring from the file path. You cannot use regular expressions.

When you call this option more than once, each instance excludes patterns from previous instances. For example, `-source-contains foo -source-contains bar` matches all file paths that contain `foo`, then all file paths that contain `bar` excluding paths that contain `foo`.

When you assign findings to multiple owners, call this option for each call to `-owner`.

Example: `-source-contains main`

set unassigned findings options — Options to specify which findings to assign

string

Options to assign all findings or only a subset based on component or individual source files. To make an assignment, specify a pattern to match against the folder or file paths to assign.

Option	Description
<code>-rte color</code>	Type of RTE finding to assign. Specify All, Red, Gray, Orange, or Green. To specify more than one argument, call the option for each argument. For example, <code>-rte Red -rte Orange</code> .
<code>-defects impact</code>	Impact of DEFECTS findings to assign. Specify All, High, Medium, or Low. To specify more than one argument, call the option for each argument. For example, <code>-defects Medium -defects Low</code> .
<code>-custom-coding-rules</code>	Assign all custom coding rules findings.
<code>-coding-rules</code>	Assign all coding rules findings.
<code>-code-metrics</code>	Assign all code metrics findings.
<code>-global-variables</code>	Assign all global variables findings.

Option	Description
<code>-review-status status</code>	Review status of the findings to assign. Specify New, Unreviewed, Unassigned, Toinvestigate, Tofix, Justified, Noactionplanned, Notadefect, Other, or Annotated. To specify more than one argument, call the option for each argument. For example, <code>-review-status Tofix -review-status Toinvestigate</code> .
<code>-severity severity</code>	Severity of the findings to assign. Specify All, High, Medium, or Low. To specify more than one argument, call the option for each argument. For example, <code>-severity High -severity Low</code> .
<code>-dryrun</code>	Display command output without making any assignment. Use this option to check that your assignments are correct.

List Projects

findingsPath — Project findings path

string

Path of the project findings. Specify this optional argument with `-list-project` to get the path and the last run ID of the corresponding project findings. If the path name includes spaces, use double quotes.

Example: `-list-project "public/Examples/Bug_Finder_Example (Bug Finder)"`

Set and Unset User Roles

role — Level of access permissions for project folder or findings

owner | contributor | forbidden

Level of access to project folder and findings for a user.

- **owner:** User can move, rename, or delete specified project folders or findings and review their content.
- **contributor:** User can review content of specified project folder or findings.
- **forbidden:** User cannot access specified project folder or findings. Set this role to restrict the access of a user to a set of project findings inside a project folder that is accessible to the user.

Example: `-set-role contributor`

username — Polyspace Access user name

string

Polyspace Access user name.

Example: `-user jsmith`

projectFolderOrFindingsPath — Project folder or findings path

string

Path of a project folder or project findings. When `projectFolderOrFindingsPath` is the path to a project folder, the user role you set applies to all subfolders and project findings under that folder. If

the path name includes spaces, use double quotes. To get the project folder or findings path, use `-list-project`.

Example: `-project-path "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-project-path public`

Migrate Results from Metrics to Polyspace Access

metrics_dir – Folder path of Polyspace Metrics projects

string

Path of folder containing the Polyspace Metrics projects you want to migrate to Polyspace Access.

Example: `-generate-migration-commands C:\Users\%username%\AppData\Roaming\Polyspace_RLDatas\results-repository`

dir – Output folder for migration scripts

string

Path to folder that stores the output of `-generate-migration-commands`. Do not specify an existing folder.

Example: `local/Polyspace_Workspace/migration_scripts`

generate migration commands options – Options to specify which projects to migrate

string

Option	Description
<code>-output-folder-path</code> <i>dir</i>	Folder path where you want to store the generated command files. Do not specify an existing folder.
<code>-max-project-runs</code> <i>int</i>	Number of most recent analysis runs you want to migrate for each project. For instance, to migrate only the last two analysis runs of a project, specify 2.
<code>-project-date-after</code> <i>YYYY[-MM[-DD]]</i>	Only migrate results that were uploaded to Polyspace Metrics on or after the specified date.
<code>-product</code> <i>productName</i>	Product used to analyze and produce project findings, specified as <code>bug-finder</code> , <code>code-prover</code> , or <code>polyspace-ada</code> .
<code>-analysis-mode</code> <i>mode</i>	Analysis mode use to generate project findings, specified as <code>integration</code> or <code>unit-by-unit</code> .

See Also

Topics

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

“Send Email Notifications with Polyspace Code Prover Results”

Introduced in R2018b

polyspace-autosar Command

(DOS/UNIX) Run Polyspace Code Prover on code implementation of AUTOSAR software components

Syntax

```
polyspace-autosar -create-project projectFolder -arxml-dir arxmlFolder -  
sources-dir codeFolder [-sources-dir codeFolder] [OPTIONS]
```

```
polyspace-autosar -update-project prevProjectFile [OPTIONS]
```

```
polyspace-autosar -update-and-clean-project prevProjectFile [OPTIONS]
```

```
polyspace-autosar -help
```

Description

`polyspace-autosar -create-project projectFolder -arxml-dir arxmlFolder -sources-dir codeFolder [-sources-dir codeFolder] [OPTIONS]` checks the code implementation of AUTOSAR software components for run-time errors and violation of data constraints in the corresponding AUTOSAR XML specifications. The analysis parses the AUTOSAR XML specifications (.arxml files) in `arxmlFolder`, modularizes the code implementation (.c files) in `codeFolder` based on the specifications, and runs Code Prover on each module for the checks. The Code Prover results are stored in `projectFolder`. After analysis, you can open the project `psar_project.psprj` from `projectFolder` in the Polyspace user interface. You can view the results for each software component individually or upload them to Polyspace Metrics for an overview.

You can use additional options for troubleshooting, for instance, to only perform certain parts of the update and track down an issue or to provide extra header files or define macros.

`polyspace-autosar -update-project prevProjectFile [OPTIONS]` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update uses the html file `prevProjectFile` from the previous analysis and only reanalyzes the code implementation of software components that changed since that analysis.

You can use additional options for troubleshooting.

`polyspace-autosar -update-and-clean-project prevProjectFile [OPTIONS]` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update only reanalyzes the code implementation of software components that changed since the previous analysis. A clean update also removes information about software components that are out of date. For instance, if you use an additional option to force the update for specific software components and other SWC-s have also changed, a clean update removes those other SWC-s from the Polyspace project.

You can use additional options for troubleshooting.

`polyspace-autosar -help` shows all options available for `polyspace-autosar`.

Examples

Run Code Prover on All Software Components

Suppose your ARXML files are in a folder `arxml` and your C source files in a folder `code` in the current folder.

Run Code Prover on all software components defined in your ARXML files. Store the results in a folder `polyspace` in the current folder.

```
polyspace-autosar -create-project polyspace -arxml-dir arxml -sources-dir code
```

The analysis creates a Polyspace project with several modules. Each module collects the C code implementation of a software component. The analysis runs Code Prover on each module and checks the code for run-time errors or mismatch with ARXML specifications.

After analysis, you can open the results in several ways. See “Create Polyspace Analysis Configuration from AUTOSAR Specifications”.

Update an ARXML or code file. For instance, in Linux®, you can `touch` a file to indicate an update. Check if the updates affected results of the Code Prover analysis. For an updated analysis, provide the project file `psar_project.html` created in the previous step.

```
polyspace-autosar -update-project polyspace\psar_project.xhtml
```

If you update an ARXML file, the entire analysis is repeated. If you update your source code, the analysis is repeated only for software components whose code implementation was updated.

Run Code Prover on Specific Software Components

Instead of running Code Prover on all software components, check specific software components only.

For instance, suppose a software component has the fully qualified path `pkg.component.bhv`. You can run Code Prover only on this software component.

```
polyspace-autosar -create-project polyspace -arxml-dir arxml -sources-dir code  
-autosar-behavior pkg.component.bhv
```

You can run Code Prover on all software components but later choose to update the analysis for specific software components only.

```
polyspace-autosar -update-project polyspace\psar_project.xhtml  
-autosar-behavior pkg.component.bhv
```

If you do not reanalyze a software component that has been updated, the analysis shows that the software component might be out of date.

You can also update the analysis for specific software components and remove all traces of other software components.

```
polyspace-autosar -update-and-clean-project polyspace\psar_project.xhtml
-autosar-behavior pkg.component.bhv
```

Input Arguments

projectFolder — Folder to store Polyspace results

string

Folder name, specified as a string (in double quotes). If the folder exists, it must be empty.

Example: "C:\Polyspace_Projects\proj_swcl"

arxmlFolder — Folder containing ARXML files

string

Folder name, specified as a string (in double quotes). You can omit the double quotes if your folder paths do not contain spaces.

UNC paths are not supported for the folder name.

Example: "C:\arxml_swcl"

codeFolder — Folder containing C files

string

Folder name, specified as a string (in double quotes). You can omit the double quotes if your folder paths do not contain spaces.

To specify multiple root folders containing sources, repeat the `-sources-dir` option. If you specify multiple root folders, they must not overlap. For instance, one root folder cannot be a subfolder of the other.

UNC paths are not supported for the folder name.

Example: "C:\code_swcl"

prevProjectFile — Path to psar_project.html

string

Path to the previously created project file `psar_project.html`, specified as a string (in double quotes). You can omit the double quotes if your folder paths do not contain spaces.

Example: "C:\Polyspace_Projects\proj1\psar_project.html"

[OPTIONS] — Options to control project creation

string

Options to control creation of Polyspace project and subsequent analysis. You primarily use the options for troubleshooting, for instance, to only perform certain parts of the update and narrow down an issue or to provide extra header files or define macros.

General options

Option	Description
-verbose	<p>Save additional information about the various phases of command execution (verbose mode). The file <code>psar_project.log</code> and other auxiliary files store this additional information.</p> <p>If an error occurs in command execution, the error message is stored in a separate file, irrespective of whether you enable verbose mode. Running in verbose mode only stores the various phases of execution. You can use this information to see when an error was introduced.</p>
-options-file <i>OPTION_FILE</i>	<p>Use an options file to supplement or replace the command line options. In the options file, specify each option on a separate line. Begin a line with <code>#</code> to indicate comments.</p> <p>An options file <code>opts.txt</code> can look like this:</p> <pre data-bbox="865 863 1224 1031"># Store Polyspace results -create-project polyspace # ARXML Folder -arxml-dir arxml # SOURCE Folder -sources-dir code</pre> <p>You can run <code>polyspace-autosar</code> as:</p> <pre data-bbox="865 1119 1438 1146">polyspace-autosar -options-file opts.txt</pre> <p>If an option that is directly specified with the <code>polyspace-autosar</code> command conflicts with an option in the options file, the directly specified option is used. For instance, in this example, the folder <code>proj</code> is used to save the Polyspace project.</p> <pre data-bbox="865 1356 1438 1413">polyspace-autosar -create-project proj -options-file opts.txt</pre> <p>You typically use an options file to store and reuse options that are common to multiple projects.</p>

Options to control update of project

If you update a project, by default, the analysis results are updated for all AUTOSAR SWCs behaviors with respect to any change in the arxml files or C source code since the last analysis. These options allow you to control the update.

Option	Description
<p><code>-autosar-behavior</code> <code>AUTOSAR_QUALIFIED_NAME</code></p>	<p>Check the implementation of software components whose internal behavior-s are specified by <code>AUTOSAR_QUALIFIED_NAME</code>. The default analysis considers all software components present in the ARXML specifications.</p> <p>To specify multiple software components, repeat the option. Alternatively, use regular expressions to specify a group of software components under the same package.</p> <p>For instance:</p> <ul style="list-style-type: none"> • To specify the software component whose internal behavior has the fully qualified name <code>pkg.component.bhv</code>, use: <pre>-autosar-behavior pkg.component.bhv</pre> • To specify the software components whose internal behavior-s have fully qualified names beginning with <code>pkg.component</code>, use: <pre>-autosar-behavior pkg.component\.*</pre> <p>The <code>\.</code> represents the package name separator <code>.</code> (dot) and the <code>.*</code> represents any number of characters.</p>
<p><code>-do-not-update-autosar-prove-environment</code></p>	<p>Do not read the ARXML specifications. Use ARXML specifications stored from the previous analysis.</p> <p>Use this option during project updates to compare the code against previous specifications. Unless you use this option, project updates read the entire ARXML specifications again.</p>
<p><code>-do-not-update-extract-code</code></p>	<p>Do not read the C source code. Use source code stored from the previous analysis.</p> <p>Use this option during project updates to compare the previous source code against ARXML specifications. Unless you use this option, project updates consider all changes to the source code since the previous analysis.</p>

Option	Description
<p><code>-do-not-update-verification</code></p>	<p>Read the ARXML specifications and C code implementation only but do not run the Code Prover analysis.</p> <p>Use this option during project updates to investigate errors introduced in the ARXML specifications or compilation errors introduced in the source code. You can first fix these issues and then run the Code Prover analysis.</p>

Options to control parsing of ARXML specifications

Option	Description
<p><code>-autosar-datatype</code> <i>AUTOSAR_QUALIFIED_NAME</i></p>	<p>Import definition of AUTOSAR data types specified by <i>AUTOSAR_QUALIFIED_NAME</i>. The default analysis only imports data types specified in the internal behavior of software components that you verify.</p> <p>To specify multiple data types, repeat the option. Alternatively, use regular expressions to specify all data types under the same package.</p> <p>For instance:</p> <ul style="list-style-type: none"> • To specify a data type that has the fully qualified name <code>pkg.datatypes.type</code>, use: <code>-autosar-datatype pkg.datatypes.type</code> • To specify data types that have fully qualified names beginning with <code>pkg.datatypes</code>, use: <code>-autosar-behavior pkg.datatypes\.*</code> <p>The <code>\.</code> represents the package name separator <code>.</code> (dot) and the <code>.*</code> represents any number of characters.</p> <ul style="list-style-type: none"> • To force import of all data types, use: <code>-autosar-datatype .*\..*</code>

Option	Description
<p>-Eautosar -xmlReaderSameUuidForDifferentElements</p> <p>-Eno -autosar -xmlReaderSameUuidForDifferentElements</p>	<p>If multiple elements in the ARXML specifications have the same universal-unique-identifier (uuid), use these options to toggle between a warning and an error.</p> <p>The default analysis stops with an error if the issue happens. To convert to a warning, use -Eno -autosar -xmlReaderSameUuidForDifferentElements. For conflicting UUID-s, the analysis stores the last element read and continues with a warning.</p> <p>The subsequent executions continue to use the warning mode. To revert back to an error, use -Eautosar -xmlReaderSameUuidForDifferentElements.</p>
<p>-Eautosar -xmlReaderTooManyUuids</p> <p>-Eno -autosar -xmlReaderTooManyUuids</p>	<p>If the same element in the ARXML specifications has different universal-unique-identifiers (uuid-s), use these options to toggle between a warning and an error.</p> <p>The default analysis stops with an error if the issue happens. To convert to a warning, use -Eno -autosar -xmlReaderTooManyUuids. For conflicting UUID-s, the analysis stores the last element read and continues with a warning.</p> <p>The subsequent executions continue to use the warning mode. To revert back to an error, use -Eautosar -xmlReaderTooManyUuids.</p>

Options to control reading of C source code

Option	Description
<p><code>-include USER_RTE_TYPE_H</code></p>	<p>Define additional data types and macros that are not part of your ARXML specifications, but needed for analysis of the code implementation.</p> <p>Add the data type and macro definitions to a file <i>USER_RTE_TYPE_H</i>. These definitions are appended to a header file <i>Rte_Type.h</i> that is used in the analysis. The file that you provide must itself not be named <i>Rte_Type.h</i>.</p> <p>You can provide the file with data type and macro definitions only during project creation. For subsequent updates, you can change the contents of this file but not provide a new file. Also, this file must not be in the same folder as the Polyspace project and results.</p> <p>If you additionally define macros or undefine them using the options <code>-D</code> or <code>-U</code>, for definitions that conflict with the ones in <i>USER_RTE_TYPE_H</i>, the <code>-D</code> or <code>-U</code> specifications prevail.</p>
<p><code>-I INCLUDE_FOLDER</code></p>	<p>Specify folders containing header files. The analysis looks for <code>#include-d</code> files in this folder. The folder must be a subfolder of your source code folder.</p> <p>Repeat the option for multiple folders. The analysis looks for header files in these folders in the order in which you specify them.</p> <p>If you want to specify folders that are not in the source code folder, use the option:</p> <pre>-extra-project-options -I INCLUDE_FOLDER"</pre>
<p><code>-D DEFINE</code></p>	<p>Specify macros that the analysis must consider as defined.</p> <p>For instance, if you specify:</p> <pre>-D _WIN32</pre> <p>the preprocessor conditional <code>#ifdef _WIN32</code> succeeds and the corresponding branch is executed.</p>

Option	Description
-U <i>UNDEFINE</i>	Specify macros that the analysis must consider as undefined. For instance, if you specify: -U <code>_WIN32</code> the preprocessor conditional <code>#ifndef _WIN32</code> succeeds and the corresponding branch is executed.

Options to control Code Prover checks

Option	Description
<p>-extra-project-options POLYSPACE_OPTIONS</p>	<p>Specify additional options for the Code Prover analysis. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis.</p> <p>Use this method to specify analysis options that you use with the <code>polyspace-code-prover-server</code> command. See “Analysis Options”.</p> <p>Note that these options of <code>polyspace-code-prover</code> do not need to be specified:</p> <ul style="list-style-type: none"> • <code>-sources</code>: <code>polyspace-autosar</code> extracts the required source files. • <code>-I</code>: You specify include folders with the <code>-I</code> option of <code>polyspace-autosar</code>. • “Inputs and Stubbing” options such as <code>-data-range-specifications</code>: External data constraints in your ARXML files are extracted automatically with <code>polyspace-autosar</code>. You cannot specify constraints explicitly. • “Multitasking” options such as <code>-entry-points</code>: You cannot perform a multitasking analysis with <code>polyspace-autosar</code>. To detect data races, create a separate project for the entire application and explicitly add your source folders. Specify the ARXML files relevant for multitasking and run Bug Finder. For more information, see <code>ARXML files selection (-autosar-multitasking)</code>. • “Code Prover Verification” options associated with <code>main</code> generation: A <code>main</code> function is generated (in the file <code>psar_prove_main.c</code>) when you create a Polyspace project from an AUTOSAR description. The <code>main</code> function calls functions that implement runnable entities in the software components. The generated <code>main</code> is needed for the Code Prover analysis. You cannot change the properties of this <code>main</code> function. • Automatic Orange Tester options: You cannot use the Automatic Orange Tester when running Polyspace on code implementation of AUTOSAR software components.

Option	Description
-extra-options-file <i>OPTIONS_FILE</i>	<p>Specify additional options for the Code Prover analysis in an options file. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis.</p> <p>For instance, you can trace your build command to gather compiler options, macro definitions and paths to include folders, and provide this information in an options file for analysis of code implementation of AUTOSAR software components.</p> <ol style="list-style-type: none"> Trace your build command (for instance, make) with <code>polyspace-configure</code> and generate an options file for subsequent Code Prover analysis. Suppress inclusion of sources in the options file with the <code>-no-sources</code> option. <pre>polyspace-configure \ -output-options-file options.txt \ -no-sources make</pre> Run Code Prover on AUTOSAR code with <code>polyspace-autosar</code>. Provide your ARXML folder, source folders and other options. In addition, provide the earlier generated options file with the <code>-extra-options-file</code> option. <pre>polyspace-autosar ... \ -extra-options-file options.txt</pre> <p>See also “Run Polyspace on AUTOSAR Code Using Build Command” (Polyspace Code Prover).</p>
-show-prove <i>AUTOSAR_QUALIFIED_NAME</i>	<p>After analysis, open results for a specific software component whose internal behavior is specified by <i>AUTOSAR_QUALIFIED_NAME</i>.</p>

See Also

Topics

“Create Polyspace Analysis Configuration from AUTOSAR Specifications”

Introduced in R2018a

polyspace-code-prover-server Command

(DOS/UNIX) Run a Code Prover verification on a server from the DOS or UNIX command line

Syntax

```
polyspace-code-prover-server
```

```
polyspace-code-prover-server -sources sourceFiles [OPTIONS]
```

```
polyspace-code-prover-server -sources-list-file listOfSources [OPTIONS]
```

```
polyspace-code-prover-server -options-file optFile
```

```
polyspace-code-prover-server -h[elp]
```

Description

`polyspace-code-prover-server [OPTIONS]` runs a Code Prover verification on a server if your current folder contains a `sources` subfolder with source files (`.c` or `.cxx` files). The verification considers files in `sources` and all subfolders under `sources`. You can customize the verification with additional options.

`polyspace-code-prover-server -sources sourceFiles [OPTIONS]` runs a Code Prover verification on a server on the source file(s) `sourceFiles`. You can customize the verification with additional options.

`polyspace-code-prover-server -sources-list-file listOfSources [OPTIONS]` runs a Code Prover verification on a server on the source files listed in the text file `listOfSources`. You can customize the verification with additional options.

`polyspace-code-prover-server -options-file optFile` runs a Code Prover verification on a server with the options specified in the option file.

`polyspace-code-prover-server -h[elp]` lists a summary of possible analysis options.

Examples

Run Verification by Directly Specifying Options

Run a Code Prover verification by specifying analysis options in the command itself. This example uses source files from a demo Polyspace Code Prover example. To run this example, replace *polyspaceserverroot* with the path to your Polyspace Server installation, for example `C:\Program Files\Polyspace Server\R2019a`.

Run a verification on `numerical.c` and `programming.c`, checking for MISRA C:2012 mandatory rules and using GNU 4.7 compiler settings. This example command is split by `^` characters for readability. In practice, you can put all commands on one line.

```
polyspaceserverroot\polyspace\bin\polyspace-code-prover-server -lang C ^  
-sources polyspaceserverroot\polyspace\examples\cxx\Code_Prover_Example\sources\*.c,^
```

```
-I polyspaceserverroot\polyspace\examples\cxx\Code_Prover_Example\sources\^
-compiler generic -misra3 mandatory^
-author jlittle -prog myProject -results-dir C:\Polyspace_Workspace\Results\
```

After analysis, you can upload the results to the Polyspace Code Prover Access™ interface for review. See:

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

Run Verification with Options File

Run a verification by using an options file to specify your source files and analysis options. To run this example, replace *polyspaceserverroot* with the path to your Polyspace Server installation, for example `C:\Program Files\Polyspace Server\R2019a`.

Save this text to a text file called `myOptsFile.txt`.

```
# Polyspace analysis options
-I polyspaceserverroot\polyspace\examples\cxx\Code_Prover_Example\sources
-verif-version 1.0
-sources-list-file polyspaceserverroot\polyspace\examples\cxx\Code_Prover_Example\sources\*.c
-lang C
-target i386
-compiler generic
-dos
-do-not-generate-results-for all-headers
-misra3 mandatory-required
-entry-points proc1,proc2,server1,server2,tregulate
-critical-section-begin Begin_CS:Cs10
-critical-section-end End_CS:Cs10
-temporal-exclusions-file polyspaceserverroot\polyspace\examples\cxx\Code_Prover_Example\^
sources\temporal_exclusions.txt
-float-rounding-mode to-nearest
-scalar-overflows-checks signed
-scalar-overflows-behavior truncate-on-error
-uncalled-function-checks none
-check-subnormal allow
-02
-to Software Safety Analysis level 2
-context-sensitivity-auto
-path-sensitivity-delta 0
-author jlittle
-prog myProject
-results-dir C:\Polyspace_Workspace\Results\
```

Run the verification with the options specified in the text file.

```
polyspaceserverroot\polyspace\bin\polyspace-code-prover-server -options-file myOptsFile.txt
```

After analysis, you can upload the results to the Polyspace Code Prover Access interface for review. See:

- `polyspace-access`

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

Input Arguments

sourceFiles — Comma-separated names of C or C++ files to analyze

`-sources string`

Comma-separated C or C++ source file names, specified as `-sources` followed by a string. If the files are not in the current folder (`pwd`), `sourceFiles` must include a full or relative path. To avoid errors because of paths with spaces, add quotes " " around the path. For more information, see `-sources`.

If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The verification considers files in `sources` and all subfolders under `sources`.

Example: `-sources myFile.c, -sources C:\mySources\myFile1.c,C:\mySources\myFile2.c`

listOfSources — Text file listing names of C or C++ files to analyze

`-sources-list-file file`

Text file which lists the name of C or C++ files, specified as `-sources-list-file` followed by the file. If the files are not in the current folder (`pwd`), `listOfSources` must include a full or relative path. To avoid errors because of paths with spaces, add quotes " " around the path. For more information, see `-sources-list-file`.

Example: `-sources-list-file filename.txt, -sources-list-file "C:\ps_analysis\source_files.txt"`

[OPTIONS] — Analysis option and corresponding value

`option name`

Analysis options and their corresponding values, specified by the option name and if applicable value. For syntax specifications, see the individual analysis option reference pages.

Example: `-lang C-CPP, -target i386`

optFile — Text file listing analysis options and values

`-options-file filepath`

Text file listing analysis options and values, specified as `-options-file` followed by the file. For more information, see `-options-file`.

Example: `-options-file opts.txt, -options-file "C:\ps_analysis\options.txt"`

See Also

Topics

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

“Prepare Scripts for Polyspace Analysis”

“Analysis Options”

Introduced in R2019a

polyspace-configure

(DOS/UNIX) Create Polyspace project from your build system at the DOS or UNIX command line

Syntax

```
polyspace-configure buildCommand
```

```
polyspace-configure [OPTIONS] buildCommand
```

Description

`polyspace-configure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system.

`polyspace-configure [OPTIONS] buildCommand` traces your build system and uses `-option` value to modify the default operation of `polyspace-configure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

Examples

Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make targetName buildOptions` to build your source code.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W makefileName` option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspace-configure -prog myProject \  
make -B targetName buildOptions
```

Open the Polyspace project in the Polyspace user interface.

Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspace-configure -no-project make -B
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspace-configure -no-build -prog myProject \  
-include-sources "glob_pattern"
```

glob_pattern is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes. For more information on the supported syntax for glob patterns, see “polyspace-configure Source Files Selection Syntax”.

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rm -r polyspace_configure_cache polyspace_configure_built_trace
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use the command `make targetName buildOptions` to build your source code. In this example, you use `polyspace-configure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W makefileName` option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspace-configure -output-options-file\  
myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

Input Arguments

buildCommand — Command for building source code

build command

Build command specified exactly as you use to build your source code.

Example: `make -B, make -W makefileName`

[OPTIONS] — Options for changing default operation of `polyspace-configure`

single option starting with `-`, followed by argument | multiple space-separated option-argument pairs

Basic Options

Option	Argument	Description
-prog	Project name	Project name that appears in the Polyspace user interface. The default is polyspace. If you do not use the option -output-project, the -prog argument also sets the project name. Example: -prog myProject creates a project that has the name myProject in the user interface. If you do not use the option -output-project, the project name is also myProject.psrprj.
-author	Author name	Name of project author. Example: -author jsmith
-output-project	Path	Project file name and location for saving project. The default is the file polyspace.psrprj in the current folder. Example: -output-project ../myProjects/project1 creates a project project1.psrprj in the folder with the relative path ../myProjects/.
-output-options-file	File name	Option to create a Polyspace analysis options file. Use this file for command-line analysis using polyspace-code-prover-server.
-allow-build-error	None	Option to create a Polyspace project even if an error occurs in the build process. If an error occurs, the build trace log shows the following message: <pre>polyspace-configure (polyspaceConfigure) ERROR: build command command_name fail [status=status_value]</pre> <i>command_name</i> is the build command name that you use and <i>status_value</i> is the non-zero exit status or error level that indicates which error occurred in your build process.
-allow-overwrite	None	Option to overwrite a project with the same name, if it exists. By default, polyspace-configure (polyspaceConfigure) throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.
-silent (default) -verbose	None	Option to suppress or display additional messages from running polyspace-configure (polyspaceConfigure).
-help	None	Option to display the full list of polyspace-configure (polyspaceConfigure) commands

Option	Argument	Description
-debug	None	Option to store debug information for use by MathWorks® technical support. This option has been superseded by the option -easy-debug.
-easy-debug	Path	Option to store debug information for use by MathWorks technical support. After a polyspace-configure (polyspaceConfigure) run, the path provided contains a zipped file ending with pscfg-output.zip. If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also “Errors in Project Creation from Build Systems”.

Options to Create Multiple Modules

Option	Argument	Description
-module	None	Option to create a separate options file for each binary created in build system. You can only create separate options files for different binaries. You cannot create multiple modules in a Polyspace project (for running in the Polyspace user interface). Use this option only for build systems that use GNU® and Visual C++® compilers. See also “Modularize Polyspace Analysis by Using Build Command”.
-output-options-path	Path name	Location where generated options files are saved. Use this option together with the option -module. The options files are named after the binaries created in the build system.

Advanced Options

Option	Argument	Description
-compiler-config	Path and file name	<p>Location and name of compiler configuration file.</p> <p>The file must be in a specific format. For guidance, see the existing configuration files in <i>polyspaceroot</i>\polyspace\configure\compiler_configuration\. For information on the contents of the file, see “Create Polyspace Analysis Configuration from Build Command”.</p> <p>Example: -compiler-configuration myCompiler.xml</p>
-no-project	None	<p>Option to trace your build system without creating a Polyspace project and save the build trace information.</p> <p>Use this option to save your build trace information for a later run of polyspace-configure (polyspaceConfigure) with the -no-build option.</p>
-no-build	None	<p>Option to create a Polyspace project using previously saved build trace information.</p> <p>To use this option, you must have the build trace information saved from an earlier run of polyspace-configure (polyspaceConfigure) with the -no-project option.</p> <p>If you use this option, you do not need to specify the buildCommand argument.</p>

Option	Argument	Description
<code>-no-sources</code>	None	<p>Option to create a Polyspace options file that does not contain the source file specifications.</p> <p>Use this option when you intend to specify the source files by other means. For instance, you can use this option when:</p> <ul style="list-style-type: none">Running Polyspace on AUTOSAR-specific code. <p>You want to create an options file that traces your build command for the compiler options:</p> <pre>-output-options-file options.txt -no-sources</pre> <p>You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with <code>polyspace-autosar</code></p> <pre>-extra-options-file options.txt</pre> <p>See also “Create Polyspace Analysis Configuration from AUTOSAR Specifications”.</p> <ul style="list-style-type: none">Running Polyspace in Eclipse™. <p>Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only.</p>

Option	Argument	Description
-extra-project-options	Options to use for subsequent Polyspace analysis. For instance, "-stubbed-pointers-are-unsafe".	<p>Options that are used for subsequent Polyspace analysis.</p> <p>Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag <code>-extra-project-options</code> allows you to pass additional options.</p> <p>Specify multiple options in a space separated list, for instance <code>"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"</code>.</p> <p>Suppose you have to set the option <code>-stubbed-pointers-are-unsafe</code> for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project:</p> <pre>-extra-project-options "-stubbed-pointers-are-unsafe"</pre> <p>For the list of options available, see:</p> <ul style="list-style-type: none"> • • "Analysis Options" <p>If you are creating an options file instead of a Polyspace project from your build command, do not use this flag.</p>
-tmp-path	Path	Location of folder where temporary files are stored.
-build-trace	Path and file name	<p>Location and name of file where build information is stored. The default is <code>./polyspace_configure_build_trace.log</code>.</p> <p>Example: <code>-build-trace ../build_info/trace.log</code></p>
-include-sources -exclude-sources	Glob pattern	<p>Option to specify which source files <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) includes in, or excludes from, the generated project. You can combine both options together.</p> <p>A source file is included if the file path matches the glob pattern that you pass to <code>-include-sources</code>.</p> <p>A source file is excluded if the file path matches the glob pattern that you pass to <code>-exclude-sources</code>.</p>

Option	Argument	Description
-print-included-sources -print-excluded-sources	None	Option to print the list of source files that <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line. Use this option to troubleshoot the glob patterns that you pass to <code>-include-sources</code> or <code>-exclude-sources</code> . You can see which files match the pattern that you pass to <code>-include-sources</code> or <code>-exclude-sources</code> .

Cache Control Options

These options are primarily useful for debugging. Use the options if `polyspace-configure` (`polyspaceConfigure`) fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option `-easy-debug` provides an easier way to provide debug information. See “Contact Technical Support About Issues with Running Polyspace”.

Option	Argument	Description
-no-cache -cache-sources (default) -cache-all-text -cache-all-files	None	Option to perform one of the following: <ul style="list-style-type: none"> -no-cache: Not create a cache -cache-sources: Cache text files temporarily created during build for later use by <code>polyspace-configure</code> (<code>polyspaceConfigure</code>). -cache-all-text: Cache all text files including sources and headers. -cache-all-files: Cache all files including binaries. Typically, you cache temporary files created by your build command to debug issues in tracing the command.
-cache-path	Path	Location of folder where cache information is stored. Example: <code>-cache-path ../cache</code>
-keep-cache -no-keep-cache (default)	None	Option to preserve or clean up cache information after <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) completes execution. If <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) fails, you can provide this cache information to technical support for debugging purposes.

See Also

Topics

“Create Polyspace Analysis Configuration from Build Command”

“Modularize Polyspace Analysis by Using Build Command”

Introduced in R2013b

polyspace-report-generator

(DOS/UNIX) Generate reports for Polyspace analysis results stored locally or on Polyspace Access

Syntax

```
polyspace-report-generator -template <template> [OPTIONS]
polyspace-report-generator -generate-results-list-file [-results-dir <
FOLDER>] [-set-language-english]
polyspace-report-generator -generate-variable-access-file [-results-dir <
FOLDER>] [-set-language-english]

polyspace-report-generator -template <template> -host <HOSTNAME> -run-id <
RUN_ID> [ACCESS_OPTIONS] [OPTIONS]
polyspace-report-generator -generate-results-list-file -host <HOSTNAME> -run-
id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]
polyspace-report-generator -generate-variable-access-file -host <HOSTNAME> -
run-id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]
polyspace-report-generator -configure-keystore
```

Description

`polyspace-report-generator -template <template> [OPTIONS]` generates a report by using `TEMPLATE` for the local analysis results that you specify with `OPTIONS`.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -generate-results-list-file [-results-dir < FOLDER>] [-set-language-english]` exports the analysis results stored locally in `FOLDER` to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the user interface. For more information on the exported results list, see “View Exported Results” (Polyspace Code Prover).

By default, the results file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -generate-variable-access-file [-results-dir < FOLDER>] [-set-language-english]` exports the list of global variables in your code from the Code Prover analysis stored locally in `FOLDER` to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the user interface. For more information on the exported variables list, see “Global Variables” (Polyspace Code Prover Access).

By default, the variables file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -template <template> -host <HOSTNAME> -run-id < RUN_ID> [ACCESS_OPTIONS] [OPTIONS]` generates a report by using `TEMPLATE` for the analysis results run `RUN_ID` stored on Polyspace Access. `HOSTNAME` is the fully qualified host name of the machine that hosts Polyspace Access.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

```
polyspace-report-generator -generate-results-list-file -host <HOSTNAME> -run-id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]
```

exports the analysis results run `RUN_ID` stored on Polyspace Access to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the Polyspace Access web interface. `HOSTNAME` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported results list, see “Results List” (Polyspace Code Prover Access).

By default, the results file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

```
polyspace-report-generator -generate-variable-access-file -host <HOSTNAME> -run-id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]
```

exports the list of global variables in your code from the Code Prover analysis run `RUN_ID` stored on Polyspace Access to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the Polyspace Access web interface. `HOSTNAME` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

By default, the variables file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

```
polyspace-report-generator -configure-keystore
```

configures the report generator to communicate with Polyspace Access over HTTPS.

Run this one-time configuration step if Polyspace Access is configured to use the HTTPS protocol and you do not have a Polyspace Bug Finder™ desktop license, or you have a desktop license but you have not configured the desktop UI to communicate with Polyspace Access over HTTPS. Before running this command, generate a client keystore to store the SSL certificate that Polyspace Access uses for HTTPS. See “Generate a Client Keystore” (Polyspace Code Prover Access).

Examples

Generate PDF Reports for Analysis Results Stored Locally

You can generate multiple reports for analysis results that you store locally.

Create a variable `template_path` to store the path to the report templates and create a variable `report_templates` to store a comma-separated list of templates to use.

```
SET template_path="C:\Program Files\Polyspace\R2019a\toolbox\polyspace^\psrptgen\templates\
SET report_templates=%template_path%\Developer.rpt,^
%template_path%\CodingStandards.rpt
```

Generate the reports from the templates that you specified in `report_templates` for analysis results of Polyspace project `myProject`.

```
polyspace-report-generator -template %report_templates% ^
-results-dir C:\Polyspace_Workspace\myProject\Module_1\CP_Result ^
-format PDF
```

The command generates two PDF reports, `myProject_Developer.PDF` and `myProject_CodingStandards.PDF`. The reports are stored in `C:\Polyspace_Workspace\myProject\Module_1\CP_Result\Polyspace-Doc`. For more information on the content of the reports, see `Bug Finder` and `Code Prover report (-report-template)`.

Configure Report Generator with Client Keystore

If you configure Polyspace Access to use the HTTPS protocol, you must generate a client keystore where you store the SSL certificate that Polyspace Access uses, and configure `polyspace-report-generator` to use that keystore. See “Generate a Client Keystore” (Polyspace Code Prover Access). This one-time configuration enables the report generator to communicate with Polyspace Access over HTTPS.

To configure the report generator with a client keystore, use the `polyspace-report-generator -configure-keystore` command. Follow the prompts to provide the URL you use to log into Polyspace Access, the full path to the keystore file you generated, and the keystore password.

```
polyspace-report-generator -configure-keystore
Location: US, user name: jsmit, id: 62600@us-jsmith, print mode: false
Enter the Polyspace Access URL using form http[s]://<host>:<port> :
https://myAccessServer:9443
Enter full path to client keystore file :
C:\R2019b\ssl\client-cert.jks
Enter client keystore password :
```

The keystore has been configured

You must run the keystore configuration command again if:

- The Polyspace Access URL changes, for instance if you use a different port number.
- The path to the keystore file changes.
- The keystore password changes.

Generate Report and Variables List from Polyspace Access

Note To use the command-line for generating reports of results stored on Polyspace Access, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to generate a report and export the variables list for the results of a Code Prover analysis stored on the Polyspace Access database.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD LAMMEACDMKEFELKMNDCONEAPECEEKPL
Command Completed
```

Store your Polyspace Access login credentials in a variable `LOGIN`.

```
set LOGIN=-host jsmith ^
-encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

To specify project results on the Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run ID, use the `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer %LOGIN% -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on the command, see `polyspace-access`.

Generate a Developer report for results with run ID 16 from the Polyspace Access instance with host name `myAccessServer`. The URL of this instance of Polyspace Access is `https://myAccessServer:9443`.

```
SET template_path=^
"C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates"
```

```
polyspace-report-generator %LOGIN% ^
-template %template_path%\Developer.rpt ^
-host myAccessServer ^
-run-id 16 ^
-output-name myReport
```

The command creates report `myReport.docx` by using the template that you specify. The report is stored in folder `Polyspace-Doc` on the path from which you called the command.

Generate a tab-delimited text file that contains a list of global variables in your code for the specified analysis results.

```
polyspace-report-generator %LOGIN%^
-generate-variable-access-file ^
-host myAccessServer ^
-run-id 16
```

The list of global variables `Variable_View.txt` is stored in the same folder as the generated report. For more information on the exported variables list, see “Global Variables” (Polyspace Code Prover Access).

Input Arguments

template — path to report template file

string

Path to the report template that you use to generate an analysis report. To generate multiple reports, specify a comma-separated list of report template paths (do not put a space after the commas). The templates are available in `polyspaceroot\toolbox\polyspace\psrptgen\templates\` as `.rpt` files. Here, `polyspaceroot` is the Polyspace installation folder. For more information on the available templates, see `Bug Finder and Code Prover report (-report-template)`.

This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`.

Example: `C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates\Developer.rpt`

Example: `TEMPLATE_PATH\BugFinder.rpt,TEMPLATE_PATH\CodingStandards.rpt`

FOLDER — Analysis results folder path

string

Path to the folder containing analysis results for which you generate a report, or analysis results from which you export a list of results or global variables (Code Prover). To generate a report for multiple verifications, specify a comma-separated list of folder paths (do not put a space after the commas). If you do not specify a folder path, the command generates a report for analysis results in the current folder.

Example: `C:\Polyspace_Workspace\My_project\Module_1\results`

Example: `C:\Polyspace_Workspace\My_project\Module_2\results,C:\Polyspace_Workspace\My_project\Module_3\other_results`

HOSTNAME — Polyspace Access machine host name

string

Fully qualified host name of the machine that hosts the Polyspace Access **Gateway API** service. You must specify a host name to generate a report for results on the Polyspace Access database.

Example: `my-company-server`

RUN_ID — Polyspace Access run ID

integer

Run ID of the project findings for which you generate a report. Polyspace assigns a unique run ID to each analysis run that you upload to the Polyspace Access. To get the run ID of project findings, use the command `polyspace-access` with option `-list-project`.

Example: `4`

OPTIONS — Options for generated report

string

Option	Description
<code>-format HTML PDF WORD</code>	<p>File format of the report that you generate. By default, the command generates a WORD document.</p> <p>To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, <code>-format PDF,HTML</code>.</p> <p>This option is not compatible with <code>-generate-variable-access-file</code> and <code>-generate-results-list-file</code>.</p>

Option	Description
-output-name <i>outputName</i>	Name of the generated report or folder name if you generate multiple reports. The command stores <i>outputName</i> on the path from which you call the command. To store the generated files in a different folder, specify the full path of the folder, for instance -output-name C:\PathTo\OtherFolder.
-results-dir <i>FOLDER_1,...,FOLDER_N</i>	Path to the locally stored results folder. To generate reports for multiple analyses, specify a comma-separated list of folder path. (Do not put a space after the commas). For example: -results-dir folderPath1, folderPath2
-set-language-english	Generate the report in English. Use this option if your display language is set to another language.
-h	Display the help information.

ACCESS_OPTIONS — Options for Polyspace Access

string

Option	Description
-host <i>HOST_NAME</i>	Fully qualified host name of the machine that hosts the Polyspace Access Gateway API service. This option is mandatory when you generate reports for results stored on the Polyspace Access database.
-run-id <i>RUN_ID</i>	Run ID of the project. Polyspace assigns a unique run ID to each analysis run that you upload. To get the last run ID of a project, use the -list-project option of the polyspace-access command. For more information on the command, see polyspace-access. This option is mandatory when you generate reports for results stored on the Polyspace Access database.
-all-units	Specify this option to generate a report for all units from a unit by unit analysis. When you use this option, specify the run ID of only one unit with -run-id. The command includes the other units from the analysis in the report.
-port <i>portNumber</i>	Port number of the Polyspace Access instance. Default value is 9443.

Option	Description
-protocol <i>http</i> <i>https</i>	HTTP protocol used to connect to Polyspace Access. Default value is <code>https</code> .
-login <i>username</i> -encrypted-password <i>ENCRYPTED_PASSWD</i>	Credentials that you use to log into Polyspace Access. The argument of <code>-encrypted-password</code> is the output of the <code>polyspace-access -encrypt -password</code> command. For more information on the command, see <code>polyspace-access</code> .

See Also

Introduced in R2013b

polyspace-comments-import

(DOS/UNIX) Import review information from previous Polyspace analysis

Syntax

```
polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder
```

Description

`polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder` imports review information from a results file in `prevResultsFolder` to `currentResultsFolder`. The review information includes the severity, status and additional notes that you assign to a result. Besides importing the review information, the command also shows the number of results where review information could not be imported either because the result changed or the result already had new review information.

Examples

Import Review Information from Previous Polyspace Results

Run Bug Finder on a sample file and add some review information. Then, run Bug Finder a second time and import the information from the previous run.

Copy the file `numerical.c` from `polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources` to a writable folder. Open a command window and navigate to the folder (using `cd`). Run Bug Finder on the file and save results in the subfolder `Run_1`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_1/
```

Depending on the product installed, you can also run `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.

Open the results file in the `Run_1` subfolder:

```
polyspace Run_1/ps_results.psbf
```

Select a result. On the **Result Details** window, select a **Severity** and **Status** and add some notes. You will import this review information to results from a later analysis.

Run Bug Finder again, but save the results in a different subfolder `Run_2`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_2/
```

You can open the results file in `Run_2` and see that there is no review information.

Import the review information from the results file in the `Run_1` subfolder to the `Run_2` subfolder:

```
polyspace-comments-import -diff-rte Run_1/ Run_2/
```

Open the results file in the Run_2 subfolder:

```
polyspace Run_2/ps_results.psbf
```

You see the review information imported from the results file in the Run_1 subfolder.

Input Arguments

prevResultsFolder — Folder containing previous Polyspace results with review information

string

Path to a folder containing a Polyspace results file (.psbf file for Bug Finder results and .pscp file for Code Prover results). The results are presumably from an earlier Polyspace analysis and contain review information that will be imported to a later results file.

Example: "C:\Polyspace\Project_1_Run_25"

currentResultsFolder — Folder containing later Polyspace results

string

Path to a folder containing Polyspace results (.psbf file for Bug Finder results and .pscp file for Code Prover results). The results are presumably from a later Polyspace analysis and have no review information or review information for new results only. You want to import review information from an earlier Polyspace analysis to these results.

Example: "C:\Polyspace\Project_1_Run_26"

See Also

-import-comments

Topics

"Import Review Information from Previous Polyspace Analysis"

Introduced in R2013b

polyspaceAutosar

Run Polyspace Code Prover on code implementation of AUTOSAR software components using MATLAB scripts

Syntax

```
[status, msg] = polyspaceAutosar('-create-project',projectFolder,'-arxml-dir',arxmlFolder,'-sources-dir',codeFolder,options)
[status, msg] = polyspaceAutosar('-update-project',prevProjectFile,options)
[status, msg] = polyspaceAutosar('-update-and-clean-project',prevProjectFile,options)

[status, msg, out] = polyspaceAutosar( ___ )
```

Description

`[status, msg] = polyspaceAutosar('-create-project',projectFolder,'-arxml-dir',arxmlFolder,'-sources-dir',codeFolder,options)` checks the code implementation of AUTOSAR software components for run-time errors and violation of data constraints in the corresponding AUTOSAR XML specifications. The analysis parses the AUTOSAR XML specifications (.arxml files) in `arxmlFolder`, modularizes the code implementation (.c files) in `codeFolder` based on the specifications, and runs Code Prover on each module for the checks. The Code Prover results are stored in `projectFolder`. After analysis, you can open the project `psar_project.psprj` from `projectFolder` in the Polyspace user interface or the file `psar_project.xhtml` in a web browser. You can view the results for each software component individually.

You can use additional options for troubleshooting, for instance, to perform only certain parts of the update and track down an issue or to provide extra header files or define macros.

`[status, msg] = polyspaceAutosar('-update-project',prevProjectFile,options)` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update uses the XHTML file `prevProjectFile` from the previous analysis and reanalyzes only the code implementation of software components that changed since that analysis.

You can use additional options for troubleshooting.

`[status, msg] = polyspaceAutosar('-update-and-clean-project',prevProjectFile,options)` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update reanalyzes only the code implementation of software components that changed since the previous analysis. A clean update also removes information about software components that are out of date. For instance, if you use an additional option to force the update for specific software components and other SWC-s have also changed, a clean update removes those other SWC-s from the Polyspace project.

You can use additional options for troubleshooting.

`[status, msg, out] = polyspaceAutosar(___)` runs a Code Prover analysis using the same options as before. The output, instead of appearing in the MATLAB® Command Window, is redirected to a character vector `out`.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Examples

Run Code Prover on All Software Components

Suppose your ARXML files are in a folder `arxml` and your C source files in a folder `code` in the current folder.

Run Code Prover on all software components defined in your ARXML files. Store the results in a folder `polyspace-project` in a temporary folder.

The folder must not already exist. If previous results exist in that folder, you can update those results. An update only reanalyzes source files that changed since the previous run.

```
exampleDir = fullfile(polyspaceroot, 'help', ...
    'toolbox', 'codeprover', 'examples', 'polyspace_autosar');
arxmlDir = fullfile(exampleDir, 'arxml');
sourceDir = fullfile(exampleDir, 'code');

tempDir = tempdir;
projectDir = fullfile(tempDir, 'polyspace-project');
prevProjectFile = fullfile(projectDir, 'psar_project.xhtml');

% Update project file if it already exists, else create new project
projectDirAlreadyExists = isfolder(projectDir);

if projectDirAlreadyExists
    [status, msg] = polyspaceAutosar('-update-project', ...
        prevProjectFile);
else
    [status, msg] = polyspaceAutosar('-create-project', projectDir, ...
        '-arxml-dir', arxmlDir, ...
        '-sources-dir', sourceDir);
end
```

Input Arguments

projectFolder — Folder to store Polyspace results

character vector

Folder name, specified as a character vector. If the folder exists, it must be empty.

Example: 'C:\Polyspace_Projects\proj_swcl'

arxmlFolder — Folder containing ARXML files

character vector

Folder name, specified as a character vector.

UNC paths are not supported for the folder name.

Example: 'C:\arxml_swcl'

codeFolder — Folder containing C files

character vector

Folder name, specified as a character vector.

UNC paths are not supported for the folder name.

Example: 'C:\code_swcl'

prevProjectFile — Path to psar_project.xhtml

character vector

Path to the previously created project file `psar_project.xhtml`, specified as a character vector.

Example: 'C:\Polyspace_Projects\proj1\psar_project.xhtml'

options — Options to control project creation

character vector

Options to control creation of a Polyspace project and subsequent analysis. You primarily use the options for troubleshooting, for instance, to perform only certain parts of the update and narrow down an issue or to provide extra header files or define macros.

Specify each option as a character vector, followed by the option value as a separate character vector. For instance, you can specify an options file `opts.txt` by using the syntax `polyspaceAutosar(..., '-options-file', 'opts.txt')`.

General options

Option	Value	Description
'-verbose'		<p>Save additional information about the various phases of command execution (verbose mode). The file <code>psar_project.log</code> and other auxiliary files store this additional information.</p> <p>If an error occurs in command execution, the error message is stored in a separate file, irrespective of whether you enable verbose mode. Running in verbose mode only stores the various phases of execution. Use this information to see when an error was introduced.</p>

Option	Value	Description
'-options-file'	Options file name, for instance, 'opts.txt'.	<p>Use an options file to supplement or replace the command-line options. In the options file, specify each option on a separate line. Begin a line with # to indicate comments.</p> <p>An options file <code>opts.txt</code> can look like this:</p> <pre># Store Polyspace results -create-project polyspace # ARXML Folder -arxml-dir arxml # SOURCE Folder -sources-dir code</pre> <p>If an option that is directly specified with the <code>polyspaceAutosar</code> function conflicts with an option in the options file, the directly specified option is used.</p> <p>You typically use an options file to store and reuse options that are common to multiple projects.</p>

Options to control update of project

If you update a project, by default, the analysis results are updated for all AUTOSAR SWC behaviors with respect to any change in the ARXML files or C source code since the last analysis. Control the update by using these options.

Option	Value	Description
'-autosar-behavior'	Full qualified name of SWC behavior, for instance, 'pkg.component.bhv'.	<p>Check the implementation of software components whose internal behavior-s are specified. The default analysis considers all software components present in the ARXML specifications.</p> <p>To specify multiple software components, repeat the option. Alternatively, use regular expressions to specify a group of software components under the same package.</p> <p>For instance:</p> <ul style="list-style-type: none"> To specify the software component whose internal behavior has the fully qualified name <code>pkg.component.bhv</code>, use: <pre>polyspaceAutosar(..., '-autosar-behavior',... 'pkg.component.bhv')</pre> To specify the software components whose internal behavior-s have fully qualified names beginning with <code>pkg.component</code>, use: <pre>polyspaceAutosar(..., '-autosar-behavior',... 'pkg.component\..*')</pre> <p>The <code>\.</code> represents the package name separator <code>.</code> (dot) and the <code>.*</code> represents any number of characters.</p>
'-do-not-update-autosar-prove-environment'		<p>Do not read the ARXML specifications. Use ARXML specifications stored from the previous analysis.</p> <p>Use this option during project updates to compare the code against previous specifications. If you do not use this option, project updates read the entire ARXML specifications again.</p>

Option	Value	Description
'-do-not-update-extract-code'		<p>Do not read the C source code. Use source code stored from the previous analysis.</p> <p>Use this option during project updates to compare the previous source code against ARXML specifications. If you do not use this option, project updates consider all changes to the source code since the previous analysis.</p>
'-do-not-update-verification'		<p>Read the ARXML specifications and C code implementation only but do not run the Code Prover analysis.</p> <p>Use this option during project updates to investigate errors introduced in the ARXML specifications or compilation errors introduced in the source code. You can first fix these issues, and then run the Code Prover analysis.</p>

Options to control parsing of ARXML specifications

Option	Value	Description
'-autosar-datatype'	Full qualified name of data type, for instance, 'pkg.datatypes.type'	<p>Import definition of AUTOSAR data types specified. The default analysis imports only data types specified in the internal behavior of software components that you verify.</p> <p>To specify multiple data types, repeat the option. Alternatively, use regular expressions to specify all data types under the same package.</p> <p>For instance:</p> <ul style="list-style-type: none"> To specify a data type that has the fully qualified name <code>pkg.datatypes.type</code>, use: <pre>polyspaceAutosar(..., '-autosar-datatype',... 'pkg.datatypes.type')</pre> To specify data types that have fully qualified names beginning with <code>pkg.datatypes</code>, use: <pre>polyspaceAutosar(..., '-autosar-datatype',... 'pkg.datatypes\..*')</pre> <p>The <code>\.</code> represents the package name separator <code>.</code> (dot) and the <code>.*</code> represents any number of characters.</p> To force import of all data types, use: <pre>polyspaceAutosar(..., '-autosar-datatype',... '.*\..*')</pre>

Option	Value	Description
<p>'-Eautosar-xmlReaderSameUuidForDifferentElements'</p> <p>'-Eno-autosar-xmlReaderSameUuidForDifferentElements'</p>		<p>If multiple elements in the ARXML specifications have the same universal-unique-identifier (UUID), use these options to toggle between a warning and an error.</p> <p>The default analysis stops with an error if this issue happens. To convert to a warning, use '-Eno-autosar-xmlReaderSameUuidForDifferentElements'. For conflicting UUIDs, the analysis stores the last element read and continues with a warning.</p> <p>The subsequent executions continue to use the warning mode. To revert back to an error, use '-Eautosar-xmlReaderSameUuidForDifferentElements'.</p>
<p>'-Eautosar-xmlReaderTooManyUuids'</p> <p>'-Eno-autosar-xmlReaderTooManyUuids'</p>		<p>If the same element in the ARXML specifications has different universal-unique-identifiers (UUID), use these options to toggle between a warning and an error.</p> <p>The default analysis stops with an error if this issue happens. To convert to a warning, use '-Eno-autosar-xmlReaderTooManyUuids'. For conflicting UUIDs, the analysis stores the last element read and continues with a warning.</p> <p>The subsequent executions continue to use the warning mode. To revert back to an error, use '-Eautosar-xmlReaderTooManyUuids'.</p>

Options to control reading of C source code

Option	Value	Description
'-include'	File with data type and macro definitions.	<p>Define additional data types and macros that are not part of your ARXML specifications, but needed for analysis of the code implementation.</p> <p>Add the data type and macro definitions to a file. These definitions are appended to a header file <code>Rte_Type.h</code> that is used in the analysis. The file that you provide must itself not be named <code>Rte_Type.h</code>.</p> <p>You can provide the file with data type and macro definitions only during project creation. For subsequent updates, you can change the contents of this file but not provide a new file. Also, this file must not be in the same folder as the Polyspace project and results.</p> <p>If you additionally define macros or undefine them using the options <code>'-D'</code> or <code>'-U'</code>, for definitions that conflict with the ones in <code>USER_RTE_TYPE_H</code>, the <code>-D</code> or <code>-U</code> specifications prevail.</p>
'-I'	Folder containing header files.	<p>Specify folders containing header files. The analysis looks for <code>#include-d</code> files in this folder. The folder must be a subfolder of your source code folder.</p> <p>Repeat the option for multiple folders. The analysis looks for header files in these folders in the order in which you specify them.</p> <p>If you want to specify folders that are not in the source code folder, use the option:</p> <pre>polyspaceAutosar(..., '-extra-project-options',... '-I INCLUDE_FOLDER')</pre>

Option	Value	Description
'-D'	Name of macro, for instance, '_WIN32.	Specify macros that the analysis must consider as defined. For instance, if you specify:the preprocessor conditional <code>#ifdef _WIN32</code> succeeds and the corresponding branch is executed.
'-U'	Name of macro, for instance, '_WIN32.	Specify macros that the analysis must consider as undefined. For instance, if you specify:the preprocessor conditional <code>#ifndef _WIN32</code> succeeds and the corresponding branch is executed.

Options to control Code Prover checks

Option	Value	Description
'-extra-project-options'	Space-separated list of options.	Specify additional options for the Code Prover analysis. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis. Use this method to specify analysis options that are used in a non-AUTOSAR Code Prover analysis. See "Analysis Options". For instance, you might want to specify a compiler and target architecture. By default, compilation of projects created from AUTOSAR specifications use the gnu4.7 compiler and i386 architecture. To specify a visual11.0 compiler with x86_64 architecture, enter this option:See also Compiler (-compiler) and Target processor type (-target).

Option	Value	Description
'-extra-options-file'	File with Polyspace options.	<p>Specify additional options for the Code Prover analysis in an options file. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis.</p> <p>For instance, you can trace your build command to gather compiler options, macro definitions and paths to include folders, and provide this information in an options file for analysis of code implementation of AUTOSAR software components.</p> <ol style="list-style-type: none"> Trace your build command (for instance, make) with the <code>polyspaceConfigure</code> function and generate an options file for subsequent Code Prover analysis. Suppress inclusion of sources in the options file with the <code>-no-sources</code> option. <pre>polyspaceConfigure ... -output-options-file ... options.txt ... -no-sources make</pre> Run Code Prover on AUTOSAR code with <code>polyspace-autosar</code>. Provide your ARXML folder, source folders and other options. In addition, provide the earlier generated options file with the <code>-extra-options-file</code> option.
'-show-prove'	Full qualified name of SWC behavior, for instance, <code>'pkg.component.bhv'</code> .	After analysis, open results for a specific software component whose internal behavior is specified.

Output Arguments

status — Value indicating completion

0 | 1-10 (error values)

Boolean flag indicating whether the analysis ran to completion. If the analysis is completed, the return value is 0, otherwise it is a nonzero value.

If you see a nonzero value, check the second output argument of `polyspaceAutosar` for error messages.

You can also look for error messages in the file `psar_project.xhtml` in your project folder. You can use this XHTML file to determine which software components were analyzed.

msg — Analysis log

structure

Analysis log, specified as a structure with these fields:

Criticality — Type of message

'info' | 'warning' | 'error'

Type of message, returned as one of three character vectors:

- 'info': Information such as current stage of analysis.
- 'warning': Warnings that do not stop analysis but can cause errors later.
- 'error': Errors that can stop the entire analysis or analysis of specific software components.

To check for errors, use this type information. For instance, to check for errors in the structure `msg`, use this code:

```
% Convert to table for logical indexing
msgTable = struct2table(msg);

% Check which messages have the type 'error'
errorMatches = (strcmp(msgTable.Criticality, 'error'));

% Read the error messages to another table
errorMessage = msgTable(errorMatches, :);
```

Message — Content of message

character vector

Content of message, returned as a character vector.

Example: 'Start Extract user-implementation for Behavior
'pkg.tst002.swc001.bhv001'...'

out — Raw data in analysis log

character vector

Analysis log, returned as a character vector.

See Also

Topics

“Create Polyspace Analysis Configuration from AUTOSAR Specifications”

Introduced in R2018b

polyspaceCodeProverServer

Run Polyspace Code Prover verification from MATLAB

Note For easier scripting, run Polyspace® analysis using a `polyspace.Project` object.

Syntax

```
polyspaceCodeProverServer(optsObject)
```

```
polyspaceCodeProverServer('-help')
```

```
polyspaceCodeProverServer('-sources', sourceFiles)
```

```
polyspaceCodeProverServer('-sources', sourceFiles, Name, Value)
```

Description

`polyspaceCodeProverServer(optsObject)` runs a verification on the Polyspace options object in MATLAB.

`polyspaceCodeProverServer('-help')` displays all options that can be supplied to the `polyspaceCodeProverServer` command to run a Polyspace Code Prover verification.

`polyspaceCodeProverServer('-sources', sourceFiles)` runs a Polyspace Code Prover verification on the source files specified in `sourceFiles`.

`polyspaceCodeProverServer('-sources', sourceFiles, Name, Value)` runs a Polyspace Code Prover verification on the source files with additional options specified by one or more `Name, Value` pair arguments.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Examples

Run Polyspace Verification with Options Object

This example shows how to run a Polyspace verification in MATLAB using objects. For this example:

- Save a C source file, `source.c`, in the folder `C:\Polyspace_Sources`.
- Save an include file in the folder `C:\Polyspace_Includes`.

Create an options object and add the source file and include folder to the properties.

```
opts = polyspace.CodeProverOptions;  
opts.Sources = {'C:\Polyspace_Sources\source.c'};  
opts.EnvironmentSettings.IncludeFolders = {'C:\Polyspace_Includes'};  
opts.ResultsDir = 'C:\Polyspace_Results';
```

Run the verification and view the results.

```
polyspaceCodeProverServer(opts);
```

Polyspace runs on the file C:\Polyspace_Sources\source.c and stores the result in C:\Polyspace_Results.

Run Polyspace Verification from MATLAB with DOS/UNIX Options

This example shows how to run a Polyspace verification on a single source file. For this example:

- Save a C source file, `source.c`, in the folder C:\Polyspace_Sources.
- Save an include file in the folder C:\Polyspace_Includes.

Run the analysis and open the results.

```
polyspaceCodeProverServer('-sources','C:\Polyspace_Sources\source.c', ...
    '-I','C:\Polyspace_Includes', ...
    '-results-dir','C:\Polyspace_Results')
```

Run Polyspace Verification with Coding Rules Checking

This example shows two different ways to customize a verification in MATLAB. You can customize as many additional options as you want by changing properties in an options object or by using Name-Value pairs. You specify checking of MISRA C® 2012 coding rules, exclude headers from coding rule checking, and generate a main.

To create variables for source file path, include folder path, and results folder path that you can use for either analysis method.

```
sourceFileName = fullfile(polyspaceroot, 'polyspace','examples', 'cxx', ...
    'Code_Prover_Example','sources','example.c');
includeFileName = fullfile(polyspaceroot, 'polyspace','examples', 'cxx', ...
    'Code_Prover_Example','sources','include.h');
resFolder1 = fullfile('Polyspace_Results_1');
resFolder2 = fullfile('Polyspace_Results_2');
```

Verify coding rules with an options object.

```
opts = polyspace.CodeProverOptions('C');
opts.Sources = {sourceFileName};
opts.EnvironmentSettings.IncludeFolders = {includeFileName};
opts.ResultsDir = resFolder1;
opts.CodingStandards.MisraC3Subset = 'mandatory';
opts.CodingStandards.EnableMisraC3 = true;
opts.CodeProverVerification.EnableMain = true;
opts.InputsStubbing.DoNotGenerateResultsFor = 'all-headers';
polyspaceCodeProverServer(opts);
```

Verify coding rules with DOS/UNIX options.

```
polyspaceCodeProverServer('-sources',sourceFileName,...
    '-I',includeFileName, ...
    '-results-dir',resFolder2,...
```

```
'-misra3','mandatory',...  
'-do-not-generate-results-for','all-headers',...  
'-main-generator');
```

Input Arguments

optsObject — Polyspace options object name

object handle

Polyspace options object name, specified as the object handle.

To create an options object, use one of the Polyspace options classes: `polyspace.Options` or `polyspace.Project`.

Example: `opts`

sourceFiles — Comma-separated names of .c or .cpp files

character vector

Comma-separated source file names with extension `.c` or `.cpp`, specified as a single character vector.

If the files are not in the current folder, `sourceFiles` must include a full or relative path.

Example: `'myFile.c','C:\mySources\myFile1.c,C:\mySources\myFile2.c'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'-target','i386','-compiler','gnu4.6'` specifies that the source code is intended for i386 processors and contains non-ANSI C syntax for the GCC 4.6 compiler.

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

See Also

`polyspace.Project`

Topics

“Integrate Polyspace Server Products with MATLAB and Simulink”

Introduced in R2019a

polyspaceConfigure

Create Polyspace project from your build system at the MATLAB command line

Syntax

```
polyspaceConfigure buildCommand
```

```
polyspaceConfigure -option value buildCommand
```

Description

`polyspaceConfigure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system. You can run an analysis on a Polyspace project only in the user interface of the Polyspace desktop products.

`polyspaceConfigure -option value buildCommand` traces your build system and uses `-option value` to modify the default operation of `polyspaceConfigure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Examples

Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make targetName buildOptions` to build your source code. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W` *makefileName* option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspaceConfigure -prog myProject ...
                   make -B targetName buildOptions
```

Open the Polyspace project in the **Project Browser**.

Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspaceConfigure -no-project make -B;
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspaceConfigure -no-build -prog myProject ...  
-include-sources "glob_pattern";
```

glob_pattern is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes. For more information on the supported syntax for glob patterns, see “Create Polyspace Analysis Configuration from Build Command”.

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rmdir('polyspace_configure_cache', 's');  
delete polyspace_configure_built_trace;
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use a build command such as `make targetName buildOptions` to build your source code. In this example, you use `polyspaceConfigure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from the command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W makefileName` option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspaceConfigure -output-options-file ...  
myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspaceCodeProverServer -options-file myOptions
```

Input Arguments

buildCommand — Command for building source code

build command

Build command specified exactly as you use to build your source code.

Example: `make -B, make -W makefileName`

-option value – Options for changing default operation of polyspaceConfigure

single option starting with -, followed by argument | multiple space-separated option-argument pairs

Basic Options

Option	Argument	Description
-prog	Project name	Project name that appears in the Polyspace user interface. The default is <code>polyspace</code> . If you do not use the option <code>-output-project</code> , the <code>-prog</code> argument also sets the project name. Example: <code>-prog myProject</code> creates a project that has the name <code>myProject</code> in the user interface. If you do not use the option <code>-output-project</code> , the project name is also <code>myProject.psrprj</code> .
-author	Author name	Name of project author. Example: <code>-author jsmith</code>
-output-project	Path	Project file name and location for saving project. The default is the file <code>polyspace.psrprj</code> in the current folder. Example: <code>-output-project ../myProjects/project1</code> creates a project <code>project1.psrprj</code> in the folder with the relative path <code>../myProjects/</code> .
-output-options-file	File name	Option to create a Polyspace analysis options file. Use this file for command-line analysis using <code>polyspace-code-prover-server</code> .
-allow-build-error	None	Option to create a Polyspace project even if an error occurs in the build process. If an error occurs, the build trace log shows the following message: <pre>polyspace-configure (polyspaceConfigure) ERROR: build command command_name fail [status=status_value]</pre> <i>command_name</i> is the build command name that you use and <i>status_value</i> is the non-zero exit status or error level that indicates which error occurred in your build process.
-allow-overwrite	None	Option to overwrite a project with the same name, if it exists. By default, <code>polyspace-configure (polyspaceConfigure)</code> throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.

Option	Argument	Description
-silent (default) -verbose	None	Option to suppress or display additional messages from running <code>polyspace-configure</code> (<code>polyspaceConfigure</code>).
-help	None	Option to display the full list of <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) commands
-debug	None	Option to store debug information for use by MathWorks technical support. This option has been superseded by the option - <code>easy-debug</code> .
-easy-debug	Path	Option to store debug information for use by MathWorks technical support. After a <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) run, the path provided contains a zipped file ending with <code>pscfg-output.zip</code> . If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also “Errors in Project Creation from Build Systems”.

Options to Create Multiple Modules

Option	Argument	Description
-module	None	Option to create a separate options file for each binary created in build system. You can only create separate options files for different binaries. You cannot create multiple modules in a Polyspace project (for running in the Polyspace user interface). Use this option only for build systems that use GNU and Visual C++ compilers. See also “Modularize Polyspace Analysis by Using Build Command”.
-output-options-path	Path name	Location where generated options files are saved. Use this option together with the option - <code>module</code> . The options files are named after the binaries created in the build system.

Advanced Options

Option	Argument	Description
-compiler-config	Path and file name	<p>Location and name of compiler configuration file.</p> <p>The file must be in a specific format. For guidance, see the existing configuration files in <i>polyspaceroot</i>\polyspace\configure\compiler_configuration\. For information on the contents of the file, see "Create Polyspace Analysis Configuration from Build Command".</p> <p>Example: -compiler-configuration myCompiler.xml</p>
-no-project	None	<p>Option to trace your build system without creating a Polyspace project and save the build trace information.</p> <p>Use this option to save your build trace information for a later run of polyspace-configure (polyspaceConfigure) with the -no-build option.</p>
-no-build	None	<p>Option to create a Polyspace project using previously saved build trace information.</p> <p>To use this option, you must have the build trace information saved from an earlier run of polyspace-configure (polyspaceConfigure) with the -no-project option.</p> <p>If you use this option, you do not need to specify the buildCommand argument.</p>

Option	Argument	Description
<code>-no-sources</code>	None	<p>Option to create a Polyspace options file that does not contain the source file specifications.</p> <p>Use this option when you intend to specify the source files by other means. For instance, you can use this option when:</p> <ul style="list-style-type: none">• Running Polyspace on AUTOSAR-specific code. <p>You want to create an options file that traces your build command for the compiler options:</p> <pre>-output-options-file options.txt -no-sources</pre> <p>You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with <code>polyspace-autosar</code></p> <pre>-extra-options-file options.txt</pre> <p>See also “Create Polyspace Analysis Configuration from AUTOSAR Specifications”.</p> <ul style="list-style-type: none">• Running Polyspace in Eclipse. <p>Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only.</p>

Option	Argument	Description
-extra-project-options	Options to use for subsequent Polyspace analysis. For instance, "-stubbed-pointers-are-unsafe".	<p>Options that are used for subsequent Polyspace analysis.</p> <p>Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag <code>-extra-project-options</code> allows you to pass additional options.</p> <p>Specify multiple options in a space separated list, for instance <code>"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"</code>.</p> <p>Suppose you have to set the option <code>-stubbed-pointers-are-unsafe</code> for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project:</p> <pre data-bbox="813 909 1474 968">-extra-project-options "-stubbed-pointers-are-unsafe"</pre> <p>For the list of options available, see:</p> <ul data-bbox="813 1052 1474 1123" style="list-style-type: none"> • • "Analysis Options" <p>If you are creating an options file instead of a Polyspace project from your build command, do not use this flag.</p>
-tmp-path	Path	Location of folder where temporary files are stored.
-build-trace	Path and file name	<p>Location and name of file where build information is stored. The default is <code>./polyspace_configure_build_trace.log</code>.</p> <p>Example: <code>-build-trace ../build_info/trace.log</code></p>
-include-sources -exclude-sources	Glob pattern	<p>Option to specify which source files <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) includes in, or excludes from, the generated project. You can combine both options together.</p> <p>A source file is included if the file path matches the glob pattern that you pass to <code>-include-sources</code>.</p> <p>A source file is excluded if the file path matches the glob pattern that you pass to <code>-exclude-sources</code>.</p>

Option	Argument	Description
-print-included-sources -print-excluded-sources	None	Option to print the list of source files that <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line. Use this option to troubleshoot the glob patterns that you pass to <code>-include-sources</code> or <code>-exclude-sources</code> . You can see which files match the pattern that you pass to <code>-include-sources</code> or <code>-exclude-sources</code> .

Cache Control Options

These options are primarily useful for debugging. Use the options if `polyspace-configure` (`polyspaceConfigure`) fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option `-easy-debug` provides an easier way to provide debug information. See “Contact Technical Support About Issues with Running Polyspace”.

Option	Argument	Description
-no-cache -cache-sources (default) -cache-all-text -cache-all-files	None	Option to perform one of the following: <ul style="list-style-type: none"> -no-cache: Not create a cache -cache-sources: Cache text files temporarily created during build for later use by <code>polyspace-configure</code> (<code>polyspaceConfigure</code>). -cache-all-text: Cache all text files including sources and headers. -cache-all-files: Cache all files including binaries. Typically, you cache temporary files created by your build command to debug issues in tracing the command.
-cache-path	Path	Location of folder where cache information is stored. Example: <code>-cache-path ../cache</code>
-keep-cache -no-keep-cache (default)	None	Option to preserve or clean up cache information after <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) completes execution. If <code>polyspace-configure</code> (<code>polyspaceConfigure</code>) fails, you can provide this cache information to technical support for debugging purposes.

See Also

Topics

“Create Polyspace Analysis Configuration from Build Command”

“Requirements for Project Creation from Build Systems”

“Compiler Not Supported for Project Creation from Build Systems”

Introduced in R2013b

polyspaceroot

Get Polyspace installation folder

Syntax

```
polyspaceroot
```

Description

`polyspaceroot` returns the Polyspace installation folder.

Starting in R2019a, to run MATLAB scripts for Polyspace analysis, you install MATLAB and Polyspace in separate folders and link between them. After installation and linking, to access files in the Polyspace installation folder from MATLAB, use this function. See also “Integrate Polyspace Server Products with MATLAB and Simulink”.

Examples

Get Polyspace Installation Folder

To determine the Polyspace installation folder, use the `polyspaceroot` function.

```
polyspaceroot
```

```
C:\Program Files\Polyspace\R2019a
```

With the products, Polyspace Bug Finder Server or Polyspace Code Prover Server, the default installation folder in Windows® is:

```
C:\Program Files\Polyspace Server\R2019a
```

Run Polyspace on Sample Files in Polyspace Installation Folder

To access sample files in the Polyspace installation folder, use the `polyspaceroot` function to get the root of the installation folder. Append subfolders to the root folder path with the `fullfile` function.

Run Bug Finder on the file `numerical.c` in the subfolder `polyspace\examples\cxx\Bug_Finder_Example\sources` of the Polyspace installation folder.

```
proj = polyspace.Project
```

```
% Specify sources and includes
```

```
sourceFile = fullfile(polyspaceroot, 'polyspace', ...  
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');  
includeFolder = fullfile(polyspaceroot, 'polyspace', ...  
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
```

```
% Configure analysis
```

```
proj.Configuration.Sources = {sourceFile};
```

```
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';  
proj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};  
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
```

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

See Also

polyspace.Project

Topics

“Integrate Polyspace Server Products with MATLAB and Simulink”

Introduced in R2019a

polyspace_report

Generate reports from Polyspace analysis results

Syntax

```
polyspace_report('-template', template, '-results-dir', resultsFolder,  
options)  
polyspace_report('-generate-results-list-file', '-results-dir',  
resultsFolder, options)  
polyspace_report('-generate-variable-access-file', '-results-dir',  
resultsFolder, options)
```

Description

`polyspace_report('-template', template, '-results-dir', resultsFolder, options)` generates a report using a predefined template specified by `template`. By default, the report is named after the results file in the folder `resultsFolder` and saved in the Polyspace-Doc subfolder. You can change the default behavior using additional options.

`polyspace_report('-generate-results-list-file', '-results-dir', resultsFolder, options)` exports the list of Polyspace results to a tab-delimited text file.

`polyspace_report('-generate-variable-access-file', '-results-dir', resultsFolder, options)` exports the list of global variables to a tab-delimited text file.

Note

- Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.
 - You need MATLAB Report Generator™ to use this function.
-

Examples

Generate PDF Report from Results

Generate a PDF report from sample Polyspace Code Prover results.

```
template = fullfile(polyspaceroot, 'toolbox', 'polyspace', 'psrptgen', 'templates', ...  
    'Developer.rpt');  
resPath = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...  
    'Module_1', 'CP_Result');  
polyspace_report('-template', template, '-results-dir', resPath, '-format', 'PDF');
```

Input Arguments

template — Path to report template file

character vector

Path to report template file, specified as a character vector. To generate multiple reports, specify a comma-separated list of report template paths in the character vector (do not put a space after the commas). The templates are available in *polyspaceroot*\toolbox\polyspace\psrptgen\templates\ as .rpt files. Here, *polyspaceroot* is the Polyspace installation folder. For more information on the available templates, see Bug Finder and Code Prover report (-report-template).

Example: `fullfile(polyspaceroot,'toolbox','polyspace','psrptgen','templates','Developer.rpt');`

resultsFolder – Folder containing analysis results

character vector

Folder containing analysis results, specified as a character vector. The folder must contain a .psbf file containing Polyspace Bug Finder results or a .pscp file containing Polyspace Code Prover results.

To generate reports for multiple analyses, specify a comma-separated list of folder paths (do not put a space after the commas).

Example: `'C:\Polyspace_Workspace\My_project\Module_1\results'`

options – Options for generating report

character vector

Options to control report generation, for instance, output format and output name.

Specify each option as a character vector, followed by the option value as a separate character vector. For instance, you can specify the PDF format by using the syntax `polyspace_report(..., '-format','PDF')`.

Option	Value	Description
'-format'	'PDF', 'HTML' or 'WORD'	<p>File format of the report that you generate. By default, the command generates a Word document.</p> <p>To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, <code>polyspace_report(..., '-format','PDF,HTML')</code>.</p> <p>This option is not compatible with <code>-generate-variable-access-file</code> and <code>-generate-results-list-file</code>.</p>
'-set-language-english'		<p>Generate the report in English. Use this option if your display option is set to another language.</p>

Option	Value	Description
'-output-name'	Report name, for instance, PolyspaceReport.	Name of the generated report or folder name if you generate multiple reports. The full path to the report is created by appending the name to the current working folder. To store the reports on a different path, specify the full path as value for this option.

See Also

Introduced in R2013b

polyspace.Project

Run Polyspace analysis on C and C++ code and read results

Description

Run a Polyspace analysis on C and C++ source files by using this MATLAB object. To specify source files and customize analysis options, use the `Configuration` property. To run the analysis, use the `run` method. To read results after analysis, use the `Results` property.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Creation

`proj = polyspace.Project` creates an object that you can use to configure and run a Polyspace analysis, and then read the analysis results.

Properties

Configuration — Analysis options

`polyspace.Options` object

Options for running Polyspace analysis, implemented as a `polyspace.Options` object. The object has properties corresponding to the analysis options. For more information on those properties, see `polyspace.Project.Configuration` Properties.

You can retain the default options or change them in one of these ways:

- Set the source code language to 'C', 'CPP', or 'C-CPP' (default). Some analysis options might not be available depending on the language setting of the object.

```
proj=polyspace.Project;
proj.Configuration=polyspace.Options('C');
```

- Modify the properties directly.

```
proj = polyspace.Project;
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
```

- Obtain the options from another `polyspace.Project` object.

```
proj1 = polyspace.Project;
proj1.Configuration.TargetCompiler.Compiler = 'gnu4.9';
```

```
proj2 = proj1;
```

To use common analysis options across multiple projects, follow this approach. For instance, you want to reuse all options and change only the source files.

- Obtain the options from a project created in the user interface of the Polyspace desktop products (.psprj file).

```
proj = polyspace.Project;
projectLocation = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.psprj')
proj.Configuration = polyspace.loadProject(projectLocation);
```

To determine the optimal set of options, set your options in the user interface and then import them to a `polyspace.Project` object. In the user interface, you can access help from features such as the Compilation Assistant and get tooltip help on options.

- Obtain the options from a Simulink® model (applies only to Polyspace desktop products). Before obtaining the options, generate code from the model.

```
modelName = 'rtwdemo_roll';
load_system(modelName);

% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName, 'Solver', 'FixedStepDiscrete');
set_param(modelName, 'SupportContinuousTime', 'on');
set_param(modelName, 'LaunchReport', 'off');
set_param(modelName, 'InitFltsAndDblsToZero', 'on');

if exist(fullfile(pwd, 'rtwdemo_roll_ert_rtw'), 'dir') == 0
    rtwbuild(modelName);
end

% Obtain configuration from model
proj = polyspace.Project;
proj.Configuration = polyspace.ModelLinkOptions(modelName);
```

Use the options to analyze the code generated from the model.

Results — Analysis results

`polyspace.BugFinderResults` or `polyspace.CodeProverResults` object

Results of Polyspace analysis. When you create a `polyspace.Project` object, this property is initially empty. The property is populated only after you execute the `run` method of the object. Depending on the argument to the `run` method, `'bugFinder'` or `'codeProver'`, the property is implemented as a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object.

To read the results, use these methods of the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object:

- `getSummary`: Obtain a summarized format of the results into a MATLAB table.

```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

run(proj, 'bugFinder');

resTable = proj.Results.getSummary('defects');
```

For more information, see `getSummary` or `getSummary`.

- `getResults`: Obtain the full results or a more readable format into a MATLAB table.


```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

run(proj, 'bugFinder');

resTable = proj.Results.getResults('readable');

For more information, see getResult or getResult.
```

Object Functions

run Run a Polyspace analysis

Examples

Check for Bugs

Run a Polyspace Bug Finder analysis on the example file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

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

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

Prove Absence of Run-Time Errors

Run a Polyspace Code Prover analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a main function must be generated, if the function does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;
```

```
% Run analysis
cpStatus = run(proj, 'codeProver');

% Read results
cpSummary = proj.Results.getSummary('runtime');
```

Check for Bugs and MISRA C:2012 Violations

Run a Polyspace Bug Finder analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Enable checking of MISRA C:2012 rules. Check for the mandatory rules only.

```
proj = polyspace.Project
```

```
% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';
```

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

```
% Read results
defectsSummary = proj.Results.getSummary('defects');
misraSummary = proj.Results.getSummary('misraC2012');
```

See Also

Topics

“Integrate Polyspace Server Products with MATLAB and Simulink”

Introduced in R2017b

polyspace.Options class

Package: polyspace

Create object for running Polyspace analysis on handwritten code

Note For easier scripting, specify the Polyspace® analysis options using the `Configuration` property of a `polyspace.Project` object. Do not create a `polyspace.Options` object directly.

Description

Run a Polyspace analysis from MATLAB by using an options object. To specify source files and customize analysis options, change the object properties.

To analyze model-generated code (using the Polyspace desktop products), use `polyspace.ModelLinkOptions` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Construction

`opts = polyspace.Options` creates an object whose properties correspond to options for running a Polyspace analysis.

`proj = polyspace.Project` creates a `polyspace.Project` object. The object has a property `Configuration`, which is a `polyspace.Options` object.

`opts = polyspace.Options(lang)` creates a Polyspace options object with options that are applicable to the language `lang`.

`opts = polyspace.loadProject(projectFile)` creates a Polyspace options object from an existing Polyspace project `projectFile`. You set the options in your project in the Polyspace user interface and create the options object from that project for programmatically running the analysis.

Input Arguments

lang — Language of analysis

'C-CPP' (default) | 'C' | 'CPP'

The language of the analysis specified as 'C-CPP', 'C', or 'CPP'. This argument determines the object properties.

Data Types: char

projectFile — Name of .psprj file

character vector

Name of Polyspace project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path. To identify the current folder, use `pwd`. To change the current folder, use `cd`.

Example: 'C:\projects\myProject.psprj'

Properties

The object properties correspond to the analysis options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS/UNIX command-line name. For syntax details, see `polyspace.Project.Configuration` Properties.

Methods

<code>copyTo</code>	Copy common settings between Polyspace options objects
<code>generateProject</code>	Generate psprj project from options object
<code>toScript</code>	Add Polyspace options object definition to a script

Examples

Customize and Run Analysis

Create a Polyspace analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties. In case you do not have write access to your current folder, a temporary folder is being used for storing analysis results.

```
sources = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', ...
    'sources', 'numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Run a Bug Finder analysis. To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

```
results = polyspaceBugFinder(opts);
```

With the Polyspace Server products, you can use the functions `polyspaceBugFinderServer` or `polyspaceCodeProverServer`.

Open the results in the Polyspace user interface of the desktop products.

```
polyspaceBugFinder('-results-dir', opts.ResultsDir);
```

Run Polyspace by Generating a Project File

Create a Polyspace analysis options object and customize the properties. Then, run a Bug Finder analysis.

Create object and customize properties.

```
sources=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...  
    'sources','numerical.c');  
opts = polyspace.Options();  
opts.Prog = 'MyProject';  
opts.Sources = {sources};  
opts.TargetCompiler.Compiler = 'gnu4.7';  
opts.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = opts.generateProject(opts.Prog);  
polyspaceBugFinder(psprj);
```

You can also analyze the project from the command line. Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(psprj, '-nodesktop');  
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

Alternatives

If you are analyzing code generated from a model, use instead.

See Also

`polyspace.Project` | `polyspaceCodeProverServer`

Topics

“Integrate Polyspace Server Products with MATLAB and Simulink”

Introduced in R2017a

polyspace.CodingRulesOptions class

Package: polyspace

Create custom list of coding rules to check

Description

Create a custom list of coding rules to check in a Polyspace analysis.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Construction

`ruleList = polyspace.CodingRulesOptions(RuleSet)` creates the coding rules object `ruleList` for the `RuleSet` coding rule set. Set the active rules in the coding rules object.

Input Arguments

RuleSet — Standard coding rule set

`misraC` (default) | `misraC2012` | `misraAcAgc` | `misraCpp` | `jsf` | `certC` | `certCpp` | `iso17961` | `autosarCpp14`

Standard coding rule set specified as one of the coding rule acronyms.

Example: `'misraCpp'`

Data Types: `char`

Properties

For each coding rule set, an object is created with all supported rules divided into sections. By default, all rules are on. To turn off a rule, set the rule to `false`. For example:

```
misraRules = polyspace.CodingRulesOptions('misraC');  
misraRules.Section_20_Standard_libraries.rule_20_1 = false;
```

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Customize List of Coding Rules to Check

Customize the coding rules that are checked in a Polyspace analysis. Since all rules are enabled by default, you can create a custom subset by disabling some rules.

Create two objects: a `polyspace.CodingRulesOptions` object for setting coding rules and a `polyspace.Project` object for running the Polyspace analysis.

```
misraRules = polyspace.CodingRulesOptions('misraC2012');
proj = polyspace.Project;
```

Customize the coding rule list by turning off rules 2.1-2.7.

```
misraRules.Section_2_Unused_code.rule_2_1 = false;
misraRules.Section_2_Unused_code.rule_2_2 = false;
misraRules.Section_2_Unused_code.rule_2_3 = false;
misraRules.Section_2_Unused_code.rule_2_4 = false;
misraRules.Section_2_Unused_code.rule_2_5 = false;
misraRules.Section_2_Unused_code.rule_2_6 = false;
misraRules.Section_2_Unused_code.rule_2_7 = false;
```

Add the customized list of coding rules to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;
```

You have to enable checkers selection by file because the Polyspace run uses an XML file underneath to enable the coding rule checkers. The XML file is saved in a `.settings` subfolder of the results folder.

You can now use the `polyspace.Project` object to run the analysis. For instance, you can enter:

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
run(proj, 'bugfinder');
```

Create Coding Rules Object Using Rule Numbers to Enable

Suppose that you want to specify a subset of MISRA C: 2012 rules for the analysis. Instead of enumerating rules that you want disabled, you can specify the rules that you want to keep enabled. You can also specify the rule numbers only without the MISRA C: 2012 sections containing the rules.

Specify the rule numbers in a cell array to the `createRulesObject` function defined as follows.

```
function rulesObject = createRulesObject(rulesToEnable)

%% This function takes a cell array of MISRA C:2012 rules and returns
%% a polyspace.CodingRulesOptions object with the rules enabled.
%% Example input argument: {'2.7', '3.1'}

    rulesObject = polyspace.CodingRulesOptions('misraC2012');

    % Coding Standards documents have many sections. Loop over all
    % sections.
    ruleSections = properties(rulesObject);
    for i=1:length(ruleSections)
        sectionName = ruleSections{i};
        rulesInSection = properties(rulesObject.(sectionName));
```

```
% Loop over all rules in a section, enable or disable rule based
% on input
for j=1:length(rulesInSection)
    ruleNumberAsProperty = rulesInSection{j};
    ruleNumber = strrep(strrep(ruleNumberAsProperty, 'rule_', ''), '_', '.');
    if(any(strcmp(rulesToEnable, ruleNumber))
        rulesObject.(sectionName).(ruleNumberAsProperty)=1;
    else
        rulesObject.(sectionName).(ruleNumberAsProperty)=0;
    end
end
end
end
```

For instance, to enable rules 1.1 and 2.2, enter:

```
createRulesObject({'1.1', '2.2'})
```

See Also

`polyspace.Options` | `polyspace.Project`

Introduced in R2016b

polyspace.GenericTargetOptions class

Package: polyspace

Create a generic target configuration

Description

Create a custom target for a Polyspace analysis if your target processor does not match one of the predefined targets,.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Construction

`genericTarget = polyspace.GenericTargetOptions` creates a generic target that you can customize. To specify the sizes and alignment of data types, change the properties of the object. For instance:

```
target = polyspace.GenericTargetOptions;  
target.CharNumBits = 16;
```

Properties

For more details about any of the properties below, see `Generic target options`.

Alignment — Largest alignment of struct or array objects

32 (default) | 16 | 8

Largest alignment of struct or array objects, specified as 32, 16, or 8. Comparable with the DOS/UNIX command-line option `-align`.

Example: `target.Alignment = 8`

CharNumBits — Define the number of bits for a char

8 (default) | 16

Define the number of bits for a char, specified as 8 or 16. Comparable with the DOS/UNIX command-line option `-char-is-16bits`.

Example: `target.CharNumBits = 16`

DoubleNumBits — Define the number of bits for a double

32 (default) | 64

Define the number of bits for a double, specified as 32 or 64. Comparable with the DOS/UNIX command-line option `-double-is-64bits`.

Example: `target.DoubleNumBits = 64`

Endianness — Endianness of target architecture`little (default) | big`

Endianness of target architecture, specified as `little` or `big`. Comparable with the DOS/UNIX command-line options `-little-endian` or `-big-endian`.

Example: `target.Endianness = 'big'`

IntNumBits — Define the number of bits for an int`16 (default) | 32`

Define the number of bits for an `int`, specified as 16 or 32. Comparable with the DOS/UNIX command-line option `-int-is-32bits`.

Example: `target.IntNumBits = 32`

LongLongNumBits — Define the number of bits for a long long`32 (default) | 64`

Define the number of bits for a `long long`, specified as 32 or 64. Comparable with the DOS/UNIX command-line option `-long-long-is-64bits`.

Example: `target.LongNumBits = 64`

LongNumBits — Define the number of bits for a long`32 (default)`

Define the number of bits for a `long`, specified as 32. Comparable with the DOS/UNIX command-line option `-long-is-32bits`.

Example: `target.LongNumBits = 32`

PointerNumBits — Define the number of bits for a pointer`16 (default) | 24 | 32`

Define the number of bits for a pointer, specified as 16, 24, or 32. Comparable with the DOS/UNIX command-line options `-pointer-is-24bits` and `-pointer-is-32bits`.

Example: `target.PointerNumBits = 32`

ShortNumBits — Define the number of bits for a short`16 (default) | 8`

Define the number of bits for an `int`, specified as 16 or 8. Comparable with the DOS/UNIX command-line option `-short-is-8bits`.

Example: `target.ShortNumBits = 8`

SignOfChar — Default sign of plain char`signed (default) | unsigned`

Default sign of plain char, specified as `signed` or `unsigned`. Comparable with the DOS/UNIX command-line option `-default-sign-of-char`.

Example: `target.SignOfChar = 'unsigned'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Customize Generic Target Settings

Use a custom target for the Polyspace analysis.

Create two objects: a `polyspace.GenericTargetOptions` object for creating a custom target and a `polyspace.Project` object for running the Polyspace analysis.

```
target = polyspace.GenericTargetOptions;  
proj = polyspace.Project;
```

Customize the generic target.

```
target.Endianness = 'big';  
target.LongLongNumBits = 64;  
target.ShortNumBits = 8;
```

Add the custom target to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.TargetCompiler.Target = target;
```

You can now use the `polyspace.Project` object to run the analysis.

Generic target options | `polyspace.CodingRulesOptions` | `polyspace.Options` | `polyspace.Project`

Introduced in R2016b

polyspace.CodeProverResults class

Package: polyspace

Read Polyspace Code Prover results from MATLAB

Description

Read Polyspace Code Prover analysis results to MATLAB tables by using this object.

You can obtain a high-level overview or read each individual result, for example, each instance of a run-time check.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB and Simulink”.

Construction

`resObj = polyspace.CodeProverResults(resultsFolder)` creates an object for reading a specific set of Code Prover results into MATLAB tables. Use the object methods to read the results.

`proj = polyspace.Project` creates a `polyspace.Project` object. The object has a property `Results`. If you run a Code Prover analysis, this property is a `polyspace.CodeProverResults` object.

Input Arguments

resultsFolder — Name of result folder

character vector

Name of result folder, specified as a character vector. The folder must contain the results file with extension `.pscp`. Even if the results file resides in a *subfolder* of the specified folder, it cannot be accessed.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

Methods

<code>getSummary</code>	View number of run-time checks organized by color and file
<code>getResults</code>	Read Code Prover results into MATLAB table
<code>variableAccess</code>	View global variables along with read/write operations in C/C++ code

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...
'Module_1', 'CP_Result');
userResPath = tempname;
copyfile(resPath, userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);
resTable = getResults (resObj);
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a main function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project
```

```
% Configure analysis
```

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...
'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;
```

```
% Run analysis
```

```
cpStatus = run(proj, 'codeProver');
```

```
% Read results
```

```
cpSummary = proj.Results.getResults('readable');
```

Alternatives

To read Bug Finder results from MATLAB, use the class `polyspace.BugFinderResults`.

Introduced in R2017a

polyspace.Project.Configuration Properties

Customize Polyspace analysis of handwritten code with options object properties

Description

To customize your Polyspace analysis, use these `polyspace.Options` or `polyspace.Project.Configuration` properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis option reference pages.
- How to create and use the object, see `polyspace.Options` or `polyspace.Project`.

The same properties are also available with the deprecated classes `polyspace.BugFinderOptions` and `polyspace.CodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

Note Some options might not be available depending on the language setting of the object. You can set the source code language (Language) to 'C', 'CPP' or 'C-CPP' during object creation, but cannot change it later.

Properties

Advanced

Additional — Additional flags for analysis

character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

PostAnalysisCommand — Command or script software should execute after analysis finishes

character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script` to apply after the end of the code verification (`-post-analysis-command`).

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

AutomaticOrangeTester — Run the Automatic Orange Tester

false (default) | true

This property affects Code Prover analysis only.

Run the Automatic Orange Tester after verification, specified as true or false.

For more information, see Automatic Orange Tester (-automatic-orange-tester).

Example: `opts.Advanced.AutomaticOrangeTester = true`**AutomaticOrangeTesterLoopMaxIteration — Number of loop iterations after which Automatic Orange Tester considers infinite loop**

1000 (default) | positive integer

This property affects Code Prover analysis only.

Number of loop iterations after which Automatic Orange Tester considers the test an infinite loop, specified as a positive integer, maximum of 1000.

For more information, see Maximum loop iterations (-automatic-orange-tester-loop-max-iteration).

Example: `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration = 500`**AutomaticOrangeTesterTestsNumber — Number of tests that Automatic Orange Tester must run**

500 (default) | positive integer

This property affects Code Prover analysis only.

Number of tests that Automatic Orange Tester must run, specified as a positive integer, maximum of 100,000.

For more information, see Number of automatic tests (-automatic-orange-tester-tests-number).

Example: `opts.Advanced.AutomaticOrangeTesterTestsNumber = 1000`**AutomaticOrangeTesterTimeout — Time in seconds allowed for a single test in Automatic Orange Tester**

5 (default) | positive integer

This property affects Code Prover analysis only.

Time in seconds allowed for a single test in Automatic Orange Tester, specified as a positive integer, maximum of 60.

For more information, see Maximum test time (-automatic-orange-tester-timeout).

Example: `opts.Advanced.AutomaticOrangeTesterTimeout = 10`**BugFinderAnalysis (Affects Bug Finder Only)****CheckersList — List of custom checkers to activate**

polyspace.DefectsOptions object | cell array of defect acronyms

This property affects Bug Finder analysis only.

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptions`.

```
Example: defects = polyspace.DefectsOptions;  
opts.BugFinderAnalysis.CheckersList = defects
```

```
Example: opts.BugFinderAnalysis.CheckersList =  
{'INT_ZERO_DIV', 'FLOAT_ZERO_DIV'}
```

CheckersPreset — Subset of Bug Finder defects

'default' (default) | 'all' | 'CWE' | 'custom'

This property affects Bug Finder analysis only.

Preset checker list, specified as a character vector of one of the preset options: 'default', 'all', 'CWE', or 'custom'. To use 'custom', specify a value for the property `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

```
Example: opts.BugFinderAnalysis.CheckersPreset = 'all'
```

ChecksUsingSystemInputValues — Activate stricter checks for system inputs

false (default) | true

This property affects Bug Finder analysis only.

Activate stricter checks that consider all possible value for:

- Global variables.
- Reads of volatile variables.
- Returns of stubbed functions.
- Inputs to functions specified with **SystemInputsFrom**.

The analysis considers all possible values for a subset of **Numerical** and **Static memory** defects.

This property is equivalent to the **Run stricter checks considering all values of system inputs** check box in the Polyspace interface.

For more information, see `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

```
Example: opts.BugFinderAnalysis.ChecksUsingSystemInputValues = true
```

EnableCheckers — Activate defect checking

true (default) | false

This property affects Bug Finder analysis only.

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

Example: `opts.BugFinderAnalysis.EnableCheckers = false`

SystemInputsFrom — List of functions for which you run stricter checks

'auto' (default) | 'uncalled' | 'all' | 'custom'

This property affects Bug Finder analysis only.

Functions for which you want to run stricter checks that consider all possible values of the function inputs. Specify the list of functions as 'auto', 'uncalled', 'all', or as a character array beginning with `custom=` followed by a comma-separated list of function names.

To enable this option, set `BugFinderAnalysis.ChecksUsingSystemInputValues = true`.

For more information, see `Consider inputs to these functions (-system-inputs-from)`

Example: `opts.BugFinderAnalysis.SystemInputsFrom = 'custom=foo,bar'`

ChecksAssumption (Affects Code Prover Only)

AllowNegativeOperandInShift — Allow left shift operations on a negative number

false (default) | true

This property affects Code Prover analysis only.

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

Example: `opts.ChecksAssumption.AllowNegativeOperandInShift = true`

AllowNonFiniteFloats — Incorporate infinities and/or NaNs

false (default) | true

This property affects Code Prover analysis only.

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see `Consider non finite floats (-allow-non-finite-floats)`.

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

AllowPtrArithOnStruct — Allow arithmetic on pointer to a structure field so that it points to another field

false (default) | true

This property affects Code Prover analysis only.

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

CheckInfinite — Detect floating-point operations that result in infinities

'allow' (default) | 'warn-first' | 'forbid'

This property affects Code Prover analysis only.

Detect floating-point operations that result in infinities.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `Infinities` (`-check-infinite`).

Example: `opts.ChecksAssumption.CheckInfinite = 'forbid'`

CheckNan — Detect floating-point operations that result in NaN-s

'allow' (default) | 'warn-first' | 'forbid'

This property affects Code Prover analysis only.

Detect floating-point operations that result in NaN-s.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `NaNs` (`-check-nan`).

Example: `opts.ChecksAssumption.CheckNan = 'forbid'`

CheckSubnormal — Detect operations that result in subnormal floating point values

'allow' (default) | 'warn-first' | 'warn-all' | 'forbid'

This property affects Code Prover analysis only.

Detect operations that result in subnormal floating point values.

For more information, see `Subnormal detection mode` (`-check-subnormal`).

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

DetectPointerEscape — Find cases where a function returns a pointer to one of its local variables

false (default) | true

This property affects Code Prover analysis only.

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope` (`-detect-pointer-escape`).

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

DisableInitializationChecks — Disable checks for noninitialized variables and pointers

false (default) | true

This property affects Code Prover analysis only.

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

PermissiveFunctionPointer — Allow type mismatch between function pointers and the functions they point to

false (default) | true

This property affects Code Prover analysis only.

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

SignedIntegerOverflows — Behavior of signed integer overflows

'forbid' (default) | 'allow' | 'warn-with-wrap-around'

This property affects Code Prover analysis only.

Enable the check for signed integer overflows and the assumptions to make following an overflow specified as 'forbid', 'allow', or 'warn-with-wrap-around'.

For more information, see `Overflow mode for signed integer (-signed-integer-overflows)`.

Example: `opts.ChecksAssumption.SignedIntegerOverflows = 'warn-with-wrap-around'`

SizeInBytes — Allow a pointer with insufficient memory buffer to point to a structure

false (default) | true

This property affects Code Prover analysis only.

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

UncalledFunctionCheck — Detect functions that are not called directly or indirectly from main or another entry-point function

'none' (default) | 'never-called' | 'called-from-unreachable' | 'all'

This property affects Code Prover analysis only.

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as none, never-called, called-from-unreachable, or all.

For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

UnsignedIntegerOverflows — Behavior of unsigned integer overflows

'allow' (default) | 'forbid' | 'warn-with-wrap-around'

This property affects Code Prover analysis only.

Enable the check for unsigned integer overflows and the assumptions to make following an overflow, specified as 'forbid', 'allow', or 'warn-with-wrap-around'.

For more information, see `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

Example: `opts.ChecksAssumption.UnsignedIntegerOverflows = 'allow'`

CodeProverVerification (Affects Code Prover only)**ClassAnalyzer — Classes that you want to verify**

'all' (default) | 'none' | 'custom=*class1*[,*class2*,...]'

This property affects Code Prover analysis only.

Classes that you want to verify, specified as 'all', 'none', or as a character array beginning with `custom=` followed by a comma-separated list of class names.

For more information, see `Class (-class-analyzer)`.

Example: `opts.CodeProverVerification.ClassAnalyzer = 'custom=myClass1,myClass2'`

ClassAnalyzerCalls — Class methods that you want to verify

'unused' (default) | 'all' | 'all-public' | 'inherited-all' | 'inherited-all-public' | 'unused-public' | 'inherited-unused' | 'inherited-unused-public' | 'custom=*method1*[,*method2*,...]'

This property affects Code Prover analysis only.

Class methods that you want to verify, specified as one of the predefined sets or as a character array beginning with `custom=` followed by a comma-separated list of method names.

For more information, see `Functions to call within the specified classes (-class-analyzer-calls)`.

Example: `opts.CodeProverVerification.ClassAnalyzerCalls = 'unused-public'`

ClassOnly — Analyze only class methods

false (default) | true

This property affects Code Prover analysis only.

Analyze only class methods, specified as true or false.

For more information, see `Analyze class contents only (-class-only)`.

Example: `opts.CodeProverVerification.ClassOnly = true`

EnableMain — Use main function provided in application

false (default) | true

This property affects Code Prover analysis only.

Use `main` function provided in application, specified as `true` or `false`. If you set this property to `false`, the analysis generates a `main` function, if it is not present in the source files.

For more information, see `Verify whole application`.

Example: `opts.CodeProverVerification.EnableMain = true`

FunctionsCalledBeforeMain — Functions that you want the generated main to call ahead of other functions

cell array of function names

This property affects Code Prover analysis only.

Functions that you want the generated main to call ahead of other functions, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-main)`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeMain = {'func1', 'func2'}`

Main — Use a Microsoft Visual C++ extensions of main

'_tmain' (default) | 'wmain' | '_tWinMain' | 'wWinMain' | 'WinMain' | 'DllMain'

This property applies to a Code Prover analysis only .

Use a Microsoft Visual C++ extension of `main`, specified as one of the predefined main extensions.

For more information, see `Main entry point (-main)`.

Example: `opts.CodeProverVerification.Main = 'wmain'`

MainGenerator — Generate a main function if it is not present in source files

true (default) | false

This property applies to a Code Prover analysis only .

Generate a `main` function if it is not present in source files, specified as `true` or `false`.

For more information, see `Verify module or library (-main-generator)`.

Example: `opts.CodeProverVerification.MainGenerator = false`

MainGeneratorCalls — Functions that you want the generated main to call after the initialization functions

'unused' (default) | 'none' | 'all' | 'custom=function1[,function2,...]'

This property applies to a Code Prover analysis only .

Functions that you want the generated main to call after the initialization functions, specified as `'unused'`, `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

For more information, see `Functions to call (-main-generator-calls)`.

Example: `opts.CodeProverVerification.MainGeneratorCalls = 'all'`

MainGeneratorWriteVariables – Global variables that you want the generated main to initialize

'uninit' (C++ default) | 'public' (C default) | 'none' | 'all' |
'custom=variable1[,variable2,...]'

This property applies to a Code Prover analysis only .

Global variables that you want the generated main to initialize, specified as one of the predefined sets, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Variables to initialize (-main-generator-writes-variables)`.

Example: `opts.CodeProverVerification.MainGeneratorWriteVariables = 'all'`

NoConstructorsInitCheck – Do not check if class constructor initializes class members

false (default) | true

This property applies to a Code Prover analysis only .

Do not check if class constructor initializes class members, specified as true or false.

For more information, see `Skip member initialization check (-no-constructors-init-check)`.

Example: `opts.CodeProverVerification.NoConstructorsInitCheck = true`

UnitByUnit – Verify each source file independently of other source files

false (default) | true

This property affects Code Prover analysis only.

Verify each source file independently of other source files, specified as true or false.

For more information, see `Verify files independently (-unit-by-unit)`.

Example: `opts.CodeProverVerification.UnitByUnit = true`

UnitByUnitCommonSource – Files that you want to include with each source file during a file-by-file verification

cell array of file paths

This property affects Code Prover analysis only.

Files that you want to include with each source file during a file-by-file verification, specified as a cell array of file paths.

For more information, see `Common source files (-unit-by-unit-common-source)`.

Example: `opts.CodeProverVerification.UnitByUnitCommonSource = {'/inc/file1.h', '/inc/file2.h'}`

CodingRulesCodeMetrics

AcAgcSubset — Subset of MISRA AC AGC rules to check

'OBL-rules' (default) | 'OBL-REC-rules' | 'single-unit-rules' | 'system-decidable-rules' | 'all-rules' | 'SQ0-subset1' | 'SQ0-subset2' | polyspace.CodingRulesOptions object | 'from-file'

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC (-misra-ac-agc)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use 'from-file' for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

Example: `opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'`

Data Types: char

AllowedPragmas — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied

cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas (-allowed-pragmas)`.

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01','pragma_02'}`

Data Types: cell

AutosarCpp14 — Set of AUTOSAR C++ 14 rules to check

'all' (default) | 'required' | 'automated' | polyspace.CodingRulesOptions object | 'from-file'

This property affects Bug Finder only.

Set of AUTOSAR C++ 14 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use 'from-file' for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check AUTOSAR C++ 14 rules, also set `EnableAutosarCpp14` to true.

Example: `opts.CodingRulesCodeMetrics.AutosarCpp14 = 'all'`

Data Types: char

BooleanTypes — Data types the coding rule checker must treat as effectively Boolean

cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t','boolean2_t'}`

Data Types: cell

CertC — Set of CERT® C rules and recommendations to check

'all' (default) | 'publish-2016' | 'all-rules' | polyspace.CodingRulesOptions object | 'from-file'

This property affects Bug Finder only.

Set of CERT C rules and recommendations to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C security checks (-cert-c)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `from-file` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C rules and recommendations, also set `EnableCertC` to true.

Example: `opts.CodingRulesCodeMetrics.CertC = 'all'`

Data Types: char

CertCpp — Set of CERT C++ rules to check

'all' (default) | polyspace.CodingRulesOptions object | 'from-file'

This property affects Bug Finder only.

Set of CERT C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C++ security checks (-cert-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use 'from-file' for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCertCpp` to true.

Example: `opts.CodingRulesCodeMetrics.CertCpp = 'all'`

Data Types: char

CheckersSelectionByFile — File that defines custom set of coding standard checkers

full file path of .xml file

File where you define a custom set of coding standards checkers to check, specified as a .xml file. You can, in the same file, define a custom set of checkers for each of the coding standards that Polyspace supports. To create a file that defines a custom selection of coding standard checkers, in the Polyspace interface, select a coding standard on the **Coding Standards & Code Metrics** node of the **Configuration** pane and click **Edit**.

For more information, see `Set checkers by file (-checkers-selection-file)`.

Example: `opts.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\ps_settings\coding_rules\custom_rules.xml'`

Data Types: char

CodeMetrics — Activate code metric calculations

false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

If you assign a coding rules options object to this property, an XML file gets created automatically with the rules specified.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

EnableAcAgc — Check MISRA AC AGC rules

false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

EnableAutosarCpp14 — Check AUTOSAR C++ 14 rules

false (default) | true

This property affects Bug Finder only.

Check AUTOSAR C++ 14 rules, specified as true or false. To customize which rules are checked, use `AutosarCpp14`.

For more information about the AUTOSAR C++ 14 checker, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.

Example: `opts.CodingRulesCodeMetrics.EnableAutosarCpp14 = true;`

EnableCertC — check CERT C rules and recommendations

false (default) | true

This property affects Bug Finder only.

Check CERT C rules and recommendations, specified as true or false. To customize which rules are checked, use `CertC`.

For more information about the CERT C checker, see `Check CERT-C security checks (-cert-c)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertC = true;`

EnableCertCpp — check CERT C++ rules

false (default) | true

This property affects Bug Finder only.

Check CERT C++ rules, specified as true or false. To customize which rules are checked, use `CertCpp`.

For more information about the CERT C++ checker, see `Check CERT-C++ security checks (-cert-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertCpp = true;`

EnableCheckersSelectionByFile — Check custom set of coding standard checkers

false (default) | true

Check custom set of coding standard checkers, specified as true or false. Use with `CheckersSelectionByFile` and these coding standards:

- `opts.CodingRulesCodeMetrics.AutosarCpp14='from-file'`
- `opts.CodingRulesCodeMetrics.CertC='from-file'`
- `opts.CodingRulesCodeMetrics.CertCpp='from-file'`
- `opts.CodingRulesCodeMetrics.Iso17961='from-file'`
- `opts.CodingRulesCodeMetrics.JsfSubset='from-file'`

- `opts.CodingRulesCodeMetrics.MisraC3Subset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCppSubset='from-file'`

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;`

EnableCustomRules — Check custom coding rules

false (default) | true

Check custom coding rules, specified as true or false. The file you specify with `CheckersSelectionByFile` defines the custom coding rules.

Use with `EnableCheckersSelectionByFile`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

EnableIso17961 — check ISO-17961 rules

false (default) | true

This property affects Bug Finder only.

Check ISO®/IEC TS 17961 rules, specified as true or false. To customize which rules are checked, use `Iso17961`.

For more information about the ISO-17961 checker, see `Check ISO-17961 security checks (-iso-17961)`.

Example: `opts.CodingRulesCodeMetrics.EnableIso17961 = true;`

EnableJsfc — Check JSF C++ rules

false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfcSubset`.

For more information, see `Check JSF C++ rules (-jsfc-coding-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableJsfc = true;`

EnableMisraC — Check MISRA C:2004 rules

false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

EnableMisraC3 — Check MISRA C:2012 rules

false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

EnableMisraCpp — Check MISRA C++:2008 rules

false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

Iso17961 — Set of ISO-17961 rules to check

'all' (default) | 'decidable' | `polyspace.CodingRulesOptions` object | 'from-file'

This property affects Bug Finder only.

Set of ISO/IEC TS 17961 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check ISO-17961 security checks (-iso-17961)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use 'from-file' for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check ISO/IEC TS 17961 rules, also set `EnableIso17961` to true.

Example: `opts.CodingRulesCodeMetrics.Iso17961 = 'all'`

Data Types: char

JsfSubset — Subset of JSF C++ rules to check

'shall-rules' (default) | 'shall-will-rules' | 'all-rules' | `polyspace.CodingRulesOptions` object | 'from-file'

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use 'from-file' for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: char

Misra3AgcMode — Use the MISRA C:2012 categories for automatically generated code

false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

MisraC3Subset — Subset of MISRA C:2012 rules to check

'mandatory-required' (default) | 'mandatory' | 'single-unit-rules' | 'system-decidable-rules' | 'all' | 'SQ0-subset1' | 'SQ0-subset2' | `polyspace.CodingRulesOptions` object | 'from-file'

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use 'from-file' for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: char

MisraCSubset — Subset of MISRA C:2004 rules to check

'required-rules' (default) | 'single-unit-rules' | 'system-decidable-rules' | 'all-rules' | 'SQ0-subset1' | 'SQ0-subset2' | `polyspace.CodingRulesOptions` object | 'from-file'

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: char

MisraCppSubset — Subset of MISRA C++ rules

`'required-rules'` (default) | `'all-rules'` | `'SQ0-subset1'` | `'SQ0-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: char

EnvironmentSettings

Dos — Consider that file paths are in MS-DOS style

`true` (default) | `false`

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

IncludeFolders — Include folders needed for compilation

cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by `**`, for instance, `'C:\includes**'`. The notation follows the syntax of the `dir` function.

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes', '/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};`

Data Types: `cell`

Includes — Files to be #include-ed by each C file

cell array of files

Files to be `#include`-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h', '/inc/inc_math.h'};`

NoExternC — Ignore linking errors inside extern blocks

false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

PostPreProcessingCommand — Command or script to run on source files after preprocessing

character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

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

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = [pwd, '/replace_keyword.pl'];`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = ' "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl" ';`

StopWithCompileError — Stop analysis if a file does not compile

false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

InputsStubbing

DataRangeSpecifications — Constrain global variables, function inputs, and return values of stubbed functions

file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see “Specify External Constraints”.

For more information about this option, see `Constraint setup (-data-range-specifications)`.

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

DoNotGenerateResultsFor — Files on which you do not want analysis results

'include-folders' (default) | 'all-headers' | 'custom=*folder1[, folder2, ...]*'

Files on which you do not want analysis results, specified by 'include-folders', 'all-headers', or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

Example: `opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'`

GenerateResultsFor — Files on which you want analysis results

'source-headers' (default) | 'all-headers' | 'custom=*folder1[, folder2, ...]*'

Files on which you want analysis results, specified by 'source-headers', 'all-headers', or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

Example: `opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'`

FunctionsToStub — Functions to stub during analysis

cell array of function names

This property affects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

NoDefInitGlob — Consider global variables as uninitialized

false (default) | true

This property affects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see Ignore default initialization of global variables (-no-def-init-glob).

Example: `opts.InputsStubbing.NoDefInitGlob = true`**NoStlStubs — Do not use Polyspace implementations of functions in the Standard Template Library**

false (default) | true

This property applies only to a Code Prover analysis of C++ code.

Do not use Polyspace implementations of functions in the Standard Template Library, specified as true or false.

For more information, see No STL stubs (-no-stl-stubs).

Example: `opts.InputsStubbing.NoStlStubs = true`**StubECoderLookupTables — Specify that the analysis must stub functions in the generated code that use lookup tables**

true (default) | false

This property applies only to a Code Prover analysis of code generated from models.

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions).

Example: `opts.InputsStubbing.StubECoderLookupTables = true`**Macros****DefinedMacros — Macros to be replaced**

cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see Preprocessor definitions (-D).

Example: `opts.Macros.DefinedMacros = {'uint32=int', 'name3=', 'var'}`**UndefinedMacros — Macros to undefine**

cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see Disabled preprocessor definitions (-U).

Example: `opts.Macros.DefinedMacros = {'name1', 'name2'}`

MergedComputingSettings

AddToResultsRepositoryBugFinder — Upload Bug Finder results to Polyspace Metrics web dashboard

false (default) | true

This property affects Bug Finder analysis only.

Upload Bug Finder analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryBugFinder = true;`

AddToResultsRepositoryCodeProver — Upload Code Prover results to Polyspace Metrics web dashboard

false (default) | true

This property affects Code Prover analysis only.

Upload Code Prover analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryCodeProver = true;`

BatchBugFinder — Send Bug Finder analysis to remote server

false (default) | true

This property affects Bug Finder analysis only.

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

BatchCodeProver — Send Code Prover analysis to remote server

false (default) | true

This property affects Code Prover analysis only.

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

FastAnalysis — Run Bug Finder analysis using faster local mode

false (default) | true

This property affects Bug Finder analysis only.

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.Example: `opts.MergedComputingSettings.FastAnalysis = true;`**MergedReporting****EnableReportGeneration — Generate a report after the analysis**

false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.Example: `opts.MergedReporting.EnableReportGeneration = true`**ReportOutputFormat — Output format of generated report**

'Word' (default) | 'HTML' | 'PDF'

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.For more information about the different values, see `Output format (-report-output-format)`.Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`**BugFinderReportTemplate — Template for generating Bug Finder analysis report**

'BugFinderSummary' (default) | 'BugFinder' | 'SecurityCWE' | 'CodeMetrics' | 'CodingStandards'

*This property affects a Bug Finder analysis only.*Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`**CodeProverReportTemplate — Template for generating Code Prover analysis report**

'Developer' (default) | 'CallHierarchy' | 'CodeMetrics' | 'CodingStandards' | 'DeveloperReview' | 'Developer_withGreenChecks' | 'Quality' | 'VariableAccess'

*This property affects a Code Prover analysis only.*Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

Multitasking

ArxmlMultitasking — Specify path of ARXML files to parse for multitasking configuration

cell array of file paths

Specify the path to the ARXML files the software parses to set up your multitasking configuration.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `autosar`.

For more information, see `ARXML files selection (-autosar-multitasking)`

Example: `opts.Multitasking.ArxmlMultitasking={'C:\Polyspace_Workspace\AUTOSAR\myFile.arxml'}`

CriticalSectionBegin — Functions that begin critical sections

cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1', 'function2:cs2'}`

CriticalSectionEnd — Functions that end critical sections

cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1', 'function2:cs2'}`

CyclicTasks — Specify functions that represent cyclic tasks

cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1', 'function2'}`

EnableConcurrencyDetection — Enable automatic detection of certain families of threading functions

false (default) | true

This property affects Code Prover analysis only.

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

EnableExternalMultitasking — Enable automatic multitasking configuration from external file definitions

false (default) | true

Enable multitasking configuration of your projects from external files you provide. Configure multitasking from ARXML files for an AUTOSAR project, or from OIL files for an OSEK project.

Activate this option to enable `Multitasking.ArxmlMultitasking` or `Multitasking.OsekMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.EnableExternalMultitasking = 1`

EnableMultitasking — Configure multitasking manually

false (default) | true

Configure multitasking manually by specifying true. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

EntryPoints — Functions that serve as entry-points to your multitasking application

cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Tasks (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

ExternalMultitaskingType — Specify type of file to parse for multitasking configuration

'osek' (default) | 'autosar'

Specify the type of file the software parses to set up your multitasking configuration:

- For `osek` type, the analysis looks for OIL files in the file or folder paths that you specify.
- For `autosar` type, the analysis looks for ARXML files in the file paths that you specify.

To activate this option, specify `Multitasking.EnableExternalMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.ExternalMultitaskingType = 'autosar'`

Interrupts — Specify functions that represent nonpreemptable interrupts

cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

InterruptsDisableAll — Specify routine that disable interrupts

cell array with one function name

This property affects Bug Finder analysis only.

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

InterruptsEnableAll — Specify routine that reenables interrupts

cell array with one function name

This property affects Bug Finder analysis only.

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

OsekMultitasking — Specify path of OIL files to parse for multitasking configuration'auto' (default) | 'custom=*folder1[, folder2, ...]*'

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In the mode specified with 'auto', the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In the mode specified with 'custom=*folder1[, folder2, ...]*', the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `osek`.

For more information, see `OIL files selection (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom=file_path, dir_path'`

TemporalExclusion — Entry-point functions that cannot execute concurrently

cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where `function1` and `function2` are temporally exclusive, and `function3`, `function4`, and `function 5` are temporally exclusive.

Precision (Affects Code Prover Only)

ContextSensitivity — Store call context information to identify function call that caused errors

'none' (default) | 'auto' | 'custom=function1[,function2,...]'

This property affects Code Prover analysis only.

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or as a character array beginning with `custom=` followed by a list of comma-separated function names.

For more information, see `Sensitivity context (-context-sensitivity)`.

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

ModulesPrecision — Source files you want to verify at higher precision

cell array of file names and precision levels

This property affects Code Prover analysis only.

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: `filename:0level`

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:00', 'file2:03'}`

0Level — Precision level for the verification

2 (default) | 0 | 1 | 3

This property affects Code Prover analysis only.

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-0)`.

Example: `opts.Precision.0Level = 3`

PathSensitivityDelta — Avoid certain verification approximations for code with fewer lines

positive integer

This property affects Code Prover analysis only.

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

Timeout — Time limit on your verification

character vector

This property affects Code Prover analysis only.

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

To — Number of times the verification process runs

'Software Safety Analysis level 2' (default) | 'Software Safety Analysis level 0' |
'Software Safety Analysis level 1' | 'Software Safety Analysis level 3' |
'Software Safety Analysis level 4' | 'Source Compliance Checking' | 'other'

This property affects Code Prover analysis only.

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

Scaling (Affects Code Prover Only)

Inline — Functions on which separate results must be generated for each function call

cell array of function names

This property affects Code Prover analysis only.

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1', 'func2'}`

KLimiting — Limit depth of analysis for nested structures

positive integer

This property affects Code Prover analysis only.

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

TargetCompiler**Compiler — Compiler that builds your source code**

'generic' (default) | 'gnu3.4' | 'gnu4.6' | 'gnu4.7' | 'gnu4.8' | 'gnu4.9' | 'gnu5.x' | 'gnu6.x' | 'gnu7.x' | 'clang3.x' | 'clang4.x' | 'clang5.x' | 'visual9.0' | 'visual10' | 'visual11.0' | 'visual12.0' | 'visual14.0' | 'visual15.x' | 'keil' | 'iar' | 'armcc' | 'armclang' | 'codewarrior' | 'diab' | 'greenhills' | 'iar-ew' | 'renesas' | 'tasking' | 'ti'

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

CppVersion — Specify C++11 standard version followed in code

'defined-by-compiler' (default) | 'cpp03' | 'cpp11' | 'cpp14'

Specify C++ standard version followed in code, specified as a character vector.

For more information, see `C++ standard version (-cpp-version)`.

Example: `opts.TargetCompiler.CppVersion = 'cpp11';`

CVersion — Specify C standard version followed in code

'defined-by-compiler' (default) | 'c90' | 'c99' | 'c11'

Specify C standard version followed in code, specified as a character vector.

For more information, see `C standard version (-c-version)`.

Example: `opts.TargetCompiler.CVersion = 'c90';`

DivRoundDown — Round down quotients from division or modulus of negative numbers

false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see `Division round down (-div-round-down)`.

Example: `opts.TargetCompiler.DivRoundDown = true`

EnumTypeDefinition — Base type representation of enum

'defined-by-compiler' (default) | 'auto-signed-first' | 'auto-unsigned-first'

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see `Enum type definition (-enum-type-definition)`.

Example: `opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'`

IgnorePragmaPack — Ignore #pragma pack directives

false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see `Ignore pragma pack directives (-ignore-pragma-pack)`.

Example: `opts.TargetCompiler.IgnorePragmaPack = true`

Language — Language of analysis`'C-CPP' (default) | 'C' | 'CPP'`

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see `Source code language (-lang)`.

LogicalSignedRightShift — Treatment of signed bit on signed variables`'Arithmetical' (default) | 'Logical'`

Treatment of signed bit on signed variables, specified as `Arithmetical` or `Logical`. For more information, see `Signed right shift (-logical-signed-right-shift)`.

Example: `opts.TargetCompiler.LogicalSignedRightShift = 'Logical'`

NoUliterals — Do not use predefined typedefs for `char16_t` or `char32_t``false (default) | true`

Do not use predefined typedefs for `char16_t` or `char32_t`, specified as `true` or `false`. For more information, see `Block char16/32_t types (-no-uliterals)`.

Example: `opts.TargetCompiler.NoUliterals = true`

PackAlignmentValue — Default structure packing alignment`'defined-by-compiler' (default) | '1' | '2' | '4' | '8' | '16'`

Default structure packing alignment, specified as `'defined-by-compiler'`, `'1'`, `'2'`, `'4'`, `'8'`, or `'16'`. This property is available only for Visual C++ code.

For more information, see `Pack alignment value (-pack-alignment-value)`.

Example: `opts.TargetCompiler.PackAlignmentValue = '4'`

SfrTypes — sfr types`cell array of sfr keywords`

`sfr` types, specified as a cell array of `sfr` keywords using the syntax `sfr_name=size_in_bits`. For more information, see `Sfr type support (-sfr-types)`.

This option only applies when you set `TargetCompiler.Compiler` to `keil` or `iar`.

Example: `opts.TargetCompiler.SfrTypes = {'sfr32=32'}`

SizeTTypeIs — Underlying type of `size_t``'defined-by-compiler' (default) | 'unsigned-int' | 'unsigned-long' | 'unsigned-long-long'`

Underlying type of `size_t`, specified as `'defined-by-compiler'`, `'unsigned-int'`, `'unsigned-long'`, or `'unsigned-long-long'`. See `Management of size_t (-size-t-type-is)`.

Example: `opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'`

Target — Target processor`'i386' (default) | 'arm' | 'arm64' | 'avr' | 'c-167' | 'c166' | 'c18' | 'c28x' | 'c6000' | 'coldfire' | 'hc08' | 'hc12' | 'm68k' | 'mcore' | 'mips' | 'mpc5xx' | 'msp430' | 'necv850'`

```
| 'powerpc' | 'powerpc64' | 'rh850' | 'rl78' | 'rx' | 's12z' | 'sharc21x61' | 'sparc' |
'superh' | 'tms320c3x' | 'tricore' | 'x86_64' | generic target object
```

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see `Target processor type (-target)`.

For more information about creating a generic target, see `polyspace.GenericTargetOptions`.

Example: `opts.TargetCompiler.Target = 'hc12'`

WcharTTypeIs — Underlying type of wchar_t

```
'defined-by-compiler' (default) | 'signed-short' | 'unsigned-short' | 'signed-int' |
'unsigned-int' | 'signed-long' | 'unsigned-long'
```

Underlying type of `wchar_t`, specified as `'defined-by-compiler'`, `'signed-short'`, `'unsigned-short'`, `'signed-int'`, `'unsigned-int'`, `'signed-long'`, or `'unsigned-long'`. See `Management of wchar_t (-wchar-t-type-is)`.

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

VerificationAssumption (Affects Code Prover Only)

ConsiderVolatileQualifierOnFields — Assume that volatile qualified structure fields can have all possible values at any point in code

```
false (default) | true
```

This property affects Code Prover analysis only.

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)`.

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

ConstraintPointersMayBeNull — Specify that environment pointers can be NULL unless constrained otherwise

```
false (default) | true
```

This property affects Code Prover analysis only.

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

FloatRoundingMode — Rounding modes to consider when determining the results of floating-point arithmetic

```
to-nearest (default) | all
```

This property affects Code Prover analysis only.

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode (-float-rounding-mode)`.

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

RespectTypesInFields — Do not cast nonpointer fields of a structure to pointers

false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer fields of a structure to pointers, specified as true or false.

For more information, see `Respect types in fields (-respect-types-in-fields)`.

Example: `opts.VerificationAssumption.RespectTypesInFields = true`

RespectTypesInGlobals — Do not cast nonpointer global variables to pointers

false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer global variables to pointers, specified as true or false.

For more information, see `Respect types in global variables (-respect-types-in-globals)`.

Example: `opts.VerificationAssumption.RespectTypesInGlobals = true`

Other Properties

Author — Project author

username of current user (default) | character vector

Name of project author, specified as a character vector.

For more information, see `-author`.

Example: `opts.Author = 'JaneDoe'`

ImportComments — Import comments and justifications from previous analysis

character vector

To import comments and justifications from a previous analysis, specify the path to the results folder of the previous analysis.

You can also point to a previous results folder to see only new results compared to the previous run. See .

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', 'Module_1', 'BF_Result')`

Prog — Project name

PolyspaceProject (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

ResultsDir — Location to store results

folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

You can also create a separate results folder for each new run. See .

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

Sources — Source files

cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by `*`, for instance, `'C:\src*'`. To specify all files in a folder and its subfolders, use folder path followed by `**`, for instance, `'C:\src**'`. The notation follows the syntax of the `dir` function. See also .

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

Version — Project version number

'1.0' (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

See Also

Topics

"Analysis Options"

Introduced in R2017a

copyTo

Class: polyspace.Options

Package: polyspace

Copy common settings between Polyspace options objects

Syntax

```
optsFrom.copyTo(optsTo)
```

Description

`optsFrom.copyTo(optsTo)` copies the common options from `optsFrom` to `optsTo`. The options objects do not need to be the same type of options object. This method copies only properties that are common between the two objects.

Input Arguments

optsFrom — Options object you want to copy properties from

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object that you want to copy properties from, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

optsTo — Options object you want to copy properties to

`polyspace.Options` object

Option object that you want to copy properties to, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

Examples

Copy Polyspace Options Object

This example shows how to set the properties of one options object and then copy that object to another one.

Create a Polyspace options object and set properties.

```
opts1 = polyspace.Options();  
opts1.Prog = 'DataRaceProject';  
opts1.Sources = {'datarace.c'};  
opts1.TargetCompiler.Compiler = 'gnu4.9';
```

Create another object and use `copyTo` to copy over options from the previous object.

```
opts2 = polyspace.Options();  
opts1.copyTo(opts2);
```

See Also

[generateProject](#) | [polyspace.Options](#)

Introduced in R2016b

generateProject

Class: polyspace.Options

Package: polyspace

Generate psprj project from options object

Syntax

```
opts.generateProject(projectName)
```

Description

`opts.generateProject(projectName)` creates a `.psprj` project called `projectName` from the options specified in the `polyspace.Options` object `opts`. You can open a `.psprj` project in the user interface of the Polyspace desktop products.

Input Arguments

opts — Options object to convert into a psprj file

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object convert into a psprj file, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

projectName — Project file name

character vector

Project file name specified as a character vector. This argument is used as the name of the psprj file.

Example: `'myProject'`

Examples

Generate Project from a Bug Finder Options Object

This example shows how to create and use a Polyspace project that was generated from an options object.

Create a Bug Finder object and set properties.

```
sources = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', ...  
    'sources', 'numerical.c');  
opts = polyspace.Options();  
opts.Prog = 'MyProject';  
opts.Sources = {sources};  
opts.TargetCompiler.Compiler = 'gnu4.7';
```

Generate a Polyspace project. Name the project using the Prog property.


```
psprj = opts.generateProject(opts.Prog);
```

Run a Bug Finder analysis using one of these commands. Both commands produce identical analysis results. The only difference is that the `psprj` project can be rerun in the Polyspace interface.

```
polyspaceBugFinder(psprj, '-nodesktop');  
polyspaceBugFinder(opts);
```

To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

Tips

If you want to include an options object in a `pslinkoptions` object:

- 1 Use this method to convert your object to a project.
- 2 Add the project to the `pslinkoptions` property `PrjConfig`.
- 3 Turn on the property `EnablePrjConfig`.

See Also

`copyTo` | `polyspace.Options`

Introduced in R2016b

toScript

Class: polyspace.Options

Package: polyspace

Add Polyspace options object definition to a script

Syntax

```
filePath = opts.toScript(fileName,positionInScript)
```

Description

`filePath = opts.toScript(fileName,positionInScript)` adds the properties of a `polyspace.Options` object to a MATLAB script. The script shows the values assigned to all the properties of the object. You can run the script later to define the object in the MATLAB workspace and use it.

Input Arguments

opts — Options object with Polyspace analysis options

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object to store in MATLAB script, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

fileName — Script name

character vector

Name or path to script, specified as a character vector. If you specify a relative path, the script is created in subfolder of the current working folder.

Example: `'runPolyspace.m'`

positionInScript — Where to add object definition

'create' (default) | 'append'

Position in script where the object properties are added, specified as 'create' or 'append'. If you specify 'append', the object properties are added to the end of an existing script. Otherwise, a new script is created.

Output Arguments

filePath — Full path to script

character vector

Full path to script, specified as a character vector.

Example: `'C:\myScripts\runPolyspace.m'`

See Also

copyTo | generateProject | polyspace.Options

Introduced in R2017b

run

Run a Polyspace analysis

Syntax

```
run(proj, product)
```

Description

`status = run(proj, product)` runs a Polyspace Bug Finder or Polyspace Code Prover analysis using the configuration specified in the `polyspace.Project` object `proj`. The analysis results are also stored in `proj`.

Input Arguments

proj — Polyspace project
`polyspace.Project` object

Polyspace project with configuration and results, specified as a `polyspace.Project` object.

product — Type of analysis
'bugFinder' | 'codeProver'

Type of analysis to run.

Output Arguments

status — Results of a Code Prover analysis
`true` | `false`

Status of analysis. If the analysis fails, the status is `false`. Otherwise, it is `true`.

The analysis can fail for multiple reasons:

- You provide source files that do not exist.
- None of your files compile. Even if one file compiles, unless you set the property `StopWithCompileError` to `true`, the analysis succeeds and returns a `true` status.

There can be many other reasons why the analysis fails. If the analysis fails, in your results folder, check the log file. You can see the results folder using the `Configuration` property of the `polyspace.Project` object:

```
proj = polyspace.Project;  
proj.Configuration.ResultsDir
```

The log file is named `Polyspace_R20##n_ProjectName_date-time.log`.

Examples

Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project
```

```
% Configure analysis
```

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...  
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};  
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';  
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
```

```
% Run analysis
```

```
bfStatus = run(proj, 'bugFinder');
```

```
% Read results
```

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

Introduced in R2017b

getSummary

Class: polyspace.CodeProverResults

Package: polyspace

View number of run-time checks organized by color and file

Syntax

```
resObj.getSummary(resultsType)
```

Description

`resSummary = resObj.getSummary(resultsType)` returns the distribution of results of type `resultsType` in a Code Prover result set denoted by the `polyspace.CodeProverResults` object `resObj`. For instance, if you choose to see run-time checks, you see how many red, orange, gray and green checks are present in each file.

Input Arguments

`resultsType` — Type of Code Prover analysis result

'runtime' (default) | 'misraC' | 'misraCAGC' | 'misraCPP' | 'misraC2012' | 'jsf' | 'metrics' | 'customRules'

Type of result, specified as a character vector.

Entry	Meaning
'runtime'	Checks for run-time errors.
'misraC'	MISRA C:2004 rules.
'misraCAGC'	MISRA C:2004 rules for generated code.
'misraCPP'	MISRA® C++ rules.
'misraC2012'	MISRA C:2012 rules.
'jsf'	JSF® C++ rules.
'metrics'	Code complexity metrics.
'customRules'	Custom rules enforcing naming conventions for identifiers.

Output Arguments

`resSummary` — Distribution of run-time checks by check color and file

table

Distribution of run-time checks by check color and file, specified as a table. For instance, an extract of the table looks like this:

File	Proven	Green	Red	Gray	Orange
file1.c	92.0%	87	3	2	8
file2.c	97.7%	41	0	1	1

The table above shows that `file1.c` has:

- 3 red, 2 gray and 8 orange checks.
- 92% of operations proven.

In other words, of every 100 operations that the verification checked, 92 operations were proven green, red or gray. See “Code Prover Result and Source Code Colors” (Polyspace Code Prover Access).

For more information on MATLAB tables, see “Tables” (MATLAB).

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...
'Module_1', 'CP_Result');
userResPath = tempname;
copyfile(resPath, userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = resObj.getSummary('runtime');
resTable = resObj.getResults();
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a main function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project
```

```
% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...
'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
```

```
proj.Configuration.CodeProverVerification.MainGenerator = true;
```

```
% Run analysis
```

```
cpStatus = proj.run('codeProver');
```

```
% Read results
```

```
cpSummary = proj.Results.getResults('readable');
```

See Also

`polyspace.CodeProverResults`

Topics

“Code Prover Result and Source Code Colors” (Polyspace Code Prover Access)

Introduced in R2017a

getResults

Class: polyspace.CodeProverResults

Package: polyspace

Read Code Prover results into MATLAB table

Syntax

```
resObj.getResults(content)
```

Description

`resTable = resObj.getResults(content)` returns a table showing all results in a Code Prover result set denoted by the `polyspace.CodeProverResults` object `resObj`. You can manipulate the table to produce graphs and statistics about your results that you cannot obtain readily from the user interface.

Input Arguments

content — Result information to include

'full' (default) | 'readable'

Amount of information to be included for each result. If you specify 'full', all information is included. If you specify 'readable', the following information is not included:

- ID: Unique number for a result for the current analysis.
- Group: Check groups (Polyspace Code Prover Access), MISRA C:2012 groups (Polyspace Code Prover Access), etc.
- Status, Severity, Comment: Information that *you* enter about a result.

If you do not specify this argument, the full table is included.

Output Arguments

resTable — Results of a Code Prover analysis

table

Table showing all results from a single Code Prover analysis. For each result, the table has information such as file, family, and so on. If a particular information is not available for a result, the entry in the table states `<undefined>`.

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example', ...  
'Module_1','CP_Result');  
userResPath = tempname;  
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);  
resTable = getResults (resObj);
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project
```

```
% Configure analysis
```

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...  
'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};  
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';  
proj.Configuration.ResultsDir = fullfile(pwd, 'results');  
proj.Configuration.CodeProverVerification.MainGenerator = true;
```

```
% Run analysis
```

```
cpStatus = proj.run('codeProver');
```

```
% Read results
```

```
cpSummary = proj.Results.getResults('readable');
```

See Also

`polyspace.CodeProverResults`

Introduced in R2017a

variableAccess

Class: polyspace.CodeProverResults

Package: polyspace

View global variables along with read/write operations in C/C++ code

Syntax

```
resObj.variableAccess()
```

Description

`varList = resObj.variableAccess()` returns the distribution of global variables in a Code Prover result set denoted by the `polyspace.CodeProverResults` object `resObj`. The list also contains all read and write operations on the global variables.

Output Arguments

varList — Distribution of global variables

table

Table showing all global variables from a single Code Prover analysis along with read and write operations on them.

- For each global variable, the table has information such as data type, number of times accessed, and so on.
- For each read or write operation, the table has information such as file and function name, line number, and so on.

If a particular information is not available for a result, the entry in the table states <undefined>.

Examples

Read Global Variables from Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...
'Module_1', 'CP_Result');
userResPath = tempname;
copyfile(resPath, userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read list of global variables to MATLAB tables using the object.

```
varList = resObj.variableAccess;
```

Run Analysis and Read Global Variables to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project
```

```
% Configure analysis
```

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...  
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};  
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';  
proj.Configuration.ResultsDir = fullfile(pwd, 'results');  
proj.Configuration.CodeProverVerification.MainGenerator = true;
```

```
% Run analysis
```

```
cpStatus = proj.run('codeProver');
```

```
% Read results
```

```
cpSummary = proj.Results.variableAccess;
```

See Also

`polyspace.CodeProverResults`

Introduced in R2017a

Analysis Options

Source code language (-lang)

Specify language of source files

Description

Specify the language of your source files. Before specifying other configuration options, choose this option because other options change depending on your language selection.

If you add files during project setup, the language selection can change from the default.

Files Added	Source Code Language
Only files with extension .c	C
Only files with extension .cpp or .cc	C++
Files with extension .c, .cpp, and .cc	C-C++

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 2-2 for ways in which the source code language can be automatically determined.

Command line: Use the option `-lang`. See “Command-Line Information” on page 2-3.

Settings

Default: Based on file extensions.

C

If your project contains only C files, choose this setting. This value restricts the verification to C language conventions. All files are interpreted as C files, regardless of their file extension.

C++

If your project contains only C++ files, choose this setting. This value restricts the verification to C++ language conventions. All files are interpreted as C++ files, regardless of their file extension.

C-C++

If your project contains C and C++ source files, choose this setting. This value allows for C and C++ language conventions. .c files are interpreted as C files. Other file extensions are interpreted as C++ files.

Dependencies

- The language option allows and disallows many options and option values. Some options change depending on your language selection. For more information, see the individual analysis option pages.
- If you create a Polyspace project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is determined by the file extensions.

For a project with both `.c` and `.cpp` files, the language option `C-CPP` is used. In the subsequent analysis, each file is compiled based on the language standard determined by the file extensions.

Command-Line Information

Parameter: `-lang`

Value: `c | cpp | c-cpp`

Default: Based on file extensions

Example (Bug Finder): `polyspace-bug-finder -lang c-cpp -sources "file1.c, file2.cpp"`

Example (Code Prover): `polyspace-code-prover -lang cpp -sources "file1.cpp, file2.cpp"`

Example: `polyspace-code-prover-server -lang cpp -sources "file1.cpp, file2.cpp"`

Example (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c, file2.c"`

Example (Code Prover): `polyspace-code-prover -lang c -sources "file1.c, file2.c"`

Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c, file2.c"`

Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c, file2.c"`

See Also

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

C standard version (-c-version)

Specify C language standard followed in source code

Description

Specify the C language standard that you follow in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 2-5 for other options that you must enable.

Command line: Use the option `-c-version`. See “Command-Line Information” on page 2-5.

Why Use This Option

Use this option so that Polyspace can allow features specific to a C standard version during compilation. For instance, if you compile with GCC using the flag `-ansi` or `-std=c90`, specify `c90` for this option. If you are not sure of the language standard, specify `defined-by-compiler`.

For instance, suppose you use the boolean data type `_Bool` in your code. This type is defined in the C99 standard but unknown in prior standards such as C90. If the Polyspace compilation follows the C90 standard, you can see compilation errors.

Some MISRA C rules are different based on whether you use the C90 or C99 standard. For instance, MISRA C C:2012 Rule 5.2 requires that identifiers in the same scope and name space shall be distinct. If you use the C90 standard, different identifiers that have the same first 31 characters violate this rule. If you use the C99 standard, the number of characters increase to 63.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

The analysis uses a standard based on your specification for `Compiler (-compiler)`.

See “C/C++ Language Standard Used in Polyspace Analysis”.

`c90`

The analysis uses the C90 Standard (ISO/IEC 9899:1990).

`c99`

The analysis uses the C99 Standard (ISO/IEC 9899:1999).

`c11`

The analysis uses the C11 Standard (ISO/IEC 9899:2011).

Dependencies

- This option is available only if you set `Source code language (-lang)` to C or C-CPP.
- If you create a project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is automatically determined from your build system.

If the build system uses different standards for different files, the subsequent Polyspace analysis can emulate your build system and use different standards for compiling those files. If you open such a project in the Polyspace user interface, the option value is shown as `defined-by-compiler`. However, instead of one standard, Polyspace uses the hidden option `-options-for-sources` to associate different standards with different files.

Command-Line Information

Parameter: `-c-version`

Value: `defined-by-compiler | c90 | c99 | c11`

Default: `defined-by-compiler`

Example (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c" -c-version c90`

Example (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c" -c-version c90`

Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c" -c-version c90`

Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c" -c-version c90`

See Also

`C++ standard version (-cpp-version) | Source code language (-lang)`

Topics

“Prepare Scripts for Polyspace Analysis”

“C/C++ Language Standard Used in Polyspace Analysis”

“C11 Language Elements Supported in Polyspace”

C++ standard version (-cpp-version)

Specify C++ language standard followed in source code

Description

Specify the C++ language standard that you follow in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 2-7 for other options that you must enable.

Command line: Use the option `-cpp-version`. See “Command-Line Information” on page 2-7.

Why Use This Option

Use this option so that Polyspace can allow features from a specific version of the C++ language standard during compilation. For instance, if you compile with GCC using the flag `-std=c++11` or `-std=gnu++11`, specify `cpp11` for this option. If you are not sure of the language standard, specify `defined-by-compiler`.

For instance, suppose you use range-based `for` loops. This type of `for` loop is defined in the C++11 standard but unrecognized in prior standards such as C++03. If the Polyspace compilation uses the C++03 standard, you can see compilation errors.

To check if your compiler allows features specific to a standard, compile code with macros specific to the standard using compiler settings that you typically use. For instance, to check for C++11-specific features, compile this code. The code contains a C++11-specific keyword `nullptr`. If the macro `__cplusplus` is not `201103L` (indicating C++11), this keyword is used and causes a compilation error.

```
#if defined(__cplusplus) && __cplusplus >= 201103L
    /* C++11 compiler */
#else
    void* ptr = nullptr;
#endif
```

If the code compiles, use `cpp11` for this option.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

The analysis uses a standard based on your specification for `Compiler (-compiler)`.

See “C/C++ Language Standard Used in Polyspace Analysis”.

`cpp03`

The analysis uses the C++03 Standard (ISO/IEC 14882:2003).

cpp11

The analysis uses the C++11 Standard (ISO/IEC 14882:2011).

cpp14

The analysis uses the C++14 Standard (ISO/IEC 14882:2014).

Dependencies

- This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.
- If you create a project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is automatically determined from your build system.

If the build system uses different standards for different files, the subsequent Polyspace analysis can emulate your build system and use different standards for compiling those files. If you open such a project in the Polyspace user interface, the option value is shown as `defined-by-compiler`. However, instead of one standard, Polyspace uses multiple standards for compiling the files. The analysis uses the hidden option `-options-for-sources` to associate different standards with different files.

Command-Line Information

Parameter: `-cpp-version`

Value: `defined-by-compiler` | `cpp03` | `cpp11` | `cpp14`

Default: `defined-by-compiler`

Example (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c" -cpp-version cpp11`

Example (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c" -cpp-version cpp11`

Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c" -cpp-version cpp11`

Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c" -cpp-version cpp11`

See Also

`C standard version (-c-version)` | `Source code language (-lang)`

Topics

"Prepare Scripts for Polyspace Analysis"

"C/C++ Language Standard Used in Polyspace Analysis"

"C++11 Language Elements Supported in Polyspace"

"C++14 Language Elements Supported in Polyspace"

Target processor type (-target)

Specify size of data types and endianness by selecting a predefined target processor

Description

Specify the processor on which you deploy your code.

The target processor determines the sizes of fundamental data types and the endianness of the target machine. You can analyze code intended for an unlisted processor type by using one of the other processor types, if they share common data properties.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. To see the sizes of types, click the **Edit** button to the right of the **Target processor type** drop-down list.

For some compilers, in the user interface, you see only the processors allowed for that compiler. For these compilers, you also cannot see the data type sizes in the user interface. See the links in the table below for the data type sizes.

Command line: Use the option `-target`. See “Command-Line Information” on page 2-10.

Why Use This Option

You specify a target processor so that some of the Polyspace run-time checks are tailored to the data type sizes and other properties of that processor.

For instance, a variable can overflow for smaller values on a 32-bit processor such as i386 compared to a 64-bit processor such as x86_64. If you select x86_64 for your Polyspace analysis, but deploy your code to the i386 processor, your Polyspace results are not always applicable.

Once you select a target processor, you can specify if the default sign of char is signed or unsigned. To determine which signedness to specify, compile this code using the compiler settings that you typically use:

```
#include <limits.h>
int array[(char)UCHAR_MAX]; /* If char is signed, the array size is -1
```

If the code compiles, the default sign of char is unsigned. For instance, on a GCC compiler, the code compiles with the `-fsigned-char` flag and fails to compile with the `-funsigned-char` flag.

Settings

Default: i386

This table shows the size of each fundamental data type that Polyspace considers. For some targets, you can modify the default size by clicking the **Edit** button to the right of the **Target processor type** drop-down list. The optional values for those targets are shown in [brackets] in the table.

Target	char	short	int	long	long long	float	double	long double ^a	ptr	Default sign of char	endian	Alignment
i386	8	16	32	32	64	32	64	96	32	signed	Little	32
sparc	8	16	32	32	64	32	64	128	32	signed	Big	64
m68k ^b	8	16	32	32	64	32	64	96	32	signed	Big	64
powerpc	8	16	32	32	64	32	64	128	32	unsigned	Big	64
c-167	8	16	16	32	32	32	64	64	16	signed	Little	64
tms320c3x	32	32	32	32	64	32	32	64	32	signed	Little	32
sharc21x61	32	32	32	32	64	32	32 [64]	32 [64]	32	signed	Little	32
necv850	8	16	32	32	32	32	32	64	32	signed	Little	32 [16, 8]
hc08 ^c	8	16	16 [32]	32	32	32	32 [64]	32 [64]	16 ^d	unsigned	Big	32 [16]
hc12	8	16	16 [32]	32	32	32	32 [64]	32 [64]	32 ^e	signed	Big	32 [16]
mpc5xx	8	16	32	32	64	32	32 [64]	32 [64]	32	signed	Big	32 [16]
c18	8	16	16	32 [24] ^e	32	32	32	32	16 [24]	signed	Little	8
x86_64	8	16	32	64 [32] ^f	64	32	64	128	64	signed	Little	64 [32]
mcpu... (Advanced) ^g	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]
Targets for ARM [®] v5 compiler	See ARM v5 Compiler (-compiler armcc).											
Targets for ARM v6 compiler	See ARM v6 Compiler (-compiler armclang).											
Targets for NPX CodeWarrior [®] compiler	See NXP CodeWarrior Compiler (-compiler codewarrior).											
Targets for Cosmic compiler	See Cosmic Compiler (-compiler cosmic).											
Targets for Diab compiler	See Diab Compiler (-compiler diab).											
Targets for Green Hills [®] compiler	See Green Hills Compiler (-compiler greenhills).											

Target	char	short	int	long	long long	float	double	long double ^a	ptr	Default sign of char	endian	Alignment
Targets for IAR Embedded Workbench compiler	See IAR Embedded Workbench Compiler (-compiler iar-ew).											
Targets for MPLAB XC8 C compiler	See MPLAB XC8 C Compiler (-compiler microchip)											
Targets for Renesas [®] compiler	See Renesas Compiler (-compiler renesas).											
Targets for TASKING compiler	See TASKING Compiler (-compiler tasking).											
Targets for Texas Instruments [™] compiler	See Texas Instruments Compiler (-compiler ti).											

- a. For targets where the size of long double is greater than 64 bits, the size used for computations is not always the same as the size listed in this table. The exceptions are:
- For targets i386, x86_64 and m68k, 80 bits are used for computations, following the practice in common compilers.
 - For the target tms320c3x, 40 bits are used for computation, following the TMS320C3x specifications.
 - If you use a Visual compiler, the size of long double used for computations is the same as size of double, following the specification of Visual C++ compilers.
- b. The M68k family (68000, 68020, and so on) includes the “ColdFire” processor
- c. Non-ANSI C specified keywords and compiler implementation-dependent pragmas and interrupt facilities are not taken into account by this support
- d. All kinds of pointers (near or far pointer) have 2 bytes (hc08) or 4 bytes (hc12) of width physically.
- e. The c18 target supports the type short long as 24 bits in size.
- f. Use option -long-is-32bits to support Microsoft[®] C/C++ Win64 target.
- g. mcpu is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets. For more information, see Generic target options.

Tips

If your processor is not listed, use a similar processor that shares the same characteristics, or create an mcpu generic target processor. See Generic target options.

You can also create a custom target by explicitly stating sizes of fundamental types and so on with the option -custom-target.

Command-Line Information

Parameter: -target

Value: i386 | sparc | m68k | powerpc | c-167 | tms320c3x | sharc21x61 | necv850 | hc08 | hc12 | mpc5xx | c18 | x86_64 | mcpu

Default: i386

Example (Bug Finder): polyspace-bug-finder -target m68k

Example (Code Prover): polyspace-code-prover -target m68k

Example (Bug Finder Server): polyspace-bug-finder-server -target m68k

Example (Code Prover Server): polyspace-code-prover-server -target m68k

You can override the default values for some targets by using specific command-line options. See the section **Command-Line Options** in Generic target options.

See Also

Polyspace Analysis Options

-custom-target

Polyspace Results

Higher Estimate of Local Variable Size | Lower Estimate of Local Variable Size

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Generic target options

Specify size of data types and endianness by creating your own target processor

Description

If a target processor is not directly supported by Polyspace, you can create your own target. You specify the target mcu representing a generic "Micro Controller/Processor Unit" and then explicitly specify sizes of fundamental data types, endianness and other characteristics.

Settings

In the user interface of the Polyspace desktop products, the **Generic target options** dialog box opens when you set the **Target processor type** to mcu. The **Target processor type** option is available on the **Target & Compiler** node in the **Configuration** pane.

	8bits	16bits	32bits	64bits	
Char	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="checkbox"/> Signed
Short	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	
Int	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	
Long	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	
Long long	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	
Float	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	
Double/Long double	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	
Pointer	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	
Alignment	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	

Use the dialog box to specify the name of a new mcu target, for example `My_target`. That new target is added to the **Target processor type** option list.

Default characteristics of a new target: listed as *type [size]*

- `char [8]`
- `short [16]`
- `int [16]`
- `long [32]`

- *long long* [32]
- *float* [32]
- *double* [32]
- *long double* [32]
- *pointer* [16]
- *alignment* [32]
- *char* is signed
- *endianness* is little-endian

Dependency

A custom target can only be created when `Target processor type` (`-target`) is set to `mcpu`.

A custom target is not available when `Compiler` (`-compiler`) is set to one of the `visual*` options.

Command-Line Options

When using the command line, use `-target mcpu` along with these target specification options.

Option	Description	Available With	Example
<code>-little-endian</code>	<p>Little-endian architectures are Less Significant byte First (LSF). For example: i386.</p> <p>Specifies that the less significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0xFF) and the most significant byte (0x00) at the second byte.</p>	<code>mcpu</code>	<pre>polyspace-code-prover-server -lang c -target mcpu -little-endian</pre>

Option	Description	Available With	Example
<code>-big-endian</code>	<p>Big-endian architectures are Most Significant byte First (MSF). For example: SPARC, m68k.</p> <p>Specifies that the most significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0x00) and the less significant byte (0xFF) at the second byte.</p>	mcpu	<pre>polyspace-code-prover-server -target mcpu -big-endian</pre>
<code>-default-sign-of-char [signed unsigned]</code>	<p>Specify default sign of char.</p> <p><code>signed</code>: Specifies that char is signed, overriding target's default.</p> <p><code>unsigned</code>: Specifies that char is unsigned, overriding target's default.</p>	All targets	<pre>polyspace-code-prover-server -default-sign-of-char unsigned -target mcpu</pre>
<code>-char-is-16bits</code>	<p>char defined as 16 bits and all objects have a minimum alignment of 16 bits</p> <p>Incompatible with <code>-short-is-8bits</code> and <code>-align 8</code></p>	mcpu	<pre>polyspace-code-prover-server -target mcpu -char-is-16bits</pre>
<code>-short-is-8bits</code>	Define short as 8 bits, regardless of sign	mcpu	<pre>polyspace-code-prover-server -target mcpu -short-is-8bits</pre>
<code>-int-is-32bits</code>	Define int as 32 bits, regardless of sign. Alignment is also set to 32 bits.	mcpu, hc08, hc12, mpc5xx	<pre>polyspace-code-prover-server -target mcpu -int-is-32bits</pre>
<code>-long-is-32bits</code>	<p>Define long as 32 bits, regardless of sign. Alignment is also set to 32 bits.</p> <p>If your project sets <code>int</code> to 64 bits, you cannot use this option.</p>	All targets	<pre>polyspace-code-prover-server -target mcpu -long-is-32bits</pre>

Option	Description	Available With	Example
<code>-long-long-is-64bits</code>	Define <code>long long</code> as 64 bits, regardless of sign. Alignment is also set to 64 bits.	<code>mcpu</code>	<code>polyspace-code-prover-server -target mcpu -long-long-is-64bits</code>
<code>-double-is-64bits</code>	Define <code>double</code> and <code>long double</code> as 64 bits, regardless of sign.	<code>mcpu</code> , <code>sharc21x61</code> , <code>hc08</code> , <code>hc12</code> , <code>mpc5xx</code>	<code>polyspace-code-prover-server -target mcpu -double-is-64bits</code>
<code>-pointer-is-24bits</code>	Define <code>pointer</code> as 24 bits, regardless of sign.	<code>c18</code>	<code>polyspace-code-prover-server -target c18-pointer-is-24bits</code>
<code>-pointer-is-32bits</code>	Define <code>pointer</code> as 32 bits, regardless of sign.	<code>mcpu</code>	<code>polyspace-code-prover-server -target mcpu -pointer-is-32bits</code>
<code>-align [32 16 8]</code>	Specifies the largest alignment of struct or array objects to the 32, 16 or 8 bit boundaries. Consequently, the array or struct storage is strictly determined by the size of the individual data objects without member and end padding.	<code>mcpu</code> , <code>hc08</code> , <code>hc12</code> , <code>mpc5xx</code> . Other than <code>mcpu</code> , all targets support only 16 or 32 bits.	<code>polyspace-code-prover-server -target mcpu -align 16</code>

See also:

- Management of `wchar_t` (`-wchar-t-type-is`)
- Management of `size_t` (`-size-t-type-is`)
- Enum type definition (`-enum-type-definition`)

You can also use the option `-custom-target` to specify sizes in bytes of fundamental data types, signedness of plain `char`, alignment of structures and underlying types of standard typedef-s such as `size_t`, `wchar_t` and `ptrdiff_t`.

Examples

Targets for GCC Based Compilers

If you select one of the `gnu#.x` compilers for `Compiler` (`-compiler`), you can specify one of the supported target processor types. See `Target processor type` (`-target`). If a target processor type is not directly listed as supported, you can create the target by using this option.

For instance, you can create these targets:

- **Tricore:** Use these options:

```
-target mcpu  
-int-is-32bits  
-long-long-is-64bits  
-double-is-64bits  
-pointer-is-32bits  
-enum-type-definition auto-signed-first  
-wchar-t-type-is signed-int
```

- **PowerPC:** Use these options:

```
-target mcpu  
-int-is-32bits  
-long-long-is-64bits  
-double-is-64bits  
-pointer-is-32bits  
-wchar-t-type-is signed-int
```

- **ARM:** Use these options:

```
-target mcpu  
-int-is-32bits  
-long-long-is-64bits  
-double-is-64bits  
-pointer-is-32bits  
-enum-type-definition auto-signed-first  
-wchar-t-type-is unsigned-int
```

- **MSP430:** Use these options:

```
-target mcpu  
-long-long-is-64bits  
-double-is-64bits  
-wchar-t-type-is signed-long  
-align 16
```

See Also

Target processor type (-target)

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Compiler (-compiler)

Specify the compiler that you use to build your source code

Description

Specify the compiler that you use to build your source code.

Polyspace fully supports the most common compilers used to develop embedded applications. See the list below. For these compilers, you can run analysis simply by specifying your compiler and target processor. For other compilers, specify `generic` as compiler name. If you face compilation errors, explicitly define compiler-specific extensions to work around the errors.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-compiler`. See “Command-Line Information” on page 2-23.

Why Use This Option

Polyspace uses this information to interpret syntax that is not part of the C/C++ Standard, but comes from language extensions.

For example, the option allows additional language keywords, such as `sfr`, `sbit`, and `bit`. If you do not specify your compiler, these additional keywords can cause compilation errors during Polyspace analysis.

Polyspace does not actually invoke your compiler for compilation. In particular:

- You cannot specify compiler flags directly in the Polyspace analysis. To emulate your compiler flags, trace your build command or manually specify equivalent Polyspace analysis options. See “Specify Target Environment and Compiler Behavior”.
- Code Prover has a linking policy that is stricter than regular compilers. For instance, if your compiler allows declaration mismatches with specific compiler options, you cannot emulate this linking policy in Code Prover. See “Troubleshoot Compilation and Linking Errors”.

Settings

Default: `generic`

`generic`

Analysis allows only standard syntax.

The language standard is determined by your choice for the following options:

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

If you do not specify a standard explicitly, the standard depends on your choice of compiler.

gnu3.4

Analysis allows GCC 3.4 syntax.

gnu4.6

Analysis allows GCC 4.6 syntax.

gnu4.7

Analysis allows GCC 4.7 syntax.

For unsupported GCC extensions, see “Limitations” on page 2-21.

gnu4.8

Analysis allows GCC 4.8 syntax.

For unsupported GCC extensions, see “Limitations” on page 2-21.

gnu4.9

Analysis allows GCC 4.9 syntax.

For unsupported GCC extensions, see “Limitations” on page 2-21.

gnu5.x

Analysis allows GCC 5.1, 5.2, 5.3, and 5.4 syntax.

If you select `gnu5.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-21.

gnu6.x

Analysis allows GCC 6.1, 6.2, and 6.3 syntax.

If you select `gnu6.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-21.

gnu7.x

Analysis allows GCC 7.1, 7.2, and 7.3 syntax.

If you select `gnu7.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-21.

clang3.x

Analysis allows Clang 3.5, 3.6, 3.7, 3.8, and 3.9 syntax.

clang4.x

Analysis allows Clang 4.0.0, and 4.0.1 syntax.

clang5.x

Analysis allows Clang 5.0.0, and 5.0.1 syntax.

visual9.0

Analysis allows Microsoft Visual C++ 2008 syntax.

visual10.0

Analysis allows Microsoft Visual C++ 2010 syntax.

This option implicitly enables the option `-no-stl-stubs`.

visual11.0

Analysis allows Microsoft Visual C++ 2012 syntax.

This option implicitly enables the option `-no-stl-stubs`.

visual12.0

Analysis allows Microsoft Visual C++ 2013 syntax.

This option implicitly enables the option `-no-stl-stubs`.

visual14.0

Analysis allows Microsoft Visual C++ 2015 syntax (supports Microsoft Visual Studio® update 2).

This option implicitly enables the option `-no-stl-stubs`.

visual15.x

Analysis allows Microsoft Visual C++ 2017 syntax (supports Microsoft Visual Studio versions 15.0 up to 15.7).

This option implicitly enables the option `-no-stl-stubs`.

keil

Analysis allows non-ANSI® C syntax and semantics associated with the Keil products from ARM (www.keil.com).

iar

Analysis allows non-ANSI C syntax and semantics associated with the compilers from IAR Systems (www.iar.com).

armcc

Analysis allows non-ANSI C syntax and semantics associated with the ARM v5 compiler.

If you select `armcc`, in the user interface of the Polyspace desktop products, the option **Target processor type** (`-target`) shows only the targets that are allowed for the ARM v5 compiler. See **ARM v5 Compiler** (`-compiler armcc`).

armclang

Analysis allows non-ANSI C syntax and semantics associated with the ARM v6 compiler.

If you select `armclang`, in the user interface of the Polyspace desktop products, the option **Target processor type** (`-target`) shows only the targets that are allowed for the ARM v6 compiler. See **ARM v6 Compiler** (`-compiler armclang`).

codewarrior

Analysis allows non-ANSI C syntax and semantics associated with the NXP CodeWarrior compiler.

If you select `codewarrior`, in the user interface of the Polyspace desktop products, the option **Target processor type** (`-target`) shows only the targets that are allowed for the NXP CodeWarrior compiler. See **NXP CodeWarrior Compiler** (`-compiler codewarrior`).

cosmic

Analysis allows non-ANSI C syntax and semantics associated with the Cosmic compiler.

If you select `cosmic`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Cosmic compiler. See `Cosmic Compiler (-compiler cosmic)`.

diab

Analysis allows non-ANSI C syntax and semantics associated with the Wind River® Diab compiler.

If you select `diab`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the NXP CodeWarrior compiler. See `Diab Compiler (-compiler diab)`.

greenhills

Analysis allows non-ANSI C syntax and semantics associated with a Green Hills compiler.

If you select `greenhills`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for a Green Hills compiler. See `Green Hills Compiler (-compiler greenhills)`.

iar-ew

Analysis allows non-ANSI C syntax and semantics associated with the IAR Embedded Workbench compiler.

If you select `iar-ew`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the IAR Embedded Workbench compiler. See `IAR Embedded Workbench Compiler (-compiler iar-ew)`.

microchip

Analysis allows non-ANSI C syntax and semantics associated with the MPLAB XC8 C compiler.

If you select `microchip`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the MPLAB XC8 C compiler. See `MPLAB XC8 C Compiler (-compiler microchip)`.

renesas

Analysis allows non-ANSI C syntax and semantics associated with the Renesas compiler.

If you select `renesas`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Renesas compiler. See `Renesas Compiler (-compiler renesas)`.

tasking

Analysis allows non-ANSI C syntax and semantics associated with the TASKING compiler.

If you select `tasking`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the TASKING compiler. See `TASKING Compiler (-compiler tasking)`.

ti

Analysis allows non-ANSI C syntax and semantics associated with the Texas Instruments compiler.

If you select `ti`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Texas Instruments compiler. See `Texas Instruments Compiler (-compiler ti)`.

cosmic

Analysis allows non-ANSI C syntax and semantics associated with the compiler used in the Cosmic software development tools.

If you select `cosmic`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Cosmic compiler.

Tips

- Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.
 - To override the macro definition, use the option `Preprocessor definitions (-D)`.
 - To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.
- If you use a Visual Studio compiler, you must use a `Target processor type (-target)` option that sets `long long` to 64 bits. Compatible targets include: `i386`, `sparc`, `m68k`, `powerpc`, `tms320c3x`, `sharc21x61`, `mpc5xx`, `x86_64`, or `mcpu` with `long long` set to 64 (`-long-long-is-64bits` at the command line).
- If you use the option `Check JSF AV C++ rules (-jsf-coding-rules)`, select the compiler `generic`. If you use another compiler, Polyspace cannot check the JSF coding rules that require conforming to the ISO standard. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Limitations

Polyspace does not support certain features of these compilers:

- GNU compilers (version 4.7 or later):
 - Nested functions.

For instance, the function `bar` is nested in function `foo`:

```
int foo (int a, int b)
{
    int bar (int c) { return c * c; }

    return bar (a) + bar (b);
}
```

- Binary operations with vector types where one operand uses the shorthand notation for uniform vectors.

For instance, in the addition operation, `2+a`, `2` is used as a shorthand notation for `{2,2,2,2}`.

```
typedef int v4si __attribute__((vector_size (16)));
v4si res, a = {1,2,3,4};

res = 2 + a; /* means {2,2,2,2} + a */
```

- Forward declaration of function parameters.

For instance, the parameter `len` is forward declared:

```
void func (int len; char data[len][len], int len)
{
    /* ... */
}
```

- Complex integer data types.

However, complex floating point data types are supported.

- Initialization of structures with flexible array members using an initialization list.

For instance, the structure `S` has a flexible array member `tab`. A variable of type `S` is directly initialized with an initialization list.

```
struct S {
    int x;
    int tab[];          /* flexible array member - not supported */
};
struct S s = { 0, 1, 2} ;
```

You see a warning during analysis and a red check in the results when you dereference, for instance, `s.tab[1]`.

- 128-bit variables.

Polyspace cannot analyze this data type semantically. Bug Finder allows use of 128-bit data types, but Code Prover shows a compilation error if you use such a data type, for instance, the GCC extension `__float128`.

- GNU compilers version 7.x:
 - Type names `_FloatN` and `_FloatNx` are not semantically supported. The analysis treats them as type `float`, `double`, or `long double`.
 - Constants of type `_FloatN` or `_FloatNx` with suffixes `fN`, `FN`, or `fNx`, such as `1.2f123` or `2.3F64x` are not supported.
- Visual Studio compilers:
 - C++ Accelerated Massive Parallelism (AMP).

C++ AMP is a Visual Studio feature that accelerates your C++ code execution for certain types of data-parallel hardware on specific targets. You typically use the `restrict` keyword to enable this feature.

```
void Buffer() restrict(amp)
{
    ...
}
```

- `__assume` statements.

You typically use `__assume` with a condition that is false. The statement indicates that the optimizer must assume the condition to be henceforth true. Code Prover cannot reconcile this contradiction. You get the error:

```
Asked for compulsory presence of absent entity : assert
```

- Managed Extensions for C++ (required for the .NET Framework), or its successor, C++/CLI (C++ modified for Common Language Infrastructure)

- `__declspec` keyword with attributes other than `noreturn`, `nothrow`, `selectany` or `thread`.

Command-Line Information

Parameter: `-compiler`

Value: `generic` | `gnu3.4` | `gnu4.6` | `gnu4.7` | `gnu4.8` | `gnu4.9` | `gnu5.x` | `gnu6.x` | `gnu7.x` | `clang3.x` | `clang4.x` | `clang5.x` | `visual9.0` | `visual10.0` | `visual11.0` | `visual12.0` | `visual14.0` | `visual15.x` | `keil` | `iar` | `armcc` | `armclang` | `codewarrior` | `cosmic` | `diab` | `greenhills` | `iar-ew` | `microchip` | `renesas` | `tasking` | `ti`

Default: `generic`

Example 1 (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c" -compiler gnu4.6`

Example 2 (Bug Finder): `polyspace-bug-finder -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`

Example 1 (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c" -lang c -compiler gnu4.6`

Example 2 (Code Prover): `polyspace-code-prover -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`

Example 1 (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c" -compiler gnu4.6`

Example 2 (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`

Example 1 (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c" -lang c -compiler gnu4.6`

Example 2 (Code Prover Server): `polyspace-code-prover-server -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`

See Also

C standard version (`-c-version`) | C++ standard version (`-cpp-version`) | Target processor type (`-target`)

Topics

"Prepare Scripts for Polyspace Analysis"

"Troubleshoot Compilation Errors"

"Specify Target Environment and Compiler Behavior"

"Supported Keil or IAR Language Extensions"

ARM v5 Compiler (-compiler armcc)

Specify ARM v5 compiler

Description

Specify `armcc` for the `Compiler (-compiler)` option if you compile your code with a ARM v5 compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `armcc` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a ARM v5 compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `armcc` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler armcc -target`

Value: `arm`

Default: `arm`

Example (Bug Finder): `polyspace-bug-finder -compiler armcc -target arm`

Example (Code Prover): `polyspace-code-prover -compiler armcc -target arm`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler armcc -target arm`

Example (Code Prover Server): `polyspace-code-prover-server -compiler armcc -target arm`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2019a

ARM v6 Compiler (-compiler armclang)

Specify ARM v6 compiler

Description

Specify `armclang` for the `Compiler (-compiler)` option if you compile your code with a ARM v6 compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `armclang` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a ARM v6 compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `armclang` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler armclang -target`

Value: `arm | arm64`

Default: `arm`

Example (Bug Finder): `polyspace-bug-finder -compiler armclang -target arm64`

Example (Code Prover): `polyspace-code-prover -compiler armclang -target arm64`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler armclang -target arm64`

Example (Code Prover Server): `polyspace-code-prover-server -compiler armclang -target arm64`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2019a

NXP CodeWarrior Compiler (-compiler codewarrior)

Specify NXP CodeWarrior compiler

Description

Specify `codewarrior` for `Compiler (-compiler)` if you compile your code using a NXP CodeWarrior compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `codewarrior` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a NXP CodeWarrior compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `codewarrior` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler codewarrior -target`

Value: `s12z | powerpc`

Default: `s12z`

Example (Bug Finder): `polyspace-bug-finder -compiler codewarrior -target powerpc`

Example (Code Prover): `polyspace-code-prover -compiler codewarrior -target powerpc`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler codewarrior -target powerpc`

Example (Code Prover Server): `polyspace-code-prover-server -compiler codewarrior -target powerpc`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2018a

Cosmic Compiler (-compiler cosmic)

Specify Cosmic compiler

Description

Specify `cosmic` for the `Compiler (-compiler)` option if you compile your code with a Cosmic compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `cosmic` for **Compiler**, in the user interface, you see only the processors allowed for a Cosmic compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `cosmic` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the target uses, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler cosmic -target`

Value: `s12z`

Default: `s12z`

Example (Bug Finder): `polyspace-bug-finder -compiler cosmic -target s12z`

Example (Code Prover): `polyspace-code-prover -compiler cosmic -target s12z`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler cosmic -target s12z`

Example (Code Prover Server): `polyspace-code-prover-server -compiler cosmic -target s12z`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2019b

Diab Compiler (-compiler diab)

Specify the Wind River Diab compiler

Description

Specify `diab` for `Compiler (-compiler)` if you compile your code using the Wind River Diab compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `diab` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for the Diab compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `diab` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

The software supports version 5.9.6 and older versions of the Diab compiler.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

If you encounter errors during Polyspace analysis, see “Errors Related to Diab Compiler”.

Command-Line Information

Parameter: `-compiler diab -target`

Value: `i386 | powerpc | arm | coldfire | mips | mcore | rh850 | superh | tricore`

Default: `powerpc`

Example (Bug Finder): `polyspace-bug-finder -compiler diab -target tricore`

Example (Code Prover): `polyspace-code-prover -compiler diab -target tricore`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler diab -target tricore`

Example (Code Prover Server): `polyspace-code-prover-server -compiler diab -target tricore`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2016b

Green Hills Compiler (-compiler greenhills)

Specify Green Hills compiler

Description

Specify `greenhills` for `Compiler (-compiler)` if you compile your code using a Green Hills compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `greenhills` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Green Hills compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `greenhills` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

- If you encounter errors during a Polyspace analysis, see “Errors Related to Green Hills Compiler”
- Polyspace supports the embedded configuration for the i386 target. If your x86 Green Hills compiler is configured for native Windows development, you can see compilation errors or incorrect analysis results with Code Prover. Contact Technical Support.

For instance, Green Hills compilers consider a size of 12 bytes for `long double` for embedded targets, but 8 bytes for native Windows. Polyspace considers 12 bytes by default.

- If you create a Polyspace project from a build command that uses a Green Hills compiler, the compiler options `-filetype` and `-os_dir` are not implemented in the project. To emulate the `-os_dir` option, you can explicitly add the path argument of the option as an include folder to your Polyspace project.

Command-Line Information

Parameter: `-compiler greenhills -target`

Value: `powerpc | powerpc64 | arm | arm64 | tricore | rh850 | arm | i386 | x86_64`

Default: `powerpc`

Example (Bug Finder): polyspace-bug-finder -compiler greenhills -target arm

Example (Code Prover): polyspace-code-prover -compiler greenhills -target arm

Example (Bug Finder Server): polyspace-bug-finder-server -compiler greenhills -target arm

Example (Code Prover Server): polyspace-code-prover-server -compiler greenhills -target arm

See Also

Compiler (-compiler) | Target processor type (-target)

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2017b

IAR Embedded Workbench Compiler (-compiler iar-ew)

Specify IAR Embedded Workbench compiler

Description

Specify `iar-ew` for **Compiler** (-compiler) if you compile your code using a IAR Embedded Workbench compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `iar-ew` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a IAR Embedded Workbench compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `iar-ew` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

Polyspace does not support some constructs specific to the IAR compiler.

For the list of unsupported constructs, see `codeprover_limitations.pdf` in `polyspaceroot` \polyspace\verifier\code_prover_desktop. Here, `polyspaceroot` is the MATLAB installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Command-Line Information

Parameter: `-compiler iar-ew -target`

Value: `arm | avr | msp430 | rh850 | rl78`

Default: `arm`

Example (Bug Finder): `polyspace-bug-finder -compiler iar-ew -target rl78`

Example (Code Prover): `polyspace-code-prover -compiler iar-ew -target rl78`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler iar-ew -target rl78`

Example (Code Prover Server): `polyspace-code-prover-server -compiler iar-ew -target rl78`

See Also

Compiler (-compiler) | Target processor type (-target)

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2018a

MPLAB XC8 C Compiler (-compiler microchip)

Specify MPLAB XC8 C compiler

Description

Specify `microchip` for the `Compiler` (-compiler) option if you compile your code with a MPLAB XC8 C compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `microchip` for **Compiler**, in the user interface, you see only the processors allowed for a MPLAB XC8 C compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `microchip` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the target uses, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tip

Polyspace does not support the Atmel families of processors, such as AVR, TinyAVR, MegaAVR, XMEGA, and SAM32.

Command-Line Information

Parameter: `-compiler microchip -target`

Value: `pic`

Default: `pic`

Example (Bug Finder): `polyspace-bug-finder -compiler microchip -target pic`

Example (Code Prover): `polyspace-code-prover -compiler microchip -target pic`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler microchip -target pic`

Example (Code Prover Server): `polyspace-code-prover-server -compiler microchip -target pic`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2020a

Renesas Compiler (-compiler renesas)

Specify Renesas compiler

Description

Specify `renesas` for the `Compiler (-compiler)` option if you compile your code with a Renesas compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `renesas` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Renesas compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `renesas` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler renesas -target`

Value: `rl78 | rh850 | rx`

Default: `rl78`

Example (Bug Finder): `polyspace-bug-finder -compiler renesas -target rx`

Example (Code Prover): `polyspace-code-prover -compiler renesas -target rx`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler renesas -target rx`

Example (Code Prover Server): `polyspace-code-prover-server -compiler renesas -target rx`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2018b

TASKING Compiler (-compiler tasking)

Specify the Altium TASKING compiler

Description

Specify tasking for Compiler (-compiler) if you compile your code using the Altium® TASKING compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select tasking for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for the TASKING compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the tasking compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

The software supports different versions of the TASKING compiler, depending on the target:

- TriCore: 6.0 and older versions
- C166: 4.0 and older versions
- ARM: 5.2 and older versions
- RH850: 2.2 and older versions

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

- Polyspace does not support some constructs specific to the TASKING compiler.

For the list of unsupported constructs, see `codeprover_limitations.pdf` in `polyspaceroot\polyspace\verifier\code_prover_desktop`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

- The CPU used is TC1793. If you use a different CPU, set the following analysis options in your project:
 - `Disabled preprocessor definitions (-U)`: Undefine the macro `__CPU_TC1793B__`.
 - `Preprocessor definitions (-D)`: Define the macro `__CPU__`. Enter `__CPU__=xxx`, where `xxx` is the name of your CPU.

Additionally, define the equivalent of the macro `__CPU_TC1793B__` for your CPU. For instance, enter `__CPU_TC1793A__`.

Instead of manually specifying your compiler, if you trace your build command (makefile), Polyspace can detect your CPU and add the required definitions in your project.

- For some errors related to TASKING compiler-specific constructs, see solutions in “Errors Related to TASKING Compiler”.

Command-Line Information

Parameter: `-compiler tasking -target`

Value: `tricore | c166 | rh850 | arm`

Default: `tricore`

Example (Bug Finder): `polyspace-bug-finder -compiler tasking -target tricore`

Example (Code Prover): `polyspace-code-prover -compiler tasking -target tricore`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler tasking -target tricore`

Example (Code Prover Server): `polyspace-code-prover-server -compiler tasking -target tricore`

See Also

Compiler (-compiler) | Target processor type (-target)

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2017a

Texas Instruments Compiler (-compiler ti)

Specify Texas Instruments compiler

Description

Specify `ti` for `Compiler (-compiler)` if you compile your code using a Texas Instruments compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `ti` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Texas Instruments compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `ti` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

Polyspace does not support some constructs specific to the Texas Instruments compiler.

For the list of unsupported constructs, see `codeprover_limitations.pdf` in `polyspaceroot\polyspace\verifier\code_prover_desktop`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Command-Line Information

Parameter: `-compiler ti -target`

Value: `c28x | c6000 | arm | msp430`

Default: `c28x`

Example (Bug Finder): `polyspace-bug-finder -compiler ti -target msp430`

Example (Code Prover): `polyspace-code-prover -compiler ti -target msp430`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler ti -target msp430`

Example (Code Prover Server): `polyspace-code-prover-server -compiler ti -target msp430`

See Also

Compiler (-compiler) | Target processor type (-target)

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Introduced in R2018a

Sfr type support (-sfr-types)

Specify sizes of sfr types for code developed with Keil or IAR compilers

Description

Specify sizes of sfr types (types that define special function registers).

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependency” on page 2-42 for other options you must also enable.

Command line: Use the option -sfr-types. See “Command-Line Information” on page 2-42.

Why Use This Option

Use this option if you have statements such as `sfr addr = 0x80;` in your code. sfr types are not standard C types. Therefore, you must specify their sizes explicitly for the Polyspace analysis.

Settings

No Default

List each sfr name and its size in bits.

Dependency

This option is available only when **Compiler** (-compiler) is set to keil or iar.

Command-Line Information

Syntax: -sfr-types *sfr_name=size_in_bits,...*

No Default

Name Value: an sfr name such as sfr16.

Size Value: 8 | 16 | 32

Example (Bug Finder): polyspace-bug-finder -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...

Example (Code Prover): polyspace-code-prover -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...

Example (Bug Finder Server): polyspace-bug-finder-server -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...

Example (Code Prover Server): polyspace-code-prover-server -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

“Supported Keil or IAR Language Extensions”

Division round down (-div-round-down)

Round down quotients from division or modulus of negative numbers instead of rounding up

Description

Specify whether quotients from division and modulus of negative numbers are rounded up or down.

Note $a = (a / b) * b + a \% b$ is always true.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-div-round-down`. See “Command-Line Information” on page 2-45.

Why Use This Option

Use this option to emulate your compiler.

The option is relevant only for compilers following C90 standard (ISO/IEC 9899:1990). The standard stipulates that *"if either operand of / or % is negative, whether the result of the / operator, is the largest integer less or equal than the algebraic quotient or the smallest integer greater or equal than the quotient, is implementation defined, same for the sign of the % operator"*. The standard allows compilers to choose their own implementation.

For compilers following the C99 standard ((ISO/IEC 9899:1999), this option is not required. The standard enforces division with rounding towards zero (section 6.5.5).

Settings

On

If either operand `/` or `%` is negative, the result of the `/` operator is the largest integer less than or equal to the algebraic quotient. The result of the `%` operator is deduced from $a \% b = a - (a / b) * b$.

Example: `assert(-5/3 == -2 && -5%3 == 1);` is true.

Off (default)

If either operand of `/` or `%` is negative, the result of the `/` operator is the smallest integer greater than or equal to the algebraic quotient. The result of the `%` operator is deduced from $a \% b = a - (a / b) * b$.

This behavior is also known as rounding towards zero.

Example: `assert(-5/3 == -1 && -5%3 == -2);` is true.

Command-Line Information

Parameter: -div-round-down

Default: Off

Example (Bug Finder): polyspace-bug-finder -div-round-down

Example (Code Prover): polyspace-code-prover -div-round-down

Example (Bug Finder Server): polyspace-bug-finder-server -div-round-down

Example (Code Prover Server): polyspace-code-prover-server -div-round-down

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Enum type definition (-enum-type-definition)

Specify how to represent an enum with a base type

Description

Allow the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition. When using this option, each enum type is represented by the smallest integral type that can hold its enumeration values.

This option is available on the **Target & Compiler** node in the **Configuration** pane.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-enum-type-definition`. See “Command-Line Information” on page 2-47.

Why Use This Option

Your compiler represents enum variables as constants of a base integer type. Use this option so that you can emulate your compiler.

To check your compiler settings:

- 1 Compile this code using the compiler settings that you typically use:

```
enum { MAXSIGNEDBYTE=127 } mysmallenum_t;

int dummy[(int)sizeof(mysmallenum_t) - (int)sizeof(int)];
```

If compilation fails, you have to use one of `auto-signed-first` or `auto-unsigned-first`.

- 2 Compile this code using the compiler settings that you typically use:

```
#include <limits.h>

enum { MYINTMAX = INT_MAX } myintenum_t;

int dummy[(MYINTMAX + 1) < 0 ? -1:1];
```

If compilation fails, use `auto-signed-first` for this option, otherwise use `auto-unsigned-first`.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Uses the signed integer type for all compilers except gnu, clang and tasking.

For the gnu and clang compilers, it uses the first type that can hold all of the enumerator values from this list: unsigned int, signed int, unsigned long, signed long, unsigned long long and signed long long.

For the tasking compiler, it uses the first type that can hold all of the enumerator values from this list: char, unsigned char, short, unsigned short, int, and unsigned int.

auto-signed-first

Uses the first type that can hold all of the enumerator values from this list: signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long, unsigned long, signed long long, and unsigned long long.

auto-unsigned-first

Uses the first type that can hold all of the enumerator values from these lists:

- If enumerator values are positive: unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long.
- If one or more enumerator values are negative: signed char, signed short, signed int, signed long, and signed long long.

Command-Line Information

Parameter: -enum-type-definition

Value: defined-by-compiler | auto-signed-first | auto-unsigned-first

Default: defined-by-compiler

Example (Bug Finder): polyspace-bug-finder -enum-type-definition auto-signed-first

Example (Code Prover): polyspace-code-prover -enum-type-definition auto-signed-first

Example (Bug Finder Server): polyspace-bug-finder-server -enum-type-definition auto-signed-first

Example (Code Prover Server): polyspace-code-prover-server -enum-type-definition auto-signed-first

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Block char16/32_t types (-no-uliterals)

Disable Polyspace definitions for char16_t or char32_t

Description

Specify that the analysis must not define char16_t or char32_t types.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 2-48 for other options you must also enable.

Command line: Use the option -no-uliterals. See “Command-Line Information” on page 2-48.

Why Use This Option

If your compiler defines char16_t and/or char32_t through a typedef statement or by using includes, use this option to turn off the standard Polyspace definition of char16_t and char32_t.

To check if your compiler defines these types, compile this code using the compiler settings that you typically use:

```
typedef unsigned short char16_t;  
typedef unsigned long char32_t;
```

If the file compiles, it means that your compiler has already defined char16_t and char32_t. Enable this Polyspace option.

Settings

On

The analysis does not allow char16_t and char32_t types.

Off (default)

The analysis allows char16_t and char32_t types.

Dependencies

You can select this option only when these conditions are true:

- Source code language (-lang) is set to CPP or C-CPP.
- Compiler (-compiler) is set to generic or a gnu version.

Command-Line Information

Parameter: -no-uliterals

Default: off

Example (Bug Finder): polyspace-bug-finder -lang cpp -compiler gnu4.7 -cpp-version cppl1 -no-uliterals

Example (Code Prover): polyspace-code-prover -compiler gnu4.7 -lang cpp -cpp-version cpp11 -no-uliterals

Example (Bug Finder Server): polyspace-bug-finder-server -lang cpp -compiler gnu4.7 -cpp-version cpp11 -no-uliterals

Example (Code Prover Server): polyspace-code-prover-server -compiler gnu4.7 -lang cpp -cpp-version cpp11 -no-uliterals

See Also

Compiler (-compiler)

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Pack alignment value (-pack-alignment-value)

Specify default structure packing alignment for code developed in Visual C++

Description

Specify the default packing alignment (in bytes) for structures, unions, and class members.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-pack-alignment-value`. See “Command-Line Information” on page 2-50.

Why Use This Option

If you use compiler options to specify how members of a structure are packed into memory, use this option to emulate your compiler.

For instance, if you use the Visual Studio option `/Zp` to specify an alignment, use this option for your Polyspace analysis.

If you use `#pragma pack` directives in your code to specify alignment, and also specify this option for analysis, the `#pragma pack` directives take precedence.

Settings

Default: 8

You can enter one of these values:

- 1
- 2
- 4
- 8
- 16

Command-Line Information

Parameter: `-pack-alignment-value`

Value: 1 | 2 | 4 | 8 | 16

Default: 8

Example (Bug Finder): `polyspace-bug-finder -compiler visual10 -pack-alignment-value 4`

Example (Code Prover): `polyspace-code-prover -compiler visual10 -pack-alignment-value 4`

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler visual10 -pack-alignment-value 4`

Example (Code Prover Server): polyspace-code-prover-server -compiler visual10 -
pack-alignment-value 4

See Also

Ignore pragma pack directives (-ignore-pragma-pack)

Ignore #pragma pack directives

Description

Specify that the analysis must ignore #pragma pack directives in the code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option -ignore-pragma-pack. See “Command-Line Information” on page 2-52.

Why Use This Option

Use this option if #pragma pack directives in your code cause linking errors.

For instance, you have two structures with the same name in your code, but one declaration follows a #pragma pack(2) statement. Because the default alignment is 8 bytes, the different packing for the two structures causes a linking error. Use this option to avoid such errors.

Settings

On

The analysis ignores the #pragma directives.

Off (default)

The analysis takes into account specifications in the #pragma directives.

Command-Line Information

Parameter: -ignore-pragma-pack

Default: Off

Example (Bug Finder): polyspace-bug-finder -ignore-pragma-pack

Example (Code Prover): polyspace-code-prover -ignore-pragma-pack

Example (Bug Finder Server): polyspace-bug-finder-server -ignore-pragma-pack

Example (Code Prover Server): polyspace-code-prover-server -ignore-pragma-pack

See Also

Management of `size_t` (-size-t-type-is)

Specify the underlying data type of `size_t`

Description

Specify the underlying data type of `size_t` explicitly: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` or `unsigned long long`. If you do not specify this option, your choice of compiler determines the underlying type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-size-t-type-is`. See “Command-Line Information” on page 2-54.

Why Use This Option

The analysis associates a data type with `size_t` when you specify your compiler. If you use a compiler option that changes this default type, emulate your compiler option by using this analysis option.

If you run into compilation errors during Polyspace analysis and trace the error to the definition of `size_t`, it is possible that you use a compiler option and change your compiler default. To probe further, compile this code with your compiler using the options that you typically use:

```
/* Header defines malloc as void* malloc (size_t size)
#include <stdio.h>

void* malloc (unsigned int size);
```

If the file does not compile, your compiler (along with compiler options) defines `size_t` using a different underlying type. Replace `unsigned int` with another type such as `unsigned long` and try again.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Your specification for **Compiler** (`-compiler`) determines the underlying type of `size_t`.

`unsigned-int`

The analysis considers `unsigned int` as the underlying type of `size_t`.

`unsigned-long`

The analysis considers `unsigned long` as the underlying type of `size_t`.

`unsigned-long-long`

The analysis considers `unsigned long long` as the underlying type of `size_t`.

Command-Line Information

Parameter: `-size-t-type-is`

Value: `defined-by-compiler | unsigned-char | unsigned-int | unsigned-short | unsigned-long | unsigned-long-long`

Default: `defined-by-compiler`

Example (Bug Finder): `polyspace-bug-finder -size-t-type-is unsigned-long`

Example (Code Prover): `polyspace-code-prover -size-t-type-is unsigned-long`

Example (Bug Finder Server): `polyspace-bug-finder-server -size-t-type-is unsigned-long`

Example (Code Prover Server): `polyspace-code-prover-server -size-t-type-is unsigned-long`

See Also

`-custom-target`

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Management of `wchar_t` (-wchar-t-type-is)

Specify the underlying data type of `wchar_t`

Description

Specify the underlying data type of `wchar_t` explicitly. If you do not specify this option, your choice of compiler determines the underlying type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-wchar-t-type-is`. See “Command-Line Information” on page 2-55.

Why Use This Option

The analysis associates a data type with `wchar_t` when you specify your compiler. If you use a compiler option that changes this default type, emulate your compiler option by using this analysis option.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Your specification for `Compiler` (`-compiler`) determines the underlying type of `wchar_t`.

`signed-short`

The analysis considers `signed short` as the underlying type of `wchar_t`.

`unsigned-short`

The analysis considers `unsigned short` as the underlying type of `wchar_t`.

`signed-int`

The analysis considers `signed int` as the underlying type of `wchar_t`.

`unsigned-int`

The analysis considers `unsigned int` as the underlying type of `wchar_t`.

`signed-long`

The analysis considers `signed long` as the underlying type of `wchar_t`.

`unsigned-long`

The analysis considers `unsigned long` as the underlying type of `wchar_t`.

Command-Line Information

Parameter: `-wchar-t-type-is`

Value: defined-by-compiler | signed-short | unsigned-short | signed-int | unsigned-int | signed-long | unsigned-long

Default: defined-by-compiler

Example (Bug Finder): polyspace-bug-finder -wchar-t-type-is signed-int

Example (Code Prover): polyspace-code-prover -wchar-t-type-is signed-int

Example (Bug Finder Server): polyspace-bug-finder-server -wchar-t-type-is signed-int

Example (Code Prover Server): polyspace-code-prover-server -wchar-t-type-is signed-int

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Signed right shift (-logical-signed-right-shift)

Specify how to treat the sign bit for logical right shifts on signed variables

Description

Choose between arithmetic and logical shift for right shift operations on negative values.

This option does not modify compile-time expressions. For more details, see “Limitation” on page 2-57.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-logical-signed-right-shift`. See “Command-Line Information” on page 2-58.

Why Use This Option

The C99 Standard (sec 6.5.7) states that for a right-shift operation $x1 \gg x2$, if $x1$ is signed and has negative values, the behavior is implementation-defined. Different compilers choose between arithmetic and logical shift. Use this option to emulate your compiler.

Settings

Default: `Arithmetical`

Arithmetical

The sign bit remains:

```
(-4) >> 1 = -2
(-7) >> 1 = -4
 7 >> 1 = 3
```

Logical

0 replaces the sign bit:

```
(-4) >> 1 = (-4U) >> 1 = 2147483646
(-7) >> 1 = (-7U) >> 1 = 2147483644
 7 >> 1 = 3
```

Limitation

In compile-time expressions, this Polyspace option does not change the standard behavior for right shifts.

For example, consider this right shift expression:

```
int arr[ ((-4) >> 20) ];
```

The compiler computes array sizes, so the expression `(-4) >> 20` is evaluated at compilation time. Logically, this expression is equivalent to 4095. However, arithmetically, the result is -1. This statement causes a compilation error (arrays cannot have negative size) because the standard right-shift behavior for signed integers is arithmetic.

Command-Line Information

When using the command line, arithmetic is the default computation mode. When this option is set, logical computation is performed.

Parameter: `-logical-signed-right-shift`

Default: Arithmetic signed right shifts

Example (Bug Finder): `polyspace-bug-finder -logical-signed-right-shift`

Example (Code Prover): `polyspace-code-prover -logical-signed-right-shift`

Example (Bug Finder Server): `polyspace-bug-finder-server -logical-signed-right-shift`

Example (Code Prover Server): `polyspace-code-prover-server -logical-signed-right-shift`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify Target Environment and Compiler Behavior”

Preprocessor definitions (-D)

Replace macros in preprocessed code

Description

Replace macros with their definitions in preprocessed code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Macros** node.

Command line: Use the option `-D`. See “Command-Line Information” on page 2-60.

Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro `_WIN32` as defined when you build your code, it executes code in a `#ifdef _WIN32` statement. If Polyspace does not consider that macro as defined, you must use this option to replace the macro with 1.

Depending on your settings for **Compiler (-compiler)**, some macros are defined by default. Use this option to define macros that are not implicitly defined.

Typically, you recognize from compilation errors that a certain macro is not defined. For instance, the following code does not compile if the macro `_WIN32` is not defined.


```
#ifdef _WIN32
    int env_var;
#endif

void set() {
    env_var=1;
}
```

The error message states that `env_var` is undefined. However, the definition of `env_var` is in the `#ifdef _WIN32` statement. The underlying cause for the error is that the macro `_WIN32` is not defined. You must define `_WIN32`.

Settings

No Default

Using the  button, add a row for the macro you want to define. The definition must be in the format `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank.

For example:

- `name1=name2` replaces all instances of `name1` by `name2`.
- `name=` instructs the software to ignore `name`.

- name with no equals sign or value replaces all instances of name by 1. To define a macro to execute code in a `#ifdef macro_name` statement, use this syntax.

Tips

- If Polyspace does not support a non-ANSI keyword and shows a compilation error, use this option to replace all occurrences of the keyword with a blank string in preprocessed code. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

For instance, if your compiler supports the `__far` keyword, to avoid compilation errors:

- In the user interface (desktop products only), enter `__far=`.
- On the command line, use the flag `-D __far=`.

The software replaces the `__far` keyword with a blank string during preprocessing. For example:

```
int __far* pValue;
```

is converted to:

```
int * pValue;
```

- Polyspace recognizes keywords such as `restrict` and does not allow their use as identifiers. If you use those keywords as identifiers (because your compiler does not recognize them as keywords), replace the disallowed name with another name using this option. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

For instance, to allow use of `restrict` as identifier:

- In the user interface, enter `restrict=my_restrict`.
- On the command line, use the flag `-D restrict=my_restrict`.
- Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.
 - To override the macro definition coming from a compiler specification, use this option.
 - To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

You can specify only one flag with each `-D` option. However, you can specify the option multiple times.

Parameter: `-D`

No Default

Value: *flag=value*

Example (Bug Finder): `polyspace-bug-finder -D HAVE_MYLIB -D int32_t=int`

Example (Code Prover): `polyspace-code-prover -D HAVE_MYLIB -D int32_t=int`

Example (Bug Finder Server): `polyspace-bug-finder-server -D HAVE_MYLIB -D int32_t=int`

Example (Code Prover Server): `polyspace-code-prover-server -D HAVE_MYLIB -D int32_t=int`

See Also

`Disabled preprocessor definitions (-U)`

Topics

“Prepare Scripts for Polyspace Analysis”

Disabled preprocessor definitions (-U)

Undefine macros in preprocessed code

Description

Undefine macros in preprocessed code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Macros** node.

Command line: Use the option -U. See “Command-Line Information” on page 2-63.

Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro `_WIN32` as undefined when you build your code, it executes code in a `#ifndef _WIN32` statement. If Polyspace considers that macro as defined, you must explicitly undefine the macro.

Some settings for `Compiler (-compiler)` enable certain macros by default. This option allows you undefine the macros.

Typically, you recognize from compilation errors that a certain macro must be undefined. For instance, the following code does not compile if the macro `_WIN32` is defined.


```
#ifndef _WIN32
    int env_var;
#endif

void set() {
    env_var=1;
}
```

The error message states that `env_var` is undefined. However, the definition of `env_var` is in the `#ifndef _WIN32` statement. The underlying cause for the error is that the macro `_WIN32` is defined. You must undefine `_WIN32`.

Settings

No Default

Using the  button, add a new row for each macro being undefined.

Tips

Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override a macro definition coming from a compiler specification, use the option `Preprocessor definitions (-D)`.

- To undefine the macro, use this option.

Command-Line Information

You can specify only one flag with each -U option. However, you can specify the option multiple times.

Parameter: -U

No Default

Value: *macro*

Example (Bug Finder): polyspace-bug-finder -U HAVE_MYLIB -U USE_COM1

Example (Code Prover): polyspace-code-prover -U HAVE_MYLIB -U USE_COM1

Example (Bug Finder Server): polyspace-bug-finder-server -U HAVE_MYLIB -U USE_COM1

Example (Code Prover Server): polyspace-code-prover-server -U HAVE_MYLIB -U USE_COM1

See Also

Preprocessor definitions (-D)

Topics

“Prepare Scripts for Polyspace Analysis”

Source code encoding (-sources-encoding)

Specify the encoding that the analysis uses to interpret non-ASCII characters in source code

Description

Specify the encoding of your source files. The analysis uses this information to interpret non-ASCII characters in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-sources-encoding`. See "Command-Line Information" on page 2-65.

Why Use This Option

If your source code contains non-ASCII characters, for instance, Japanese or Korean characters, the Polyspace analysis can interpret the characters and later display the source code correctly.

If you still have compilation errors or display issues from non-ASCII characters, you can explicitly specify your source code encoding using this option.

Settings

Default: system

system

The analysis uses the default encoding of the operating system.

shift-jis

The analysis uses the Shift JIS (Shift Japanese Industrial Standards) encoding, a character encoding for the Japanese language.

iso-8859-1

The analysis uses the ISO/IEC 8859-1:1998 encoding, a character encoding that encodes what it refers to as "Latin alphabet no.1", consisting of 191 characters from the Latin script.

windows-1252

The analysis uses the Windows-1252 encoding, a single-byte character encoding of the Latin alphabet, used by default in the legacy components of Windows for English and some other Western languages.

UTF-8

The analysis uses the UTF-8 encoding, a variable width character encoding capable of encoding all valid code points in Unicode.

Polyspace supports many more encodings. To specify an encoding that is not in the above list in the Polyspace user interface, enter `-sources-encoding encodingname` in the **Other** field. In particular, if your source files contain a mix of different encodings, you can use `-sources-encoding`

auto. In this mode, the analysis uses internal heuristics to determine the encoding of your source files from their contents.

For the full list of supported encodings, at the command line, enter:

```
-list-all-values -sources-encoding
```

with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command. Pipe the output to a file and search the file for the encoding that you are using.

Command-Line Information

Parameter: -sources-encoding

Default: system

Value: auto | system | shift-jis | iso-8859-1 | windows-1252 | UTF-8

Example (Bug Finder): `polyspace-bug-finder -sources-encoding windows-1252`

Example (Code Prover): `polyspace-code-prover -sources-encoding windows-1252`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources-encoding windows-1252`

Example (Code Prover Server): `polyspace-code-prover-server -sources-encoding windows-1252`

Polyspace supports many more encodings besides the above list. For the full list of supported encodings, at the command line, enter:

```
-list-all-values -sources-encoding
```

with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command. Pipe the output to a file and search the file for the encoding that you are using.

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Code from DOS or Windows file system (-dos)

Consider that file paths are in MS-DOS style

Description

Specify that DOS or Windows files are provided for analysis.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-dos`. See “Command-Line Information” on page 2-66.

Why Use This Option

Use this option if the contents of the **Include** or **Source** folder come from a DOS or Windows file system. The option helps you resolve case sensitivity and control character issues.

Settings

On (default)

Analysis understands file names and include paths for Windows/DOS files

For example, with this option,

```
#include "..\mY_TEst.h"^M
#include "..\mY_other_FILE.H"^M
```

resolves to:

```
#include "../my_test.h"
#include "../my_other_file.h"
```

In this mode, you see an error if your include folder has header files whose names differ only in case.

Off

Characters are not controlled for files names or paths.

Command-Line Information

Parameter: `-dos`

Default: Off

Example (Bug Finder): `polyspace-bug-finder -dos -I ./my_copied_include_dir -D test=1`

Example (Code Prover): `polyspace-code-prover -dos -I ./my_copied_include_dir -D test=1`

Example (Bug Finder Server): polyspace-bug-finder-server -dos -I ./
my_copied_include_dir -D test=1

Example (Code Prover Server): polyspace-code-prover-server -dos -I ./
my_copied_include_dir -D test=1

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Stop analysis if a file does not compile (-stop-if-compile-error)

Specify that a compilation error must stop the analysis

Description

Specify that even a single compilation error must stop the analysis.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Environment Settings** node.

Command line: Use the option `-stop-if-compile-error`. See “Command-Line Information” on page 2-69.

Why Use This Option

Use this option to first resolve all compilation errors and then perform the Polyspace analysis. This sequence ensures that all files are analyzed.

Otherwise, only files without compilation errors are fully analyzed. The analysis might return some results for files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. This assumption can sometimes make the analysis less precise.







The option is more useful for a Code Prover analysis because the Code Prover run-time checks rely more heavily on range propagation across functions.

Settings

On

The analysis stops even if a single compilation error occurs.

In the user interface of the Polyspace desktop products, you see the compilation errors on the **Output Summary** pane.

Type	Message	File	Line	Col
	C verification starts at Thu Dec 17 22:26:17 2015			
	6 core(s) detected but the verification uses 4 core(s).			
	identifier "x" is undefined	my_file.c	1	
	Failed compilation.	my_file.c		
	Verifier has detected compilation error(s) in the code.			
	Exiting because of previous error			

For information on how to resolve the errors, see “Troubleshoot Compilation Errors”.

You can also see the errors in the analysis log, a text file generated during the analysis. The log is named `Polyspace_R20##n_ProjectName_date-time.log` and contains lines starting with `Error:` indicating compilation errors. To view the log from the analysis results:

- In the user interface of the Polyspace desktop products, select **Window > Show/Hide View > Run Log**.
- In the Polyspace Access web interface, open the **Review** tab. Select **Layout > Show/Hide View > Run Log**.

Despite compilation errors, you can see some analysis results, for instance, coding rule violations.

Off (default)

The analysis does not stop because of compilation errors, but only files without compilation errors are analyzed. The analysis does not consider files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. If the analysis needs the definition of such a function, it makes broad assumptions about the function.

- The function return value can take any value in the range allowed by its data type.
- The function can modify arguments passed by reference so that they can take any value in the range allowed by their data types.

If the assumptions are too broad, the analysis can be less precise. For instance, a run-time check can flag an operation in orange even though it does not fail in practice.

If compilation errors occur, in the user interface of the Polyspace desktop products, the **Dashboard** pane has a link, which shows that some files failed to compile. You can click the link and see the compilation errors on the **Output Summary** pane.

You can also see the errors in the analysis log, a text file generated during the analysis. The log is named `Polyspace_R20##n_ProjectName_date-time.log` and contains lines starting with `Error:` indicating compilation errors. To view the log from the analysis results:

- In the user interface of the Polyspace desktop products, select **Window > Show/Hide View > Run Log**.
- In the Polyspace Access web interface, open the **Review** tab. Select **Layout > Show/Hide View > Run Log**.

Command-Line Information

Parameter: `-stop-if-compile-error`

Default: Off

Example (Bug Finder): `polyspace-bug-finder -sources filename -stop-if-compile-error`

Example (Code Prover): `polyspace-code-prover -sources filename -stop-if-compile-error`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources filename -stop-if-compile-error`

Example (Code Prover Server): `polyspace-code-prover-server -sources filename -stop-if-compile-error`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2017a

Command/script to apply to preprocessed files (-post-preprocessing-command)

Specify command or script to run on source files after preprocessing phase of analysis

Description

Specify a command or script to run on each source file after preprocessing.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-post-preprocessing-command`. See “Command-Line Information” on page 2-73.

Why Use This Option

You can run scripts on preprocessed files to work around compilation errors or imprecisions of the analysis while keeping your original source files untouched. For instance, suppose Polyspace does not recognize a compiler-specific keyword. If you are certain that the keyword is not relevant for the analysis, you can run a Perl script to remove all instances of the keyword. When you use this option, the software removes the keyword from your preprocessed code but keeps your original code untouched.

Use a script only if the existing analysis options do not meet your requirements. For instance:

- For direct replacement of one keyword with another, use the option `Preprocessor definitions (-D)`.

However, the option does not allow search and replacement involving regular expressions. For regular expressions, use a script.


- For mapping your library function to a standard library function, use the option `-code-behavior-specifications`.

However, the option supports mapping to only a subset of standard library functions. To map to an unsupported function, use a script.

If you are unsure about removing or replacing an unsupported construct, do not use this option. Contact MathWorks Support for guidance.

Settings

No Default

Enter full path to the command or script or click  to navigate to the location of the command or script. This script is executed before verification.

Tips

- Your script must be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.
- Your script must preserve the number of lines in the preprocessed file. In other words, it must not add or remove entire lines to or from the file.

Adding a line or removing one can potentially result in some unpredictable behavior on the location of checks and macros in the Polyspace user interface.

- For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script.

For example:

- To specify a Perl command that replaces all instances of the `far` keyword, enter `polyspaceroot\sys\perl\win32\bin\perl.exe -p -e "s/far//g"`.
- To specify a Perl script `replace_keyword.pl` that replaces all instances of a keyword, enter `polyspaceroot\sys\perl\win32\bin\perl.exe absolute_path \replace_keyword.pl`.

Here, `polyspaceroot` is the location of the current Polyspace installation such as `C:\Program Files\Polyspace\R2019a\` and `absolute_path` is the location of the Perl script. If the paths contain spaces, use quotes to enclose the full path names.

- Use this Perl script as template. The script removes all instances of the `far` keyword.

```
#!/usr/bin/perl

binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{

    # Remove far keyword
    $line =~ s/far//g;

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

You can use Perl regular expressions to perform substitutions. For instance, you can use the following expressions.

Expression	Meaning
.	Matches any single character except newline
[a-z0-9]	Matches any single letter in the set a - z, or digit in the set 0 - 9
[^a-e]	Matches any single letter not in the set a - e
\d	Matches any single digit
\w	Matches any single alphanumeric character or <code>_</code>
x?	Matches 0 or 1 occurrence of x

Expression	Meaning
x*	Matches 0 or more occurrences of x
x+	Matches 1 or more occurrences of x

For complete list of regular expressions, see Perl documentation.

- When you specify this option, the Compilation Assistant is automatically disabled.

Command-Line Information

Parameter: -post-preprocessing-command

Value: Path to executable file or command in quotes

No Default

Example in Linux (Bug Finder): polyspace-bug-finder -sources *file_name* -post-preprocessing-command `pwd`/replace_keyword.pl

Example in Linux (Code Prover): polyspace-code-prover -sources *file_name* -post-preprocessing-command `pwd`/replace_keyword.pl

Example in Linux (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -post-preprocessing-command `pwd`/replace_keyword.pl

Example in Linux (Code Prover Server): polyspace-code-prover-server -sources *file_name* -post-preprocessing-command `pwd`/replace_keyword.pl

Example in Windows: polyspace-bug-finder -sources *file_name* -post-preprocessing-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"

Note that in Windows, you use the full path to the Perl executable.

See Also

-regex-replace-rgx -regex-replace-fmt | Command/script to apply after the end of the code verification (-post-analysis-command)

Topics

"Prepare Scripts for Polyspace Analysis"

"Remove or Replace Keywords Before Compilation"

Include (-include)

Specify files to be #include-ed by each C file in analysis

Description

Specify files to be #include-ed by each C file involved in the analysis. The software enters the #include statements in the preprocessed code used for analysis, but does not modify the original source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option -include. See “Command-Line Information” on page 2-74.

Why Use This Option

There can be many reasons why you want to #include a file in all your source files.

For instance, you can collect in one header file all workarounds for compilation errors. Use this option to provide the header file for analysis. Suppose you have compilation issues because Polyspace does not recognize certain compiler-specific keywords. To work around the issues, #define the keywords in a header file and provide the header file with this option.

Settings

No Default

Specify the file name to be included in every file involved in the analysis.

Polyspace still acts on other directives such as #include <include_file.h>.

Command-Line Information

Parameter: -include

Default: None

Value: *file* (Use -include multiple times for multiple files)

Example (Bug Finder): polyspace-bug-finder -include `pwd`/sources/a_file.h -include /inc/inc_file.h

Example (Code Prover): polyspace-code-prover -include `pwd`/sources/a_file.h -include /inc/inc_file.h

Example (Bug Finder Server): polyspace-bug-finder-server -include `pwd`/sources/a_file.h -include /inc/inc_file.h

Example (Code Prover Server): polyspace-code-prover-server -include `pwd`/sources/a_file.h -include /inc/inc_file.h

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

“Gather Compilation Options Efficiently”

Include folders (-I)

View include folders used for analysis

Description

This option is relevant only for the user interface of the Polyspace desktop products.

View the include folders used for analysis.

Set Option

This is not an option that you set in your project configuration. You can only view the include folders in the configuration associated with a result. For instance, in the user interface:

- To add include folders, on the **Project Browser**, right-click your project. Select **Add Source**.
- To view the include folders that you used, with your results open, select **Window > Show/Hide View > Configuration**. Under the node **Environment Settings**, you see the folders listed under **Include folders**.

Settings

This is a read-only option available only when viewing results in the user interface of the Polyspace desktop products. Unlike other options, you do not specify include folders on the **Configuration** pane. Instead, you add your include folders on the **Project Browser** pane.

See Also

-I | Include (-include)

Ignore link errors (-no-extern-c)

Ignore certain linking errors

Description

Specify that the analysis must ignore certain linking errors.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node. See “Dependency” on page 2-77 for other options that you must also enable.

Command line: Use the option `-no-extern-C`. See “Command-Line Information” on page 2-77.

Why Use This Option

Some functions may be declared inside an `extern "C" { }` block in some files and not in others. Then, their linkage is not the same and it causes a link error according to the ANSI standard.

Applying this option will cause Polyspace to ignore this error. This permissive option may not resolve all the extern C linkage errors.

Settings

On

Ignore linking errors if possible.

Off (default)

Stop analysis for linkage errors.

Dependency

This option is available only if you set `Source code language (-lang)` to CPP or C-CPP.

Command-Line Information

Parameter: `-no-extern-C`

Default: `off`

Example (Bug Finder): `polyspace-bug-finder -lang cpp -no-extern-C`

Example (Code Prover): `polyspace-code-prover -lang cpp -no-extern-C`

Example (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -no-extern-C`

Example (Code Prover Server): `polyspace-code-prover-server -lang cpp -no-extern-C`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Constraint setup (-data-range-specifications)

Constrain global variables, function inputs and return values of stubbed functions

Description

This option applies primarily to a Code Prover analysis. In Bug Finder, you can only specify external constraints on global variables.

Specify constraints (also known as data range specifications or DRS) for global variables, function inputs and return values of stubbed functions using a **Constraint Specification** template file. The template file is an XML file that you can generate in the Polyspace user interface.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-data-range-specifications`. See “Command-Line Information” on page 2-79.

Why Use This Option

Use this option for specifying constraints outside 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 does not occur at run time, the orange check indicates 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 of stubbed functions.

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

Settings

No Default

Enter full path to the template file. Alternately, click  to open a **Constraint Specification** wizard. This wizard allows you to generate a template file or navigate to an existing template file.

For more information, see “Specify External Constraints”.

Command-Line Information

Parameter: -data-range-specifications

Value: *file*

No Default

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -data-range-specifications "C:\DRS\range.xml"

Example (Code Prover): polyspace-code-prover -sources *file_name* -data-range-specifications "C:\DRS\range.xml"

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -data-range-specifications "C:\DRS\range.xml"

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -data-range-specifications "C:\DRS\range.xml"

See Also

Functions to stub (-functions-to-stub) | Ignore default initialization of global variables (-no-def-init-glob)

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify External Constraints”

Ignore default initialization of global variables (`-no-def-init-glob`)

Consider global variables as uninitialized unless explicitly initialized in code

Description

This option applies to Code Prover only. It does not affect a Bug Finder analysis.

Specify that Polyspace must not consider global and static variables as initialized unless they are explicitly initialized in the code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-no-def-init-glob`. See “Command-Line Information” on page 2-81.

Why Use This Option

The C99 Standard specifies that global variables are implicitly initialized. The default analysis follows the Standard and considers this implicit initialization.

If you want to initialize specific global variables explicitly, use this option to find the instances where global variables are not explicitly initialized.

Settings

On

Polyspace ignores implicit initialization of global and static variables. The verification generates a red **Non-initialized variable** error if your code reads a global or static variable before writing to it.

If you enable this option, global variables are considered uninitialized unless you explicitly initialize them in the code. Note that this option overrides the option **Variables to initialize** (`-main-generator-writes-variables`). Even if you initialize variables with the generated `main`, this option forces the analysis to ignore the initialization.

Off (default)

Polyspace considers global variables and static variables to be initialized according to C99 or ISO C++ standards. For instance, the default values are:

- 0 for `int`
- 0 for `char`
- 0.0 for `float`

Tips

Static local variables have the same lifetime as global variables even though their visibility is limited to the function where they are defined. Therefore, the option applies to static local variables.

Command-Line Information

Parameter: -no-def-init-glob

Default: Off

Example (Code Prover): polyspace-code-prover -sources *file_name* -no-def-init-glob

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -no-def-init-glob

See Also

Non-initialized variable

Topics

“Prepare Scripts for Polyspace Analysis”

Functions to stub (-functions-to-stub)

Specify functions to stub during analysis

Description

Specify functions to stub during analysis.

For specified functions, Polyspace :

- Ignores the function definition even if it exists.
- Assumes that the function inputs and outputs have full range of values allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-functions-to-stub`. See “Command-Line Information” on page 2-83.

Why Use This Option

If you want the analysis to ignore the code in a function body, you can stub the function.



For instance:

- Suppose you have not completed writing the function and do not want the analysis to consider the function body. You can use this option to stub the function and then specify constraints on its return value and modifiable arguments.
- Suppose the analysis of a function body is imprecise. The analysis assumes that the function returns all possible values that the function return type allows. You can use this option to stub the function and then specify constraints on its return value.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

When entering function names, use either the basic syntax or, to differentiate overloaded functions, the argument syntax. For the argument syntax, separate function arguments with semicolons. See the following code and table for examples.

```
//simple function  
void test(int a, int b);
```

```
//C++ template function

Template <class myType>
myType test(myType a, myType b);

//C++ class method

class A {
    public:
    int test(int var1, int var2);
};

//C++ template class method

template <class myType> class A
{
    public:
    myType test(myType var1, myType var2);
};
```

Function Type	Basic Syntax	Argument Syntax
Simple function	test	test(int; int)
C++ template function	test	test(myType; myType)
C++ class method	A::test	A::test(int;int)
C++ template class method	A<myType>::test	A<myType>::test(myType;myType)

Tips

- Code Prover makes assumptions about the arguments and return values of stubbed functions. For example, Polyspace assumes that the return values of stubbed functions are full range. These assumptions can affect checks in other sections of the code. See “Stubbed Functions” (Polyspace Code Prover).
- If you stub a function, you can constrain the range of function arguments and return value. To specify constraints, use the analysis option `Constraint setup (-data-range-specifications)`.
- For C functions, these special characters are allowed: () < > ; _

For C++ functions, these special characters are allowed : () < > ; _ * & []

Space characters are allowed for C++, but are not allowed for C functions.

Command-Line Information

Parameter: -functions-to-stub

No Default

Value: *function1[,function2[,...]]*

Example (Code Prover): `polyspace-code-prover -sources file_name -functions-to-stub function_1,function_2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -functions-to-stub function_1,function_2`

See Also

Constraint setup (-data-range-specifications)

Topics

“Prepare Scripts for Polyspace Analysis”

Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)

Stub autogenerated functions that use lookup tables and model them more precisely

Description

This option is available only for model-generated code. The option is relevant only if you generate code from a Simulink model that uses Lookup Table blocks using MathWorks code generation products.

Specify that the verification must stub autogenerated functions that use certain kinds of lookup tables in their body. The lookup tables in these functions use linear interpolation and do not allow extrapolation. That is, the result of using the lookup table always lies between the lower and upper bounds of the table.

Set Option

If you are running verification from Simulink, use the option “Stub lookup tables” (Polyspace Code Prover) in Simulink Configuration Parameters, which performs the same task.

User interface (desktop products only): In your Polyspace project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-stub-embedded-coder-lookup-table-functions`. See “Command-Line Information” on page 2-86.

Why Use This Option

If you use this option, the verification is more precise and has fewer orange checks. The verification of lookup table functions is usually imprecise. The software has to make certain assumptions about these functions. To avoid missing a run-time error, the verification assumes that the result of using the lookup table is within the full range allowed by the result data type. This assumption can cause many unproven results (orange checks) when a lookup table function is called. By using this option, you narrow down the assumption. For functions that use lookup tables with linear interpolation and no extrapolation, the result is at least within the bounds of the table.

The option is relevant only if your model has Lookup Table blocks. In the generated code, the functions corresponding to Lookup Table blocks also use lookup tables. The function names follow specific conventions. The verification uses the naming conventions to identify if the lookup tables in the functions use linear interpolation and no extrapolation. The verification then replaces such functions with stubs for more precise verification.

Settings

On (default)

For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:

- Does not check for run-time errors in the function body.
- Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.

To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses the function name. In your analysis results, you see that the function is not analyzed. If you place your cursor on the function name, you see the following message:

```
Function has been recognized as an Embedded Coder Lookup-Table function.  
It was stubbed by Polyspace to increase precision.  
Unset the -stub-embedded-coder-lookup-table-functions option to analyze  
the code below.
```

Off

The verification does not stub autogenerated functions that use lookup tables.

Tips

- The option applies to only autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, these functions do not follow the naming conventions for autogenerated functions. The option does not cause them to be stubbed. If you want the same behavior for your handwritten lookup table functions as the autogenerated functions, use the option `-code-behavior-specifications` and map your function to the `__ps_lookup_table_clip` function.
- If you run verification from Simulink, the option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

Command-Line Information

Parameter: `-stub-embedded-coder-lookup-table-functions`

Default: On

Example (Code Prover): `polyspace-code-prover -sources file_name -stub-embedded-coder-lookup-table-functions`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -stub-embedded-coder-lookup-table-functions`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016b

Generate results for sources and (-generate-results-for)

Specify files on which you want analysis results

Description

Specify files on which you want analysis results.

The option applies only to coding rule violations and code metrics. You cannot suppress Code Prover run-time checks from select source and header files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-generate-results-for`. See “Command-Line Information” on page 2-88.

Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. Often, other header files belong to a third-party library. Though these header files are required for a precise analysis, you are not interested in reviewing findings in those headers. Therefore, by default, results are not generated for those headers. If you *are interested* in certain headers from third-party libraries, change the default value of this option.

Settings

Default: source-headers

source-headers

Results appear on source files and header files in the same folder as the source files or in subfolders of source file folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

all-headers


Results appear on source files and all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

custom

Results appear on source files and the files that you specify. If you enter a folder name, results appear on header files in that folder.

Click  to add a field. Enter a file or folder name.

Tips

- 1 Use this option in combination with appropriate values for the option **Do not generate results for** (-do-not-generate-results-for).

If you choose **custom** and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

Generate results for sources and	Do not generate results for	Final Result
custom: C:\Includes \Custom_Library\ 	custom: C:\Includes 	Results are displayed on header files in C:\Includes\Custom_Library\ but not generated for other header files in C:\Includes and its subfolders.
custom: C:\Includes \my_header.h 	custom: C:\Includes\ 	Results are displayed on the header file my_header.h in C:\Includes\ but not generated for other header files in C:\Includes\ and its subfolders.

Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

- 2 If you choose **all-headers** for this option, results are displayed on all header files irrespective of what you specify for the option **Do not generate results for**.

Command-Line Information

Parameter: -generate-results-for

Value: source-headers | all-headers | custom=*file1*[,*file2*[,...]] | custom=*folder1*[,*folder2*[,...]]

Example (Bug Finder): polyspace-bug-finder -lang c -sources *file_name* -misra2 required-rules -generate-results-for custom="C:\usr\include"

Example (Code Prover): polyspace-code-prover -lang c -sources *file_name* -misra2 required-rules -generate-results-for custom="C:\usr\include"

Example (Bug Finder Server): polyspace-bug-finder-server -lang c -sources *file_name* -misra2 required-rules -generate-results-for custom="C:\usr\include"

Example (Code Prover Server): polyspace-code-prover-server -lang c -sources *file_name* -misra2 required-rules -generate-results-for custom="C:\usr\include"

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016a

Do not generate results for (-do-not-generate-results-for)

Specify files on which you do not want analysis results

Description

Specify files on which you do not want analysis results.

The option applies only to coding rule violations, code metrics and unused global variables. You cannot suppress Code Prover run-time checks from source and header files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-do-not-generate-results-for`. See “Command-Line Information” on page 2-93.

Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. If you are not interested in reviewing the findings in those headers, change the default value of this option.

Settings

Default: `include-folders`

`include-folders`

Results are not generated for header files in include folders.

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

If an include folder is a subfolder of a source folder, results are generated for files in that include folder even if you specify the option value `include-folders`. In this situation, use the option value `custom` and explicitly specify the include folders to ignore.

`all-headers`

Results are not generated for all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

custom

Results are not generated for the files that you specify. If you enter a folder name, results are suppressed from files in that folder.

Click  to add a field. Enter a file or folder name.

Tips

- 1 Use this option appropriately in combination with appropriate values for the option **Generate results for sources and** (-generate-results-for).

If you choose **custom** and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

Generate results for sources and	Do not generate results for	Final Result
custom: C:\Includes \Custom_Library\	custom: C:\Includes	Results are displayed on header files in C:\Includes\Custom_Library\ but not generated for other header files in C:\Includes and its subfolders.
custom: C:\Includes \my_header.h	custom: C:\Includes\	Results are displayed on the header file my_header.h in C:\Includes\ but not generated for other header files in C:\Includes\ and its subfolders.

Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

- 2 If you choose **all-headers** for this option, results are suppressed from all header files irrespective of what you specify for the option **Generate results for sources and**.
- 3 If a defect or coding rule violation involves two files and you do not generate results for one of the files, the defect or rule violation still appears. For instance, if you define two variables with similar-looking names in files `myFile.cpp` and `myFile.h`, you get a violation of the MISRA C++ rule 2-10-1, even if you do not generate results for `myFile.h`. MISRA C++ rule 2-10-1 states that different identifiers must be typographically unambiguous.

The following results can involve more than one file:

MISRA C: 2004 Rules

- MISRA C: 2004 Rule 5.1 — Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
- MISRA C: 2004 Rule 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.
- MISRA C: 2004 Rule 8.8 — An external object or function shall be declared in one file and only one file.

- MISRA C: 2004 Rule 8.9 — An identifier with external linkage shall have exactly one external definition.

MISRA C: 2012 Directives and Rules

- MISRA C: 2012 Directive 4.5 — Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
- MISRA C: 2012 Rule 5.2 — Identifiers declared in the same scope and name space shall be distinct.
- MISRA C: 2012 Rule 5.3 — An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C: 2012 Rule 5.4 — Macro identifiers shall be distinct.
- MISRA C: 2012 Rule 5.5 — Identifiers shall be distinct from macro names.
- MISRA C: 2012 Rule 8.5 — An external object or function shall be declared once in one and only one file.
- MISRA C: 2012 Rule 8.6 — An identifier with external linkage shall have exactly one external definition.

MISRA C++ Rules

- MISRA C++ Rule 2-10-1 — Different identifiers shall be typographically unambiguous.
- MISRA C++ Rule 2-10-2 — Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C++ Rule 3-2-2 — The One Definition Rule shall not be violated.
- MISRA C++ Rule 3-2-3 — A type, object or function that is used in multiple translation units shall be declared in one and only one file.
- MISRA C++ Rule 3-2-4 — An identifier with external linkage shall have exactly one definition.
- MISRA C++ Rule 7-5-4 — Functions should not call themselves, either directly or indirectly.
- MISRA C++ Rule 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.

JSF C++ Rules

- JSF C++ Rule 46 — User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.
- JSF C++ Rule 48 — Identifiers will not differ by only a mixture of case, the presence/absence of the underscore character, the interchange of the letter O with the number 0 or the letter D, the interchange of the letter I with the number 1 or the letter l, the interchange of the letter S with the number 5, the interchange of the letter Z with the number 2 and the interchange of the letter n with the letter h.
- JSF C++ Rule 137 — All declarations at file scope should be static where possible.
- JSF C++ Rule 139 — External objects will not be declared in more than one file.

Polyspace Bug Finder Defects

- Variable shadowing — Variable hides another variable of same name with nested scope.
- Declaration mismatch — Mismatch occurs between function or variable declarations.

- 4 If a global variable is never used after declaration, it appears in Code Prover results as an unused global variable. However, if it is declared in a file for which you do not want results, you do not see the unused variable in your verification results.
- 5 If a result (coding rule violation or Bug Finder defect) is inside a macro, Polyspace typically shows the result on the macro definition instead of the macro occurrences so that you review the result only once. Even if the macro is used in a suppressed file, the result is still shown on the macro definition, *if the definition occurs in an unsuppressed file*.

Command-Line Information

Parameter: -do-not-generate-results-for

Value: all-headers | include-folders | custom=*file1*[,*file2*[,...]] | custom=*folder1*[,*folder2*[,...]]

Example (Bug Finder): polyspace-bug-finder -lang c -sources *file_name* -misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"

Example (Code Prover): polyspace-code-prover -lang c -sources *file_name* -misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"

Example (Bug Finder Server): polyspace-bug-finder-server -lang c -sources *file_name* -misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"

Example (Code Prover Server): polyspace-code-prover-server -lang c -sources *file_name* -misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"

See Also

Generate results for sources and (-generate-results-for)

Topics

"Prepare Scripts for Polyspace Analysis"

Introduced in R2016a

No STL stubs (-no-stl-stubs)

Do not use Polyspace implementations of functions in the Standard Template Library

Description

Specify that the verification must not use Polyspace implementations of the Standard Template Library.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node. See “Dependency” on page 2-94 for other options that you must also enable.

Command line: Use the option `-no-stl-stubs`. See “Command-Line Information” on page 2-94.

Why Use This Option

The analysis uses an efficient implementation of all class templates from the Standard Template Library (STL). If your compiler redefines the templates, in some cases, your compiler implementation can conflict with the Polyspace implementation.

Use this option to prevent Polyspace from using its implementations of STL templates. You must also explicitly provide the path to your compiler includes. See “C++ Standard Template Library Stubbing Errors”.

Settings

On

The verification does not use Polyspace implementations of the Standard Template Library.

Off (default)

The verification uses efficient Polyspace implementations of the Standard Template Library.

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

Command-Line Information

Parameter: `-no-stl-stubs`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -no-stl-stubs`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -no-stl-stubs`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)

Automatically detect certain families of multithreading functions

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify whether the analysis must automatically detect POSIX®, VxWorks®, Windows, µC/OS II and other multithreading functions.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” (Polyspace Code Prover) for other options that you must enable or disable.

Command line: Use the option `-enable-concurrency-detection`. See “Command-Line Information” on page 2-97.

Why Use This Option

If you use this option, Polyspace determines your multitasking model from your use of multithreading functions. In Bug Finder, automatic concurrency detection is enabled by default. In Code Prover, you have to explicitly enable automatic concurrency detection.

In some cases, using automatic concurrency detection can slow down the Code Prover analysis. In those cases, you can choose to not enable this option and explicitly specify your multitasking model. See “Configuring Polyspace Multitasking Analysis Manually”.

Settings

On

If you use one of the supported functions for multitasking, the analysis automatically detects your multitasking model from your code.

For a list of supported multitasking functions and limitations in auto-detection of threads, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

Off (default)

The analysis does not attempt to detect the multitasking model from your code.

If you want to manually configure your multitasking model, see “Configuring Polyspace Multitasking Analysis Manually”.

Dependencies

If you enable this option, your code must contain a `main` function. You cannot use the Code Prover options to generate a `main`.

Command-Line Information

Parameter: `-enable-concurrency-detection`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -enable-concurrency-detection`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -enable-concurrency-detection`

See Also

Show global variable sharing and usage only (`-shared-variables-mode`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Auto-Detection of Thread Creation and Critical Section in Polyspace”

External multitasking configuration

Enable setup of multitasking configuration from external file definitions

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify whether you want to use definitions from external files to set up the multitasking configuration of your Polyspace project. The supported external file formats are:

- ARXML files for AUTOSAR projects
- OIL files for OSEK projects

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node.

Command line: See “Command-Line Information” on page 2-98.

Why Use This Option

If your AUTOSAR project includes ARXML files with ECU configuration parameters, or if your OSEK project includes OIL files, Polyspace can parse these files. The software sets up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

On

Polyspace parses the external files that you provide in the format that you specify to set up the multitasking configuration of your project.

osek

Look for and parse OIL files to extract multitasking description.

autosar

Look for and parse AUTOSAR XML files to extract multitasking description.

Off (default)

Polyspace does not set up the multitasking configuration of your project.

Command-Line Information

There is no single command-line option to turn on external multitasking configuration. By using the `-osek-multitasking` option or the `-autosar-multitasking` option, you enable external multitasking configuration.

See Also

ARXML files selection (`-autosar-multitasking`) | OIL files selection (`-osek-multitasking`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

Introduced in R2018a

OIL files selection (-osek-multitasking)

Set up multitasking configuration from OIL file definition

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify the OIL files that Polyspace parses to set up the multitasking configuration of your OSEK project.

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 2-104 for other options you must also enable.

Command line: Use the option -osek-multitasking. See “Command-Line Information” on page 2-104.

Why Use This Option

If your project includes OIL files, Polyspace can parse these files to set up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

On

Polyspace looks for and parses OIL files to set up your multitasking configuration.

auto

Look for OIL files in your project source and include folders, but not in their subfolders.

custom

Look for OIL files on the specified path and the path subfolders. You can specify a path to the OIL files or to the folder containing the files.

When you select this option, in your source code, Polyspace supports these OSEK multitasking keywords:

- TASK
- DeclareTask
- ActivateTask
- DeclareResource
- GetResource
- ReleaseResource
- ISR
- DeclareEvent
- DeclareAlarm

Polyspace parses the OIL files that you provide for TASK, ISR, RESOURCE, and ALARM definitions. The analysis uses these definitions and the supported multitasking keywords to configure tasks, interrupts, cyclical tasks, and critical sections.

Example: Analyze Your OSEK Multitasking Project

This example shows how to set up the multitasking configuration of an OSEK project and run an analysis on this project. To try the steps in this example, use the demo files in the folder *polyspaceroot/help/toolbox/bugfinder/examples/External_multitasking/OSEK* or *polyspaceroot/help/toolbox/codeprover/examples/External_multitasking/OSEK*. *polyspaceroot* is the Polyspace installation folder. The analysis results apply to this example code.

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;

DeclareAlarm(Cyclic_task_activate);
DeclareResource(res1);
DeclareTask(init);
TASK(afterinit1);

TASK(init) // task
{

    var2++;
    ActivateTask(afterinit1);
    var3++;
    GetResource(res1); // critical section begins
    var1++;
    ReleaseResource(res1); // critical section ends
}

TASK(afterinit1) // task
{
    var3++;
    var2++;
    GetResource(res1); // critical section begins
    var1++;
    ReleaseResource(res1); // critical section ends
}

int var4;
void func()
{
    var4++;
}

TASK(Cyclic_task) // cyclic task
{
    func();
}

void main()
{}
```

To set up your multitasking configuration and analyze the code:

- 1 Copy the contents of *polyspaceroot/help/toolbox/bugfinder/examples/External_multitasking/OSEK* or *polyspaceroot/help/toolbox/codeprover/examples/External_multitasking/OSEK* to your machine, for instance in *C:\Polyspace_workspace\OSEK*.
- 2 Run an analysis on your OSEK project by using the command:

- Bug Finder:

```
polyspace-bug-finder -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

- Code Prover:

```
polyspace-code-prover -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

- Code Prover Server:

```
polyspace-code-prover-server -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

Bug Finder detects a data race on variable `var3` because of multiple read and write operation from tasks `init` and `afterinit1`. See [Data race](#).

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;
```

There is no defect on `var2` since `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Similarly, there is no defect on `var1` because it is protected by the `GetResource()` and `ReleaseResource()` calls.

Code Prover detects that `var3` is a potentially unprotected global variable because it is used in tasks `init` and `afterinit1` with no protection from interruption during the read and write operations. The analysis also shows that the cyclic task operation on `var4` can potentially cause an overflow. See [Potentially unprotected variable](#) and [Overflow](#).

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;

...
void func()
{
    var4++;
}
```

Variable `var2` is not shared because `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Variable `var1` is a protected variable through the critical sections from the `GetResource()` and `ReleaseResource()` calls.

To see how Polyspace models the TASK, ISR, and RESOURCE definitions from your OIL files, open the **Concurrency window** from the **Dashboard** pane.

Off (default)

Polyspace does not set up a multitasking configuration for your OSEK project.

Additional Considerations

- The analysis ignores `TerminateTask()` declarations in your source code and considers that subsequent code is executed.
- Polyspace ignores syntax elements of your OIL files that do not follow the syntax defined here.

Dependencies

To enable this option in the user interface of the desktop products, first select the option External multitasking configuration.

Command-Line Information

Parameter: `-osek-multitasking`

Value: `auto|custom='file1 [,file2, dir1,...]'`

Default: Off

Example (Bug Finder): `polyspace-bug-finder -sources source_path -I include_path -osek-multitasking custom='path\to\file1.oil, path\to\dir'`

Example (Code Prover): `polyspace-code-prover -sources source_path -I include_path -osek-multitasking custom='path\to\file1.oil, path\to\dir'`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources source_path -I include_path -osek-multitasking custom='path\to\file1.oil, path\to\dir'`

Example (Code Prover Server): `polyspace-code-prover-server -sources source_path -I include_path -osek-multitasking custom='path\to\file1.oil, path\to\dir'`

See Also

Show global variable sharing and usage only (`-shared-variables-mode`)

Introduced in R2017b

ARXML files selection (-autosar-multitasking)

Set up multitasking configuration from ARXML file definitions

Description

To detect data races in large AUTOSAR applications, use this option with Polyspace Bug Finder.

This option is not available for code generated from MATLAB code or Simulink models.

Specify the ARXML files that Polyspace parses to set up the multitasking configuration of your AUTOSAR project.

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 2-106 for other options you must also enable.

Command line: Use the option -autosar-multitasking. See “Command-Line Information” on page 2-104.

Why Use This Option

If your project includes ARXML files with <ECUC-CONTAINER-VALUE> elements, Polyspace can parse these files to set up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

On

Polyspace looks for and parses ARXML files to set up your multitasking configuration.

When you select this option, the software assumes that you use the OSEK multitasking API in your source code to declare and define tasks and interrupts. Polyspace supports these OSEK multitasking keywords:

- TASK
- DeclareTask
- ActivateTask
- DeclareResource
- GetResource
- ReleaseResource
- ISR
- DeclareEvent
- DeclareAlarm

Polyspace parses the ARXML files that you provide for `OsTask`, `OsIsr`, `OsResource`, `OsAlarm`, and `OsEvent` definitions. The analysis uses these definitions and the supported multitasking keywords to configure tasks, interrupts, cyclical tasks, and critical sections.

To see how Polyspace models the `OsTask`, `OsIsr`, and `OsResource` definitions from your ARXML files, open the **Concurrency window** from the **Dashboard** pane. In that window, under the **Entry points** column, the names of the elements are extracted from their `<SHORT-NAME>` values in the ARXML files.

Off (default)

Polyspace does not set up a multitasking configuration for your AUTOSAR project.

Additional Considerations

- The analysis ignores `TerminateTask()` declarations in your source code and considers that subsequent code is executed.
- Polyspace supports multitasking configuration only from ARXML files for AUTOSAR specification version 4.0 and later.

Dependencies

To enable this option in the user interface of the desktop products, first select the option **External multitasking configuration**.

Command-Line Information

Parameter: `-autosar-multitasking`

Value: `file1 [,file2, dir1,...]`

Default: Off

Example (Bug Finder): `polyspace-bug-finder -sources source_path -I include_path -autosar-multitasking C:\Polyspace_Workspace\AUTOSAR\myFile.arxml`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources source_path -I include_path -autosar-multitasking C:\Polyspace_Workspace\AUTOSAR\myFile.arxml`

See Also

Enable automatic concurrency detection for Code Prover (`-enable-concurrency-detection`) | External multitasking configuration | OIL files selection (`-osek-multitasking`) | Show global variable sharing and usage only (`-shared-variables-mode`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

Introduced in R2018a

Configure multitasking manually

Consider that code is intended for multitasking

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify whether your code is a multitasking application. This option allows you to manually configure the multitasking structure for Polyspace.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node.

Command line: See “Command-Line Information” on page 2-108.

Why Use This Option

By default, Bug Finder determines your multitasking model from your use of multithreading functions. In Code Prover, you have to enable automatic concurrency detection with the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`. However, in some cases, using automatic concurrency detection can slow down the Code Prover analysis.

In cases where automatic concurrency detection is not supported, you can explicitly specify your multitasking model by using this option. Once you select this option, you can explicitly specify your entry point functions, cyclic tasks, interrupts and protection mechanisms for shared variables, such as critical section details.

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.
See “Global Variables” (Polyspace Code Prover Access).
- Whether a run-time error can occur.

For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see “Concurrency Defects” (Polyspace Bug Finder Access).

Settings

On

The code is intended for a multitasking application.

You have to explicitly specify your multitasking configuration using other Polyspace options. See “Configuring Polyspace Multitasking Analysis Manually”.

Off (default)

The code is not intended for a multitasking application.

Disabling the option has this additional effect in Code Prover:

- If a `main` exists, Code Prover verifies only those functions that are called by the `main`.
- If a `main` does not exist, Polyspace verifies the functions that you specify. To verify the functions, Polyspace generates a `main` function and calls functions from the generated `main` in a sequence that you specify. For more information, see `Verify module or library (-main-generator)`.

Tips

If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

Command-Line Information

There is no single command-line option to turn on multitasking analysis. By using any of the options `Tasks (-entry-points)`, `Cyclic tasks (-cyclic-tasks)` or `Interrupts (-interrupts)`, you turn on multitasking analysis.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Critical section details (-critical-section-begin -critical-section-end)` | `Cyclic tasks (-cyclic-tasks)` | `Tasks (-entry-points)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”

“Protections for Shared Variables in Multitasking Code”

Tasks (-entry-points)

Specify functions that serve as tasks to your multitasking application

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that serve as tasks to your code. If the function does not exist, the verification warns you and continues the verification.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-110 for other options you must also enable.

Command line: Use the option `-entry-points`. See “Command-Line Information” on page 2-110.

Why Use This Option

Use this option when your code is intended for multitasking.

To specify cyclic tasks and interrupts, use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`. Use this option to specify other tasks.

A Code Prover analysis uses your specifications to determine:

- Whether a global variable is shared.
See “Global Variables” (Polyspace Code Prover Access).
- Whether a run-time error can occur.



For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see “Concurrency Defects” (Polyspace Bug Finder Access).

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option **Configure multitasking** manually.

Tips

- In Code Prover, the functions representing entry points must have the form

```
void functionName (void)
```
- If a function `func` takes arguments, you cannot use it directly as task. To use `func` as task:
 - 1 Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
 - 2 Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
 - 3 Specify `newFunc` as a task.
- If you specify a function as a task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

```
task func_name must be a userdef function without parameters
```

A Bug Finder analysis continues but does not consider the function as an entry point.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.
- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.

Command-Line Information

Parameter: `-entry-points`

No Default

Value: `function1[,function2[,...]]`

Example (Bug Finder): `polyspace-bug-finder -sources file_name -entry-points func_1,func_2`

Example (Code Prover): `polyspace-code-prover -sources file_name -entry-points func_1,func_2`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -entry-points func_1,func_2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -entry-points func_1,func_2`

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Show global variable sharing and usage only (-shared-variables-mode)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”

“Protections for Shared Variables in Multitasking Code”

Cyclic tasks (-cyclic-tasks)

Specify functions that represent cyclic tasks

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that represent cyclic tasks. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Can be interrupted by noncyclic tasks, other cyclic tasks and interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and interrupts are specified with the option `Interrupts (-interrupts)`.

To model a cyclic task that cannot be interrupted by other cyclic tasks, specify the task as nonpreemptable. See `-non-preemptable-tasks`. For examples, see “Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server).

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-113 for other options you must also enable.

Command line: Use the option `-cyclic-tasks`. See “Command-Line Information” on page 2-114.

Why Use This Option

Use this option to specify cyclic tasks in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.
See “Global Variables” (Polyspace Code Prover Access).
- Whether a run-time error can occur.

For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For the **Data race** defect, the software establishes the following relations between preemptable tasks and other tasks.

- *Data race between two preemptable tasks:*

Unless protected, two operations in different preemptable tasks can interfere with each other. If the operations use the same shared variable without protection, a data race can occur.

If both operations are atomic, to see the defect, you have to enable the checker **Data race including atomic operations**.



- *Data race between a preemptable task and a nonpreemptable task or interrupt:*
 - An atomic operation in a preemptable task cannot interfere with an operation in a nonpreemptable task or an interrupt. Even if the operations use the same shared variable without protection, a data race cannot occur.
 - A nonatomic operation in a preemptable task also cannot interfere with an operation in a nonpreemptable task or an interrupt. However, the latter operation can interrupt the former. Therefore, if the operations use the same shared variable without protection, a data race can occur.

For more information, see “Concurrency Defects” (Polyspace Bug Finder Access).

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option **Configure multitasking manually**.

Tips

- In Code Prover, the functions representing cyclic tasks must have the form


```
void functionName (void)
```
- If a function `func` takes arguments, you cannot use it directly as a cyclic task. To use `func` as cyclic task:
 - 1 Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
 - 2 Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
 - 3 Specify `newFunc` as cyclic task.
- If you specify a function as a cyclic task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:


```
task func_name must be a userdef function without parameters
```

A Bug Finder analysis continues but does not consider the function as a cyclic task.
- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.

Command-Line Information

Parameter: `-cyclic-tasks`

No Default

Value: `function1[,function2[,...]]`

Example (Bug Finder): `polyspace-bug-finder -sources file_name -cyclic-tasks func_1,func_2`

Example (Code Prover): `polyspace-code-prover -sources file_name -cyclic-tasks func_1,func_2`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -cyclic-tasks func_1,func_2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -cyclic-tasks func_1,func_2`

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Interrupts (-interrupts)` | `Show global variable sharing and usage only (-shared-variables-mode)` | `Tasks (-entry-points)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”

“Protections for Shared Variables in Multitasking Code”

“Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server)

Introduced in R2016b

Interrupts (-interrupts)

Specify functions that represent nonpreemptable interrupts

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that represent nonpreemptable interrupts. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Cannot be interrupted by noncyclic tasks, cyclic tasks or other interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and cyclic tasks are specified with the option `Cyclic tasks (-cyclic-tasks)`.

To model an interrupt that can be interrupted by other interrupts, specify the interrupt as preemptable. See `-preemptable-interrupts`. For examples, see “Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server).

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-116 for other options you must also enable.

Command line: Use the option `-interrupts`. See “Command-Line Information” on page 2-117.

Why Use This Option

Use this option to specify interrupts in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.
See “Global Variables” (Polyspace Code Prover Access).
- Whether a run-time error can occur.

For instance, if the operation `var=INT_MAX;` occurs in an interrupt and `var++` occurs in the body of a task, an overflow can occur if the interrupt excepts before the operation in the task. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For the `Data race` defect, the analysis establishes the following relations between interrupts and other tasks:

- *Data race between two interrupts:*

Two operations in different interrupts cannot interfere with each other (unless one of the interrupts is preemptable). Even if the operations use the same shared variable without protection, a data race cannot occur.



- *Data race between an interrupt and another task:*
 - An operation in an interrupt cannot interfere with an atomic operation in any other task. Even if the operations use the same shared variable without protection, a data race cannot occur.
 - An operation in an interrupt can interfere with a nonatomic operation in any other task unless the other task is also a nonpreemptable interrupt. Therefore, if the operations use the same shared variable without protection, a data race can occur.

See “Concurrency Defects” (Polyspace Bug Finder Access).

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option **Configure multitasking manually**.

Tips

- In Code Prover, the functions representing interrupts must have the form

```
void functionName (void)
```
- If a function `func` takes arguments, you cannot use it directly as an interrupt. To use `func` as interrupt:
 - 1 Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
 - 2 Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
 - 3 Specify `newFunc` as `interrupt`.
- If you specify a function as an interrupt, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

```
task func_name must be a userdef function without parameters
```

A Bug Finder analysis continues but does not consider the function as an interrupt.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.
- The Polyspace multitasking analysis assumes that an interrupt cannot interrupt itself.

Command-Line Information

Parameter: -interrupts

No Default

Value: *function1[,function2[,...]]*

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -interrupts *func_1,func_2*

Example (Code Prover): polyspace-code-prover -sources *file_name* -interrupts *func_1,func_2*

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -interrupts *func_1,func_2*

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -interrupts *func_1,func_2*

See Also

-non-preemptable-tasks | -preemptable-interrupts | Cyclic tasks (-cyclic-tasks) | Show global variable sharing and usage only (-shared-variables-mode) | Tasks (-entry-points)

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”

“Protections for Shared Variables in Multitasking Code”

“Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server)

Introduced in R2016b

Critical section details (-critical-section-begin -critical-section-end)

Specify functions that begin and end critical sections

Description

This option is not available for code generated from MATLAB code or Simulink models.

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock function and an unlock function.

```
lock();
/* Critical section code */
unlock();
```

Specify the lock and unlock function names for your critical sections (for instance, `lock()` and `unlock()` in above example).

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-119 for other options you must also enable.

Command line: Use the option `-critical-section-begin` and `-critical-section-end`. See “Command-Line Information” on page 2-120.

Why Use This Option

When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Therefore, critical section operations in the other tasks cannot interrupt critical section operations in `my_task`.

For instance, the operation `var++` in `my_task1` and `my_task2` cannot interrupt each other.

```
int var;

void my_task1() {
    my_lock();
    var++;
    my_unlock();
}

void my_task2() {
    my_lock();
    var++;
    my_unlock();
}
```

Using your specifications, a Code Prover verification checks if your placement of lock and unlock functions protects all shared variables from concurrent access. When determining values of those variables, the verification accounts for the fact that critical sections in different tasks do not interrupt each other.

A Bug Finder analysis uses the critical section information to look for concurrency defects such as data race and deadlock.



Settings

No Default

Click  to add a field.

- In **Starting routine**, enter name of lock function.
- In **Ending routine**, enter name of unlock function.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option **Configure multitasking manually**.

Tips

- 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”.
- For function calls that begin and end critical sections, Polyspace ignores the function arguments.

For instance, Polyspace treats the two code sections below as the same critical section.

Starting routine: my_lock	
Ending routine: my_unlock	
<pre>void my_task1() { my_lock(1); /* Critical section code */ my_unlock(1); }</pre>	<pre>void my_task2() { my_lock(2); /* Critical section code */ my_unlock(2); }</pre>

To work around the limitation, see “Define Critical Sections with Functions That Take Arguments”.

- The functions that begin and end critical sections must be functions. For instance, if you define a function-like macro:

```
#define init() num_locks++
```

You cannot use the macro `init()` to begin or end a critical section.

- When you use multiple critical sections, you can run into issues such as:

- **Deadlock:** A sequence of calls to lock functions causes two tasks to block each other.
- **Double lock:** A lock function is called twice in a task without an intermediate call to an unlock function.

Use Polyspace Bug Finder to detect such issues. See “Concurrency Defects” (Polyspace Bug Finder Access).

Then, use Polyspace Code Prover to detect if your placement of lock and unlock functions actually protects all shared variables from concurrent access. See “Global Variables” (Polyspace Code Prover Access).

- When considering possible values of shared variables, a Code Prover verification takes into account your specifications for critical sections.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information

Parameter: `-critical-section-begin` | `-critical-section-end`

No Default

Value: `function1:cs1[,function2:cs2[,...]]`

Example (Bug Finder): `polyspace-bug_finder -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

Example (Code Prover): `polyspace-code-prover -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

Example (Bug Finder Server): `polyspace-bug_finder-server -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | Cyclic tasks (`-cyclic-tasks`) | Interrupts (`-interrupts`) | Tasks (`-entry-points`) | Temporally exclusive tasks (`-temporal-exclusions-file`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”

“Protections for Shared Variables in Multitasking Code”

“Define Atomic Operations in Multitasking Code”

“Define Critical Sections with Functions That Take Arguments”

“Concurrency Defects” (Polyspace Bug Finder Access)

“Global Variables” (Polyspace Code Prover Access)

Temporally exclusive tasks (-temporal-exclusions-file)

Specify entry point functions that cannot execute concurrently

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify entry point functions that cannot execute concurrently. The execution of the functions cannot overlap with each other.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-121 for other options you must also enable.

Command line: Use the option `-temporal-exclusions-file`. See “Command-Line Information” on page 2-122.

Why Use This Option


Use this option to implement temporal exclusion in multitasking code.

A Code Prover verification checks if specifying certain tasks as temporally exclusive protects all shared variables from concurrent access. When determining possible values of those shared variables, the verification accounts for the fact that temporally exclusive tasks do not interrupt each other. See “Global Variables” (Polyspace Code Prover Access).



A Bug Finder analysis uses the temporal exclusion information to look for concurrency defects such as data race. See “Concurrency Defects” (Polyspace Bug Finder Access).

Settings

No Default

Click  to add a field. In each field, enter a space-separated list of functions. Polyspace considers that the functions in the list cannot execute concurrently.

Enter the function names manually or choose from a list.

- Click  to add a field and enter the function names.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option **Configure multitasking manually**.

Tips

When considering possible values of shared variables, a Code Prover verification takes into account your specifications for temporally exclusive tasks.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking in Code Prover, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information

For the command-line option, create a temporal exclusions file in the following format:

- On each line, enter one group of temporally excluded tasks.
- Within a line, the tasks are separated by spaces.

To enter comments, begin with #. For an example, see the file *polyspaceroot*\polyspace\examples\cxx\Code_Prover_Example\sources\temporal_exclusions.txt. Here, *polyspaceroot* is the Polyspace installation folder, for example C:\Program Files\Polyspace\R2019a.

Parameter: -temporal-exclusions-file

No Default

Value: Name of temporal exclusions file

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"

Example (Code Prover): polyspace-code-prover -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"

See Also

-non-preemptable-tasks | -preemptable-interrupts | Critical section details (-critical-section-begin -critical-section-end) | Cyclic tasks (-cyclic-tasks) | Interrupts (-interrupts) | Tasks (-entry-points)

Topics

"Prepare Scripts for Polyspace Analysis"

"Analyze Multitasking Programs in Polyspace"

"Configuring Polyspace Multitasking Analysis Manually"

"Protections for Shared Variables in Multitasking Code"

"Define Atomic Operations in Multitasking Code"

"Concurrency Defects" (Polyspace Bug Finder Access)

"Global Variables" (Polyspace Code Prover Access)

Set checkers by file (-checkers-selection-file)

Define a custom set of coding standards checks for your analysis

Description

Specify the full path of a configuration XML file where you define custom selections of coding standards checkers. You can, in the same file, define a custom selection of checkers for each of these coding standards:

- 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*)

You can also define custom rules to match identifiers in your code against text patterns you specify.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line: Use the option `-checkers-selection-file`. See “Command-Line Information” on page 2-125.

When you enable this option, set the coding standards you select to `from-file` to use the specified configuration file.


Why Use This Option

Use this option to define a selection of coding standard checkers specific to your organization. The configuration of different coding standards is consolidated in a single XML file which you can reuse across projects to enforce common coding standards.

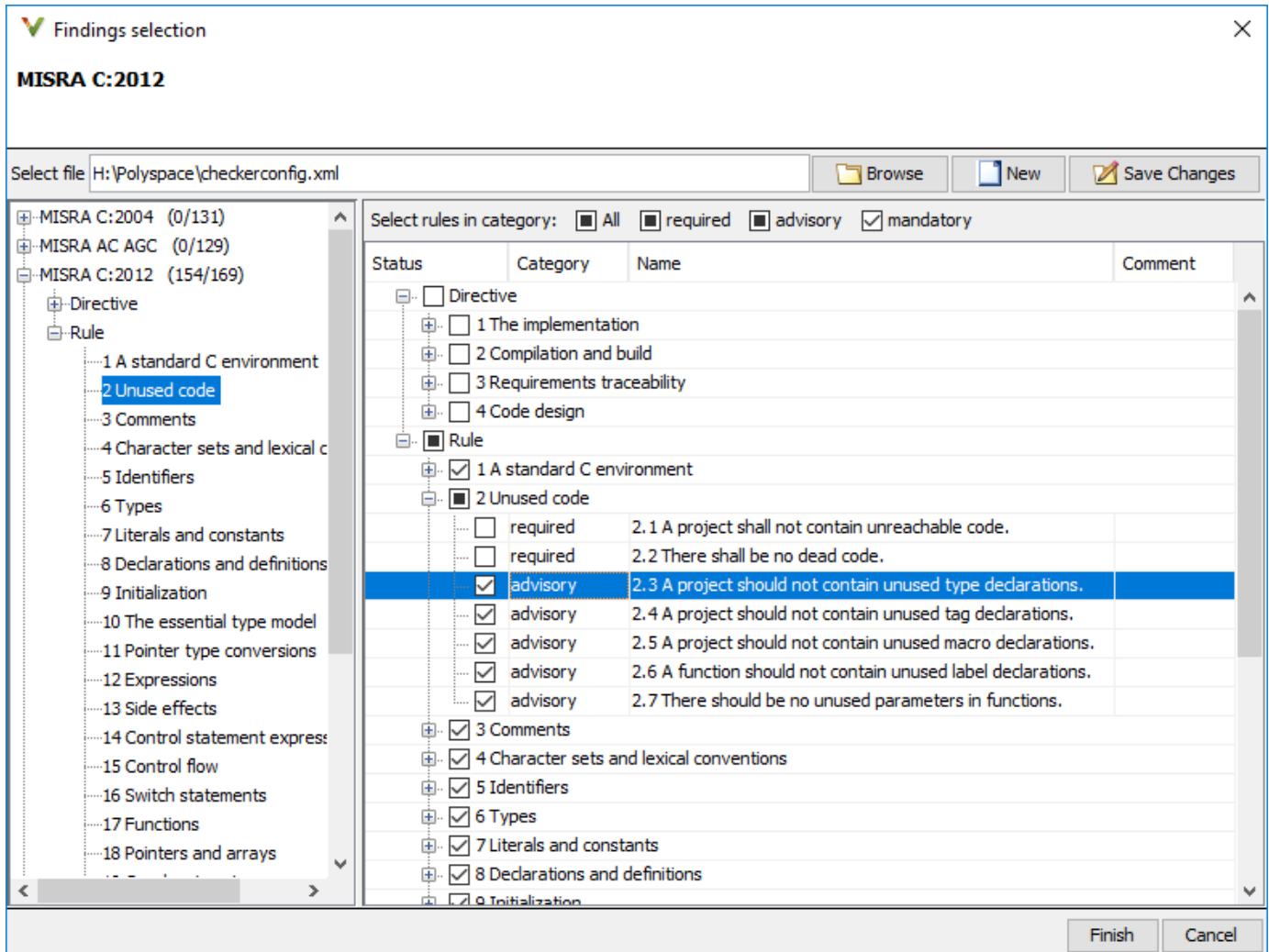
Settings

On

Polyspace checks your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file you specify.

To create a configuration file, open the **Findings selection** window by clicking . In the left pane, choose the coding standard you want to configure, then select the rules you want to check for this coding standard in the right pane.

To use or update an existing file, enter the full path to the file in the field provided or click **Browse** in the **Findings selection** window.



Off (default)

Polyspace does not check your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file you specify.

Tips

- With the Polyspace desktop products, specify the coding standard configuration in the user interface of the desktop products. When you save the configuration, an XML file is automatically created for use in the current and other projects.
- 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 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>
```

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.

Command-Line Information

Parameter: -checkers-selection-file

Value: Full path of XML configuration file

Default: Off

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file

Example (Code Prover): polyspace-code-prover -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file

See Also

Do not generate results for (-do-not-generate-results-for)

Topics

“Prepare Scripts for Polyspace Analysis”

“Check for Coding Standard Violations”

Check MISRA C:2004 (-misra2)

Check for violation of MISRA C:2004 rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of MISRA C:2004 rules. Each value of the option corresponds to a subset of rules to check.


Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-128 for other options that you must also enable.

Command line: Use the option `-misra2`. See “Command-Line Information” on page 2-128.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

`required-rules`

Check required coding rules.

`single-unit-rules`

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all-rules`

Check required and advisory coding rules.

SQ0-subset1

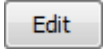
Check only a subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2004)”.

SQ0-subset2

Check a subset of rules including SQ0-subset1 and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2004)”.

from-file

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click , then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to from-file, enable Set checkers by file (-checkers-selection-file).

Dependencies

- This option is available only if you set Source code language (-lang) to C or C-CPP.
For projects with mixed C and C++ code, the MISRA C:2004 checker analyzes only .c files.
- If you set Source code language (-lang) to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:
 - 1 Find coding rule violations in SQ0-subset1. Fix your code to address the violations and rerun verification.
 - 2 Find coding rule violations in SQ0-subset2. Fix your code to address the violations and rerun verification.
- If you select the option single-unit-rules or system-decidable-rules and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

Command-Line Information

Parameter: -misra2

Value: required-rules | all-rules | SQ0-subset1 | SQ0-subset2 | single-unit-rules | system-decidable-rules | from-file

Default: required-rules

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -misra2 all-rules

Example (Code Prover): `polyspace-code-prover -sources file_name -misra2 all-rules`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -misra2 all-rules`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -misra2 all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a


Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics**

node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename.xml*, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_c_2004_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in `polyspaceroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
<code>-misra2 "custom_standard.conf"</code>	<code>-checkers-selection-file misra_c_2004_rules.xml -misra2 from-file</code>

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="on">
    </check>
    ...
  </section>
  ...
</standard>
```

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)

See Also

Do not generate results for (-do-not-generate-results-for)

Topics

"Prepare Scripts for Polyspace Analysis"

"Check for Coding Standard Violations"

"MISRA C:2004 Rules" (Polyspace Code Prover Access)

Check MISRA AC AGC (-misra-ac-agc)

Check for violation of MISRA AC AGC rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*. Each value of the option corresponds to a subset of rules to check.


Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-132 for other options that you must also enable.

Command line: Use the option `-misra-ac-agc`. See “Command-Line Information” on page 2-132.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 AC AGC rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: OBL-rules

OBL-rules

Check required coding rules.

OBL-REC-rules

Check required and recommended rules.

single-unit-rules

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

system-decidable-rules

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

all-rules

Check required, recommended and readability-related rules.

SQ0-subset1


Check a subset of rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (AC AGC)”.

SQ0-subset2

Check a subset of rules including SQ0-subset1 and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (AC AGC)”.

from-file

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click , then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to from-file, enable Set checkers by file (-checkers-selection-file).

Dependencies

- This option is available only if you set Source code language (-lang) to C or C-CPP.
For projects with mixed C and C++ code, the MISRA AC AGC checker analyzes only .c files.
- If you set Source code language (-lang) to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:
 - 1 Find coding rule violations in SQ0-subset1. Fix your code to address the violations and rerun verification.
 - 2 Find coding rule violations in SQ0-subset2. Fix your code to address the violations and rerun verification.
- If you select the option single-unit-rules or system-decidable-rules and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

Command-Line Information

Parameter: -misra-ac-agc

Value: OBL-rules | OBL-REC-rules | single-unit-rules | system-decidable-rules | all-rules | SQ0-subset1 | SQ0-subset2 | from-file

Default: OBL-rules

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -misra-ac-agc all-rules

Example (Code Prover): `polyspace-code-prover -sources file_name -misra-ac-agc all-rules`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -misra-ac-agc all-rules`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -misra-ac-agc all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a


Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics**

node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename.xml*, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_ac_agc_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in `polyspaceroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
<code>-misra-ac-agc "custom_standard.conf"</code>	<code>-checkers-selection-file misra_ac_agc_rules.xml -misra-ac-agc from-file</code>

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="on">
    </check>
    ...
  </section>
  ...
</standard>
```

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)

See Also

Do not generate results for (-do-not-generate-results-for)

Topics

“Prepare Scripts for Polyspace Analysis”

“Check for Coding Standard Violations”

“MISRA C:2004 Rules” (Polyspace Code Prover Access)

Check MISRA C:2012 (-misra3)

Check for violations of MISRA C:2012 rules and directives

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violations of MISRA C:2012 guidelines. Each value of the option corresponds to a subset of guidelines to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-136 for other options that you must also enable.

Command line: Use the option `-misra3`. See “Command-Line Information” on page 2-137.

Why Use This Option

Use this option to specify the subset of MISRA C:2012 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

Settings

Default: mandatory-required

mandatory

Check for mandatory guidelines.


mandatory-required

Check for mandatory and required guidelines.

- Mandatory guidelines: Your code must comply with these guidelines.
- Required guidelines: You may deviate from these guidelines. However, you must complete a formal deviation record, and your deviation must be authorized.

See Section 5.4 of the MISRA C:2012 guidelines. For an example of a deviation record, see Appendix I of the MISRA C:2012 guidelines.

Note To turn off some required guidelines, instead of mandatory-required select custom. To

clear specific guidelines, click . In the **Comment** column, enter your rationale for disabling a guideline. For instance, you can enter the Deviation ID that refers to a deviation record for the guideline. The rationale appears in your generated report.

single-unit-rules

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

system-decidable-rules

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

all

Check for mandatory, required, and advisory guidelines.

SQ0-subset1

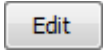
Check for only a subset of guidelines. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2012)”.

SQ0-subset2

Check for the subset `SQ0-subset1`, plus some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2012)”.

from-file

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click , then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to C or C-CPP.

For projects with mixed C and C++ code, the MISRA C:2012 checker analyzes only `.c` files.

- If you set `Source code language (-lang)` to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:
 - 1 Find coding rule violations in `SQ0-subset1`. Fix your code to address the violations and rerun verification.
 - 2 Find coding rule violations in `SQ0-subset2`. Fix your code to address the violations and rerun verification.

- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.
- Polyspace Code Prover does not support checking of the following:
 - MISRA C:2012 Directive 4.13 and 4.14
 - MISRA C:2012 Rule 21.13, 21.14, and 21.17 - 21.20
 - MISRA C:2012 Rule 22.1 - 22.4 and 22.6 - 22.10

For support of all MISRA C: 2012 rules including the security guidelines in Amendment 1, use Polyspace Bug Finder.

Command-Line Information

Parameter: `-misra3`

Value: `mandatory` | `mandatory-required` | `single-unit-rules` | `system-decidable-rules` | `all` | `SQ0-subset1` | `SQ0-subset2` | `from-file`

Default: `mandatory-required`

Example (Bug Finder): `polyspace-bug-finder -lang c -sources file_name -misra3 mandatory-required`

Example (Code Prover): `polyspace-code-prover -lang c -sources file_name -misra3 mandatory-required`

Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources file_name -misra3 mandatory-required`

Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources file_name -misra3 mandatory-required`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a


Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics**

node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as `filename.xml`, where `filename` is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_c_2012_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in `polyspaceroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, `polyspaceroot` is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
<code>-misra3 "custom_standard.conf"</code>	<code>-checkers-selection-file misra_c_2012_rules.xml -misra3 from-file</code>

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="on">
      </check>
    ...
  </section>
  ...
</standard>
```

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)

See Also

Do not generate results for (`-do-not-generate-results-for`)

Topics

"Prepare Scripts for Polyspace Analysis"

“Check for Coding Standard Violations”
“MISRA C:2012 Directives and Rules” (Polyspace Code Prover Access)

Use generated code requirements (-misra3-agc-mode)

Check for violations of MISRA C:2012 rules and directives that apply to generated code

Description

Specify whether to use the MISRA C:2012 categories for automatically generated code. This option changes which rules are mandatory, required, or advisory.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 2-141 for other options that you must also enable.

Command line: Use the option -misra3-agc-mode. See “Command-Line Information” on page 2-141.

Why Use This Option

Use this option to specify that you are checking for MISRA C:2012 rules in generated code. The option modifies the MISRA C:2012 subsets so that they are tailored for generated code.

Settings

Off (default)

Use the normal categories (mandatory, required, advisory) for MISRA C:2012 coding guideline checking.

On (default for analyses from Simulink)

Use the generated code categories (mandatory, required, advisory, readability) for MISRA C:2012 coding guideline checking.

For analyses started from the Simulink plug-in, this option is the default value.

Category changed to Advisory

These rules are changed to advisory:

- 5.3
- 7.1
- 8.4, 8.5, 8.14
- 10.1, 10.2, 10.3, 10.4, 10.6, 10.7, 10.8
- 14.1, 14.4
- 15.2, 15.3
- 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7

- 20.8

Category changed to Readability

These guidelines are changed to readability:

- Dir 4.5
- 2.3, 2.4, 2.5, 2.6, 2.7
- 5.9
- 7.2, 7.3
- 9.2, 9.3, 9.5
- 11.9
- 13.3
- 14.2
- 15.7
- 17.5, 17.7, 17.8
- 18.5
- 20.5

Dependency

To use this option, first select the Check MISRA C:2012 (-misra3) option.

Command-Line Information

Parameter: -misra3-agc-mode

Default: Off

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -misra3 all -misra3-agc-mode

Example (Code Prover): polyspace-code-prover -sources *file_name* -misra3 all -misra3-agc-mode

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -misra3 all -misra3-agc-mode

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -misra3 all -misra3-agc-mode

See Also

Check MISRA C:2012 (-misra3) | Do not generate results for (-do-not-generate-results-for)

Topics

“Prepare Scripts for Polyspace Analysis”

“Check for Coding Standard Violations”

“MISRA C:2012 Directives and Rules” (Polyspace Code Prover Access)

Check custom rules (-custom-rules)

Follow naming conventions for identifiers

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Define naming conventions for identifiers and check your code against them.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line: Use the option `-custom-rules`. See “Command-Line Information” on page 2-144.

Why Use This Option

Use this option to impose naming conventions on identifiers. Using a naming convention allows you to easily determine the nature of an identifier from its name. For instance, if you define a naming convention for structures, you can easily tell whether an identifier represents a structured variable or not.


After analysis, the **Results List** pane lists violations of the naming conventions. On the **Source** pane, for every violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

Settings

On

Polyspace matches identifiers in your code against text patterns you define. Define the text patterns in a custom coding rules file. To create a coding rules file,

- Use the custom rules wizard:

- 1 Click . A **Findings selection** window opens.
- 2 The **Custom** node in the left pane is highlighted. Expand the nodes in the right pane to select custom rule you want to check.
- 3 For every custom rule you want to check:
 - a Select **On** .
 - b In the **Convention** column, enter the error message you want to display if the rule is violated.

For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter `All struct fields must begin with s_`. This message appears on the **Result Details** pane if:

- You specify the **Pattern** as `s_[A-Za-z0-9_]+`.
 - A structure field in your code does not begin with `s_`.
- c** In the **Pattern** column, enter the text pattern.

For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter `s_[A-Za-z0-9_]+`. Polyspace reports violation of rule 4.3 if a structure field does not begin with `s_`.

You can use Perl regular expressions to define patterns. For instance, you can use the following expressions.

Expression	Meaning
.	Matches any single character except newline
[a-z0-9]	Matches any single letter in the set a-z, or digit in the set 0-9
[^a-e]	Matches any single letter not in the set a-e
\d	Matches any single digit
\w	Matches any single alphanumeric character or _
x?	Matches 0 or 1 occurrence of x
x*	Matches 0 or more occurrences of x
x+	Matches 1 or more occurrences of x

For frequent patterns, you can use the following regular expressions:

- `(?!_)[a-z0-9_]+(?!_)`, matches a text pattern that does not start and end with two underscores.

```
int __text; //Does not match
int _text_; //Matches
```


- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)`, matches a text pattern that ends with a specific suffix.

```
int _text_; //Does not match
int _text_s16; //Matches
int _text_s33; // Does not match
```

- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)(_b3|_b8)?`, matches a text pattern that ends with a specific suffix and an optional second suffix.

```
int _text_s16; //Matches
int _text_s16_b8; //Matches
```

For a complete list of regular expressions, see Perl documentation.

To use or update an existing coding rules file, click  to open the **Findings selection** window then do one of the following:

- Enter the full path to the file in the field provided
- Click **Browse** and navigate to the file location.

Off (default)

Polyspace does not check your code against custom naming conventions.

Command-Line Information

Parameter: -custom-rules

Value: from-file, specify the file using Set checkers by file (-checkers-selection-file)

Default: Off

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"

Example (Code Prover): polyspace-code-prover -sources *file_name* -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a


Starting in R2019a, the file where you define custom coding rules uses the XML format. You can save selections for custom coding rules and all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your selection for each coding standard and custom coding rules in separate text files. Polyspace will stop supporting custom coding rule files in text format in a future release.

Desktop user interface:

If you have a project that contains custom coding rules and coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics**

node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename.xml*, where *filename* is the name of the first selected file alphabetically. For instance, if you select *foo.conf* and *bar.conf*, they are saved as *bar.conf.xml*.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file *custom_rules.xml* as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML*. Here, *polyspaceroot* is the root installation folder for the

Polyspace products, for instance, C:\Program Files\Polyspace\R2019a. To update your script, replace reference to the old file format with the new XML file format .

Example of Configuration File in XML Format

To turn on and define custom coding rule 8.1, use this entry:

```
<standard name="CUSTOM RULES">
  ...
  <section name="8 Constants">
    ...
    <check id="8.1" state="on">
      </check>
    ...
  </section>
  ...
</standard>
```

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)

See Also

Topics

"Prepare Scripts for Polyspace Analysis"

"Check for Coding Standard Violations"

"Create Custom Coding Rules"

"Custom Coding Rules" (Polyspace Code Prover Access)

Effective boolean types (-boolean-types)

Specify data types that coding rule checker must treat as effectively Boolean

Description

Specify data types that the coding rule checker must treat as effectively Boolean. You can specify a data type as effectively Boolean only if you have defined it through an `enum` or `typedef` statement in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-147 for other options that you must also enable.

Command line: Use the option `-boolean-types`. See “Command-Line Information” on page 2-147.

Why Use This Option

Use this option to allow Polyspace to check the following coding rules:

- MISRA C: 2004 and MISRA AC AGC

Rule Number	Rule Statement
12.6	Operands of logical operators, <code>&&</code> , <code> </code> , and <code>!</code> , should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to other operators.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.

- MISRA C: 2012

Rule Number	Rule Statement
10.1	Operands shall not be of an inappropriate essential type
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
10.5	The value of an expression should not be cast to an inappropriate essential type
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
16.7	A switch-expression shall not have essentially Boolean type.

For example, in the following code, unless you specify `myBool` as effectively Boolean, Polyspace detects a violation of MISRA C: 2012 rule 14.4.


```
typedef int myBool;
```

```
void func1(void);
void func2(void);

void func(myBool flag) {
    if(flag)
        func1();
    else
        func2();
}
```

Settings

No Default

Click  to add a field. Enter a type name that you want Polyspace to treat as Boolean.

Dependencies

This option is enabled only if you select one of these options:

- Check MISRA C:2004 (-misra2)
- Check MISRA AC AGC (-misra-ac-agc).
- Check MISRA C:2012 (-misra3)

Command-Line Information

Parameter: -boolean-types

Value: *type1[,type2[,...]]*

No Default

Example (Bug Finder): polyspace-bug-finder -sources *filename* -misra2 required-rules -boolean-types boolean1_t,boolean2_t

Example (Code Prover): polyspace-code-prover -sources *filename* -misra2 required-rules -boolean-types boolean1_t,boolean2_t

Example (Bug Finder Server): polyspace-bug-finder-server -sources *filename* -misra2 required-rules -boolean-types boolean1_t,boolean2_t

Example (Code Prover Server): polyspace-code-prover-server -sources *filename* -misra2 required-rules -boolean-types boolean1_t,boolean2_t

See Also

Check MISRA AC AGC (-misra-ac-agc) | Check MISRA C:2004 (-misra2) | Check MISRA C:2012 (-misra3)

Topics

“Prepare Scripts for Polyspace Analysis”

“Check for Coding Standard Violations”

“MISRA C:2004 Rules” (Polyspace Code Prover Access)

“MISRA C:2012 Directives and Rules” (Polyspace Code Prover Access)

Allowed pragmas (-allowed-pragmas)

Specify pragma directives that are documented

Description

Specify pragma directives that must not be flagged by MISRA C:2004 rule 3.4 or MISRA C++ rule 16-6-1. These rules require that you document all pragma directives.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-148 for other options that you must also enable.


Command line: Use the option `-allowed-pragmas`. See “Command-Line Information” on page 2-148.

Why Use This Option

MISRA C:2004/MISRA AC AGC rule 3.4 and MISRA C++ rule 16-6-1 require that all pragma directives are documented within the documentation of the compiler. If you list a pragma as documented using this analysis option, Polyspace does not flag use of the pragma as a violation of these rules.

Settings

No Default

Click  to add a field. Enter the pragma name that you want Polyspace to ignore during coding rule checking .

Dependencies

This option is enabled only if you select one of these options:

- Check MISRA C:2004 (`-misra2`)
- Check MISRA AC AGC (`-misra-ac-agc`).
- Check MISRA C++:2008 (`-misra-cpp`)

Command-Line Information

Parameter: `-allowed-pragmas`

Value: `pragma1[,pragma2[,...]]`

No Default

Example (Bug Finder): `polyspace-bug-finder -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

Example (Code Prover): `polyspace-code-prover -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

Example (Code Prover Server): `polyspace-code-prover-server -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

See Also

Check MISRA AC AGC (-misra-ac-agc) | Check MISRA C++:2008 (-misra-cpp) | Check MISRA C:2004 (-misra2)

Topics

"Prepare Scripts for Polyspace Analysis"

"Check for Coding Standard Violations"

"MISRA C:2004 Rules" (Polyspace Code Prover Access)

"MISRA C++:2008 Rules" (Polyspace Code Prover Access)

Calculate code metrics (-code-metrics)

Compute and display code complexity metrics

Description

Specify that Polyspace must compute and display code complexity metrics for your source code. The metrics include file metrics such as number of lines and function metrics such as cyclomatic complexity and estimated size of local variables.

For more information, see “Compute Code Complexity Metrics”.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line: Use the option `-code-metrics`. See “Command-Line Information” on page 2-151.

Why Use This Option

By default, Polyspace does not calculate code complexity metrics. If you want these metrics in your analysis results, before running analysis, set this option.

High values of code complexity metrics can lead to obscure code and increase chances of coding errors. Additionally, if you run a Code Prover verification on your source code, you might benefit from checking your code complexity metrics first. If a function is too complex, attempts to verify the function can lead to a lot of unproven code. For information on how to cap your code complexity metrics, see “Compute Code Complexity Metrics”.

Settings

On

Polyspace computes and displays code complexity metrics on the **Results List** pane.

Off (default)

Polyspace does not compute complexity metrics.

Tips

If you want to compute only the code complexity metrics for your code:

- In Bug Finder, disable checking of defects. See `Find defects (-checkers)`.
- In Code Prover, run verification up to the `Source Compliance Checking` phase. See `Verification level (-to)`.

A Code Prover analysis computes the stack usage metrics after the source compliance checking phase. If you stop a Code Prover verification before source compliance checking, the stack usage metrics are not reported.

Command-Line Information

Parameter: -code-metrics

Default: Off

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -code-metrics

Example (Code Prover): polyspace-code-prover -sources *file_name* -code-metrics

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -code-metrics

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -code-metrics

See Also

Topics

“Compute Code Complexity Metrics”

“Code Metrics” (Polyspace Code Prover Access)

Check MISRA C++:2008 (-misra-cpp)

Check for violations of MISRA C++ rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of MISRA C++ rules. Each value of the option corresponds to a subset of rules to check.


Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 2-153 for other options that you must also enable.

Command line: Use the option `-misra-cpp`. See “Command-Line Information” on page 2-153.

Why Use This Option

Use this option to specify the subset of MISRA C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

`required-rules`

Check required coding rules.

`all-rules`

Check required and advisory coding rules.

`SQ0-subset1`


Check only a subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C++)”.

`SQ0-subset2`

Check a subset of rules including `SQ0-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C++)”

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click , then select the rules and recommendations you want

to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to from-file, enable Set checkers by file (-checkers-selection-file).

Dependency

This option is available only if you set Source code language (-lang) to CPP or C-CPP.

For projects with mixed C and C++ code, the MISRA C++ checker analyzes only .cpp files.

Command-Line Information

Parameter: -misra-cpp

Value: required-rules | all-rules | SQ0-subset1 | SQ0-subset2 | from-file

Default: required-rules

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -misra-cpp all-rules

Example (Code Prover): polyspace-code-prover -sources *file_name* -misra-cpp all-rules

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -misra-cpp all-rules

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -misra-cpp all-rules

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a


Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics**

node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename.xml*, where *filename* is the name of the first selected file alphabetically. For instance, if you select *foo.conf* and *bar.conf*, they are saved as *bar.conf.xml*.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_cpp_2008_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in `polyspaceroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, `polyspaceroot` is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
<code>-misra-cpp "custom_standard.conf"</code>	<code>-checkers-selection-file misra_cpp_2008_rules.xml -misra-cpp from-file</code>

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="on">
      </check>
    ...
  </section>
  ...
</standard>
```

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)

See Also

Do not generate results for (`-do-not-generate-results-for`)

Topics

"Prepare Scripts for Polyspace Analysis"

“Check for Coding Standard Violations”
“MISRA C++:2008 Rules” (Polyspace Code Prover Access)

Check JSF AV C++ rules (-jsf-coding-rules)

Check for violations of JSF C++ rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of JSF AV C++ rules (JSF++:2005). Each value of the option corresponds to a subset of rules to check.


Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 2-157 for other options that you must also enable.

Command line: Use the option `-jsf-coding-rules`. See “Command-Line Information” on page 2-157.

Why Use This Option

Use this option to specify the subset of JSF C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `shall-rules`

`shall-rules`

Check all **Shall** rules. **Shall** rules are mandatory requirements and require verification.

`shall-will-rules`


Check all **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements but do not require verification.

`all-rules`

Check all **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click , then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Tips

- If your project uses a setting other than `generic` for `Compiler (-compiler)`, some rules might not be completely checked. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

For projects with mixed C and C++ code, the JSF C++ checker analyzes only `.cpp` files.

Command-Line Information

Parameter: `-jsf-coding-rules`

Value: `shall-rules` | `shall-will-rules` | `all-rules` | `from-file`

Default: `shall-rules`

Example (Bug Finder): `polyspace-bug-finder -sources file_name -jsf-coding-rules all-rules`

Example (Code Prover): `polyspace-code-prover -sources file_name -jsf-coding-rules all-rules`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -jsf-coding-rules all-rules`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -jsf-coding-rules all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file


Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click . In the **Findings selection** window, select the files then

click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename.xml*, where *filename* is the name of the first selected file alphabetically. For instance, if you select *foo.conf* and *bar.conf*, they are saved as *bar.conf.xml*.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file *StandardsConfiguration.xml* as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceserverroot\polyspace\examples\cxx\Bug_Finder_Example\sources* or *polyspaceserverroot\polyspace\examples\cxx\Code_Prover_Example\sources*. Here, *polyspaceserverroot* is the root installation folder for the Polyspace products, for instance, *C:\Program Files\Polyspace\R2019a*. To update your script, see this table

Option	Use Instead
-jsf-coding-rules "custom_standard.conf"	-checkers-selection-file "custom_standard.conf.xml" -jsf-coding-rules from-file

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="on">
      </check>
    ...
  </section>
  ...
</standard>
```

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)

See Also

Do not generate results for (-do-not-generate-results-for)

Topics

"Prepare Scripts for Polyspace Analysis"

“Check for Coding Standard Violations”
“JSF C++ Rules” (Polyspace Code Prover Access)

Verify whole application

Stop verification if sources files are incomplete and do not contain a `main` function

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that Polyspace verification must stop if a `main` function is not present in the source files.

If you select a Visual C++ setting for `Compiler (-compiler)`, you can specify which function must be considered as `main`. See `Main entry point (-main)`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: There is no corresponding command-line option. See “Command-Line Information” on page 2-160.

Settings

On

Polyspace verification stops if it does not find a `main` function in the source files.

Off (default)

Polyspace continues verification even when a `main` function is not present in the source files. If a `main` is not present, it generates a file `__polyspace_main.c` that contains a `main` function.

Tips

If you use this option, your code must contain a `main` function. Otherwise you see the error:

Error: required main procedure not found

If your code does not contain a `main` function, use the option `Verify module or library (-main-generator)` to generate a `main` function.

Command-Line Information

Unlike the user interface, by default, a verification from the command line stops if it does not find a `main` function in the source files. If you specify the option `-main-generator`, Polyspace generates a `main` if it cannot find one in the source files.

See Also

Show global variable sharing and usage only (`-shared-variables-mode`) | `Verify module or library (-main-generator)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C Application Without main Function”

“Verify C++ Classes”

Show global variable sharing and usage only (-shared-variables-mode)

Compute global variable sharing and usage without running full analysis

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify this option to run a less extensive analysis that computes the global variable sharing and usage in your entire application. The analysis does not verify your code for run-time errors. The analysis results also include coding standards violations if you enable coding standards checking, and code metrics if you enable code metrics computation.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-shared-variables-mode`. See “Command-Line Information” on page 2-163.

Why Use This Option

You can see global variable sharing and usage without running a full analysis on your entire application that includes run-time error detection. Run-time error detection on an entire application can take a long time.

Settings

On

Polyspace computes global variable sharing and usage but does not verify your code for run-time errors.

Off (default)

Polyspace runs a full analysis on your code, including run-time error detection.

Dependencies

- You can use this option only if your program contains a `main` function and you enable the option `Verify whole application` (implicitly set by default at command line).
- When you enable this option, you must also enable at least one of these options.
 - Enable automatic concurrency detection for Code Prover (`-enable-concurrency-detection`)
 - Tasks (`-entry-points`)

- Cyclic tasks (-cyclic-tasks)
- Interrupts (-interrupts)
- ARXML files selection (-autosar-multitasking)
- OIL files selection (-osek-multitasking)

Tips

- After you analyze your complete application to see global variable sharing and usage, run a component-by-component Code Prover analysis to detect run-time errors.
- In the desktop product, you can see all read and write operations on global variables in the “Variable Access” (Polyspace Code Prover) pane.
- In this less extensive analysis mode, the analysis checks for most but not all coding standards violations, and computes most but not all code metrics.

Command-Line Information

Parameter: -shared-variables-mode

Default: Off

Example (Code Prover): polyspace-code-prover -sources *file_name* -shared-variables-mode -enable-concurrency-detection

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -shared-variables-mode -enable-concurrency-detection

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2019b

Verify initialization section of code only (-init-only-mode)

Check initialization code alone for run-time errors and other issues

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must check only the section of code marked as initialization code for run-time errors and other issues.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the `main` function (only once). The initialization code starts from the beginning of `main` and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-init-only-mode`. See “Command-Line Information” on page 2-166.

Why Use This Option

Often, issues in the initialization code can invalidate the analysis of the remaining code. You can use this option to check the initialization code alone and fix the issues, and then disable this option to verify the remaining program.

For instance, in this example:

```
#include <limits.h>

int aVar;
const int aConst = INT_MAX;
int anotherVar;

int main() {
    aVar = aConst + 1;
    #pragma polyspace_end_of_init
    anotherVar = aVar - 1;
    return 0;
}
```

the overflow in the line `aVar = aConst+1` must be fixed first before the value of `aVar` is used in subsequent code.

Settings

On

Polyspace checks the code from the beginning of `main` and continues up to the pragma `polyspace_end_of_init`.

Off (default)

Polyspace checks the complete application beginning from the `main` function.

Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to C.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef-s`).

You cannot use this option with the following options:

- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

Tips

- Use this option along with the option `Check that global variables are initialized after warm reboot (-check-globals-init)` to thoroughly check the initialization code before checking the remaining program. If you use both options, the verification checks for the following:
 - Definite or possible run-time errors in the initialization code.
 - Whether all non-const global variables are initialized along all execution paths through the initialization code.
- Multitasking options are disabled if you check initialization code only because the initialization of global variables is expected to happen before the tasks (threads) begin. As a result, task bodies are not verified.

See also “Multitasking”.

- If you check initialization code only, the analysis truncates execution paths containing the pragma at the location of the pragma but continues to check other execution paths.

For instance, in this example, the pragma appears in an `if` block. A red non-initialized variable check appears on the line `int a = var` because the path containing the initialization stops at the location of the pragma. On the only other remaining path that bypasses the `if` block, the variable `var` is not initialized.

```
int var;

int func();
```

```
int main() {
    int err = func();
    if(err) {
        var = 0;
    #pragma polyspace_end_of_init
    }
    int a = var;
    return 0;
}
```

To avoid these situations, try to place the pragma outside a block. See other suggestions for placement of the pragma in the reference for `Check that global variables are initialized after warm reboot (-check-globals-init)`.

- To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

If you check initialization code only using this option, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

Command-Line Information

Parameter: `-init-only-mode`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -init-only-mode`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -init-only-mode`

See Also

`Check that global variables are initialized after warm reboot (-check-globals-init)` | `Global variable not assigned a value in initialization code`

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2020a

Verify module or library (-main-generator)

Generate a main function if source files are modules or libraries that do not contain a main

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must generate a main function if it does not find one in the source files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-main-generator`. See “Command-Line Information” on page 2-168.

For the analogous option for model generated code, see `Verify model generated code (-main-generator)`.

Why Use This Option

Use this option if you are verifying a module or library. A Code Prover analysis requires a main function. When verifying a module or library, your code might not have a main.

When you use this option, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Settings

On (default)

Polyspace generates a main function if it does not find one in the source files. The generated main:

- 1 Initializes variables specified by `Variables to initialize (-main-generator-writes-variables)`.
- 2 Before calling other functions, calls the functions specified by `Initialization functions (-functions-called-before-main)`.
- 3 In all possible orders, calls the functions specified by `Functions to call (-main-generator-calls)`.
- 4 (C++ only) Calls class methods specified by `Class (-class-analyzer)` and `Functions to call within the specified classes (-class-analyzer-calls)`.

If you do not specify the function and variable options above, the generated main:

- Initializes all global variables except those declared with keywords `const` and `static`.
- In all possible orders, calls all functions that are not called anywhere in the source files. Polyspace considers that global variables can be written between two consecutive function calls. Therefore, in each called function, global variables initially have the full range of values allowed by their type.

Off

Polyspace stops if a `main` function is not present in the source files.

Tips

- If a `main` function is present in your source files, the verification uses that `main` function, irrespective of whether you enable or disable this option.

The option is relevant only if a `main` function is not present in your source files.

- If you use the option `Verify whole application` (default on the command line), your code must contain a `main` function. Otherwise you see the error:

```
Error: required main procedure not found
```

If your code does not contain a `main` function, use this option to generate a `main` function.

- If you specify multitasking options, the verification ignores your specifications for `main` generation. Instead, the verification introduces an empty `main` function.

For more information on the multitasking options, see “Configuring Polyspace Multitasking Analysis Manually”.

Command-Line Information

Parameter: `-main-generator`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator`

See Also

`Class (-class-analyzer) | Functions to call (-main-generator-calls) | Functions to call within the specified classes (-class-analyzer-calls) | Initialization functions (-functions-called-before-main) | Variables to initialize (-main-generator-writes-variables) | Verify whole application`

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C Application Without main Function”

Main entry point (-main)

Specify a Microsoft Visual C++ extensions of main

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify the function that you want to use as main. If the function does not exist, the verification stops with an error message. Use this option to specify Microsoft Visual C++ extensions of main.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-169 for other options that you must also enable.

Command line: Use the option -main. See “Command-Line Information” on page 2-170.

Settings

Default: `_tmain`

`_tmain`

Use `_tmain` as entry point to your code.

`wmain`

Use `wmain` as entry point to your code.

`_tWinMain`

Use `_tWinMain` as entry point to your code.

`wWinMain`

Use `wWinMain` as entry point to your code.

`WinMain`

Use `WinMain` as entry point to your code.

`DllMain`

Use `DllMain` as entry point to your code.

Dependencies

This option is enabled only if you:

- Set Source code language (-lang) to CPP.
- Select Verify whole application.

Command-Line Information

Parameter: -main

Value: _tmain | wmain | _tWinMain | wWinMain | WinMain |DllMain

Example (Code Prover): polyspace-code-prover -sources *file_name* -compiler
visuall4.0 -main _tmain

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -
compiler visuall4.0 -main _tmain

See Also

Verify module or library (-main-generator) | Verify whole application

Topics

“Prepare Scripts for Polyspace Analysis”

Variables to initialize (-main-generator-writes-variables)

Specify global variables that you want the generated `main` to initialize

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify global variables that you want the generated `main` to initialize. Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-172 for other options that you must also enable.

Command line: Use the option `-main-generator-writes-variables`. See “Command-Line Information” on page 2-172.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option to specify which global variables the generated `main` must initialize.

Settings

Default:

- C code — `public`
- C++ Code — `uninit`

`uninit`

C++ Only

The generated `main` only initializes global variables that you have not initialized during declaration.

`none`

The generated `main` does not initialize global variables.


`public`

The generated `main` initializes all global variables except those declared with keywords `static` and `const`.

`all`

The generated `main` initializes all global variables except those declared with keyword `const`.

custom

The generated `main` only initializes global variables that you specify. Click  to add a field. Enter a global variable name.

Dependencies

You can use this option only if the following are true:

- Your code does not contain a `main` function.
- `Verify module or library (-main-generator)` is selected.

The option is disabled if you enable the option `Ignore default initialization of global variables (-no-def-init-glob)`. Global variables are considered as uninitialized until you explicitly initialize them in the code.

Tips

This option only affects global variables that are defined in the project. If a global variable is declared as `extern`, the analysis considers that the variable can have any value allowed by its data type, irrespective of the value of this option.

Command-Line Information

Parameter: `-main-generator-writes-variables`

Value: `uninit | none | public | all | custom=variable1[,variable2[,...]]`

Default: (C) `public` | (C++) `uninit`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -main-generator-writes-variables all`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -main-generator-writes-variables all`

See Also

`Verify module or library (-main-generator)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C Application Without main Function”

Initialization functions (-functions-called-before-main)

Specify functions that you want the generated main to call ahead of other functions

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that you want the generated main to call ahead of other functions.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-174 for other options that you must also enable.

Command line: Use the option -functions-called-before-main. See “Command-Line Information” on page 2-174.

Why Use This Option



If you are verifying a module or library, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Use this option along with the option Functions to call (-main-generator-calls) to specify which functions the generated main must call. Unless a function is called directly or indirectly from main, the software does not analyze the function.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

If the function or method is not overloaded, specify the function name. Otherwise, specify the function prototype with arguments. For instance, in the following code, you must specify the prototypes `func(int)` and `func(double)`.

```
int func(int x) {
    return(x * 2);
}
double func(double x) {
    return(x * 2);
}
```

For C++, if the function is:

- A class method: The generated `main` calls the class constructor before calling this function.
- Not a class method: The generated `main` calls this function before calling class methods.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Dependencies

This option is enabled only if you select **Verify module or library** under **Code Prover Verification** and your code does not contain a `main` function.

Tips

Although these functions are called ahead of other functions, they can be called in arbitrary order. If you want to call your initialization functions in a specific order, manually write a `main` function to call them.

Command-Line Information

Parameter: `-functions-called-before-main`

Value: `function1[,function2[,...]]`

No Default

Example 1 (Code Prover): `polyspace-code-prover -sources file_name -main-generator -functions-called-before-main myfunc`

Example 2 (Code Prover): `polyspace-code-prover -sources file_name -main-generator -functions-called-before-main myClass::init(int)`

Example 1 (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -functions-called-before-main myfunc`

Example 2 (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -functions-called-before-main myClass::init(int)`

See Also

`Class (-class-analyzer)` | `Functions to call (-main-generator-calls)` | `Functions to call within the specified classes (-class-analyzer-calls)` | `Variables to initialize (-main-generator-writes-variables)` | `Verify module or library (-main-generator)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C Application Without main Function”

“Verify C++ Classes”

Functions to call (-main-generator-calls)

Specify functions that you want the generated main to call after the initialization functions

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that you want the generated main to call. The main calls these functions after the ones you specify through the option `Initialization functions (-functions-called-before-main)`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-176 for other options that you must also enable.

Command line: Use the option `-main-generator-calls`. See “Command-Line Information” on page 2-176.

Why Use This Option

If you are verifying a module or library, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Use this option along with the option `Initialization functions (-functions-called-before-main)` to specify which functions the generated main must call. Unless a function is called directly or indirectly from main, the software does not analyze the function.

Settings

Default: unused

none

The generated main does not call any function.

unused

The generated main calls only those functions that are not called in the source code. It does not call inlined functions.


all


The generated main calls all functions except inlined ones.

custom

The generated main calls functions that you specify.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.

- Click  to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

This option is available only if you select `Verify module or library (-main-generator)`.

Tips

- Select `unused` when you use **Code Prover Verification > Verify files independently**.
- If you want the generated `main` to call an inlined function, select `custom` and specify the name of the function.
- To verify a multitasking application without a `main`, select `none`.
- The generated `main` can call the functions in arbitrary order. If you want to call your functions in a specific order, manually write a `main` function to call them.
- To specify instantiations of templates as arguments, run analysis once with the option argument `all`. Search for the template name in the analysis log and use the template name as it appears in the analysis log for the option argument.

For instance, to specify this template function instantiation as option argument:

```
template <class T>
T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
}
template int GetMax<int>(int, int); // explicit instantiation
```

Run an analysis with the option `-main-generator-calls all`. Search for `getMax` in the analysis log. You see the function format:

```
T1 getMax<int>(T1, T1)
```

To call only this template instantiation, remove the space between the arguments and use the option:

```
-main-generator-calls custom="T1 getMax<int>(T1,T1)"
```

Command-Line Information

Parameter: `-main-generator-calls`

Value: `none | unused | all | custom=function1[,function2[,...]]`

Default: `unused`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -main-generator-calls all`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -main-generator-calls all`

See Also

Class (-class-analyzer) | Functions to call within the specified classes (-class-analyzer-calls) | Initialization functions (-functions-called-before-main) | Verify module or library (-main-generator)

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C Application Without main Function”

Verify files independently (-unit-by-unit)

Verify each source file independently of other source files

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that each source file must be verified independently of other source files. Each file is verified individually, independent of other files in the module. Verification results can be viewed for the entire project or for individual files.

After you open the verification result for one file, in the user interface of the Polyspace desktop products, you can see a summary of results for all files on the **Dashboard** pane. You can open the results for each file directly from this summary table.

Each result file (with name `ps_results.pscp`) is saved in a subfolder of the results folder. The subfolder has the same name as the source file being analyzed.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-178 for other options that you must also enable.

Command line: Use the option `-unit-by-unit`. See “Command-Line Information” on page 2-179.

Why Use This Option

There are many reasons you might want to verify each source file independently of other files.

For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

Settings

On

Polyspace creates a separate verification job for each source file.

Off (default)

Polyspace creates a single verification job for all source files in a module.

Dependencies

This option is enabled only if you select `Verify module` or `library (-main-generator)`.

Tips

- Code Prover requires a `main` function as the starting point of verification. In the file-by-file mode, because most files do not have a `main`, Code Prover generates a `main` function when required. By default, the generated `main` calls uncalled functions (uncalled non-private methods and out-of-class functions in C++). For more information, see:
 - “Verify C Application Without main Function”
 - “Verify C++ Classes”
- If you perform a file by file verification, you cannot specify multitasking options.
- If your verification for the entire project takes very long, perform a file by file verification. After the verification is complete for a file, you can view the results while other files are still being verified.
- You can generate a report of the verification results for each file or for all the files together. To generate a single report for all files, perform the report generation after verification (and not along with verification using analysis options).

If you use the product Polyspace Code Prover Server to run a verification, to generate a single report for all files:

- Upload the results for all files to the Polyspace Access server.
- Use the `polyspace-report-generator` command with option `-all-units` to generate a single report for all the files.
- When you perform a file-by-file verification, you can see many instances of unused variables. Some of these variables might be used in other files but show as unused in a file-by-file verification.

Command-Line Information

Parameter: `-unit-by-unit`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -unit-by-unit`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -unit-by-unit`

See Also

Common source files (`-unit-by-unit-common-source`)

Topics

“Prepare Scripts for Polyspace Analysis”

Common source files (-unit-by-unit-common-source)

Specify files that you want to include with each source file during a file by file verification

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

For a file by file verification, specify files that you want to include with each source file verification. These files are compiled once, and then linked to each verification.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-180 for other options that you must also enable.

Command line: Use the option `-unit-by-unit-common-source`. See “Command-Line Information” on page 2-181.



Why Use This Option

There are many reasons you might want to verify each source file independently of other files. For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

If you perform a file by file verification, some of your files might be missing information present in the other files. Place the missing information in a common file and use this option to specify the file for verification. For instance, if multiple source files call the same function, use this option to specify a file that contains the function definition or a function stub. Otherwise, Polyspace uses its own stubs for functions that are called but not defined in the source files. The assumptions behind the Polyspace stubs can be broader than what you want, leading to orange checks.

Settings

No Default

Click  to add a field. Enter the full path to a file. Otherwise, use the  button to navigate to the file location.

Dependencies

This option is enabled only if you select `Verify files independently (-unit-by-unit)`.

Command-Line Information

Parameter: -unit-by-unit-common-source

Value: *file1[,file2[,...]]*

No Default

Example (Code Prover): polyspace-code-prover -sources *file_name* -unit-by-unit -unit-by-unit-common-source definitions.c

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -unit-by-unit -unit-by-unit-common-source definitions.c

See Also

Verify files independently (-unit-by-unit)

Topics

“Prepare Scripts for Polyspace Analysis”

Verify model generated code (-main-generator)

Specify that a main function must be generated if it is not present in source files

Description

In Bug Finder, use this option only for code generated from MATLAB code or Simulink models.

Specify that Polyspace must generate a main function if it does not find one in the source files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-main-generator`. See “Command-Line Information” on page 2-182.

Settings

This option is always enabled for code generated from models.

Polyspace generates a main function for the analysis. The generated main contains cyclic code that executes in a loop. The loop can run an unspecified number of times.

The main performs the following functions before the loop begins:

- Initializes variables specified by `Parameters (-variables-written-before-loop)`.
- Calls the functions specified by `Initialization functions (-functions-called-before-loop)`.

The main then performs the following functions in the loop:

- Calls the functions specified by `Step functions (-functions-called-in-loop)`.
- Writes to variables specified by `Inputs (-variables-written-in-loop)`.

Finally, the main calls the functions specified by `Termination functions (-functions-called-after-loop)`.

Command-Line Information

Parameter: `-main-generator`

Default: On

Example (Bug Finder): `polyspace-bug-finder -sources file_name -main-generator ...`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator ...`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -main-generator ...`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator ...`

See Also

Initialization functions (-functions-called-before-loop) | Inputs (-variables-written-in-loop) | Parameters (-variables-written-before-loop) | Step functions (-functions-called-in-loop) | Termination functions (-functions-called-after-loop) | Verify model generated code (-main-generator)

Parameters (-variables-written-before-loop)

Specify variables that the generated main must initialize before the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify variables that the generated main must initialize before the cyclic code loop begins. Before the loop begins, Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-variables-written-before-loop`. See “Command-Line Information” on page 2-184.

Settings

Default: none


none

The generated main does not initialize variables.

all

The generated main initializes all variables except those declared with keyword `const`.

custom

The generated main only initializes variables that you specify. Click  to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-before-loop`

Value: `none | all | custom=variable1[,variable2[,...]]`

Default: none

Example (Bug Finder): `polyspace-bug-finder -sources file_name -main-generator -variables-written-before-loop all`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -variables-written-before-loop all`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -main-generator -variables-written-before-loop all`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -variables-written-before-loop all`

See Also

Inputs (-variables-written-in-loop) | Verify model generated code (-main-generator)

Inputs (-variables-written-in-loop)

Specify variables that the generated main must initialize in the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify variables that the generated main must initialize at the beginning of every iteration of the cyclic code loop. At the beginning of every loop iteration, Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-variables-written-in-loop`. See “Command-Line Information” on page 2-185.

Settings

Default: none


none

The generated main does not initialize variables.

all

The generated main initializes all variables except those declared with keyword `const`.

custom

The generated main only initializes variables that you specify. Click  to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-in-loop`

Value: `none | all | custom=variable1[,variable2[,...]]`

Default: none

Example (Bug Finder): `polyspace-bug-finder -sources file_name -main-generator -variables-written-in-loop all`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -variables-written-in-loop all`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -main-generator -variables-written-in-loop all`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -variables-written-in-loop all`

See Also

Parameters (-variables-written-before-loop) | Verify model generated code (-main-generator)

Initialization functions (-functions-called-before-loop)

Specify functions that the generated main must call before the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated main must call before the cyclic code begins.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-before-loop`. See “Command-Line Information” on page 2-187.

Settings

No Default

Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Tips

- If you specify a function for the option **Termination functions (-functions-called-after-loop)**, you cannot specify it for this option.

Command-Line Information

Parameter: `-functions-called-before-loop`

No Default

Value: `function1[,function2[,...]]`

Example (Bug Finder): `polyspace-bug-finder -sources file_name -main-generator -functions-called-before-loop myfunc`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -functions-called-before-loop myfunc`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -main-generator -functions-called-before-loop myfunc`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -functions-called-before-loop myfunc`

See Also

Step functions (-functions-called-in-loop) | Termination functions (-functions-called-after-loop) | Verify model generated code (-main-generator)

Step functions (-functions-called-in-loop)

Specify functions that the generated main must call in the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated main must call in each cycle of the cyclic code.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-in-loop`. See “Command-Line Information” on page 2-189.

Settings

Default: none


none

The generated main does not call functions in the cyclic code.

all

The generated main calls all functions except inlined ones. If you specify certain functions for the options **Initialization functions** or **Termination functions**, the generated main does not call those functions in the cyclic code.

custom

The generated main calls functions that you specify. Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Tips

If you have specified a function for the option **Initialization functions** (`-functions-called-before-loop`) or **Termination functions** (`-functions-called-after-loop`), to call it inside the cyclic code, use `custom` and specify the function name.

Command-Line Information

Parameter: `-functions-called-in-loop`

Value: `none | all | custom=function1[,function2[,...]]`

Default: none

Example (Bug Finder): `polyspace-bug-finder -sources file_name -main-generator -functions-called-in-loop all`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -functions-called-in-loop all`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -main-generator -functions-called-in-loop all`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -functions-called-in-loop all`

See Also

Initialization functions (-functions-called-before-loop) | Termination functions (-functions-called-after-loop) | Verify model generated code (-main-generator)

Termination functions (-functions-called-after-loop)

Specify functions that the generated main must call after the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated main must call after the cyclic code ends.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-after-loop`. See “Command-Line Information” on page 2-191.

Settings

No Default

Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Tips

- If you specify a function for the option **Initialization functions (-functions-called-before-loop)**, you cannot specify it for this option.

Command-Line Information

Parameter: `-functions-called-after-loop`

No Default

Value: `function1[,function2[,...]]`

Example (Bug Finder): `polyspace-bug-finder -sources file_name -main-generator -functions-called-after-loop myfunc`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -functions-called-after-loop myfunc`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -main-generator -functions-called-after-loop myfunc`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -functions-called-after-loop myfunc`

See Also

Initialization functions (-functions-called-before-loop) | Step functions (-functions-called-in-loop) | Verify model generated code (-main-generator)

Class (-class-analyzer)

Specify classes that you want to verify

Description

This option affects a Code Prover analysis only.

Specify classes that Polyspace uses to generate a main.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-193 for other options that you must also enable.

Command line: Use the option `-class-analyzer`. See “Command-Line Information” on page 2-194.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Functions to call within the specified classes (-class-analyzer-calls)` to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

Settings

Default: all

all

Polyspace can use all classes to generate a `main`. The generated `main` calls methods that you specify using **Functions to call within the specified classes**.

none

The generated `main` cannot call any class method.

custom

Polyspace can use classes that you specify to generate a `main`. The generated `main` calls methods from classes that you specify using **Functions to call within the specified classes**.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- Source code language (`-lang`) is set to CPP or C-CPP.
- Verify module or library (`-main-generator`) is selected.

Tips

If you select none for this option, Polyspace will not verify class methods that you do not call explicitly in your code.

Command-Line Information

Parameter: `-class-analyzer`

Value: `all` | `none` | `custom=class1[,class2,...]`

Default: `all`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -class-analyzer custom=myClass1,myClass2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -class-analyzer custom=myClass1,myClass2`

See Also

Analyze class contents only (`-class-only`) | Functions to call within the specified classes (`-class-analyzer-calls`) | Skip member initialization check (`-no-constructors-init-check`) | Verify module or library (`-main-generator`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C++ Classes”

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

Specify class methods that you want to verify

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify class methods that Polyspace uses to generate a `main`. The generated `main` can call static, public and protected methods in classes that you specify using the **Class** option.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-196 for other options that you must also enable.

Command line: Use the option `-class-analyzer-calls`. See “Command-Line Information” on page 2-196.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Class` (`-class-analyzer`) to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

Settings

Default: unused

`all`

The generated `main` calls all public and protected methods. It does not call methods inherited from a parent class.

`all-public`

The generated `main` calls all public methods. It does not call methods inherited from a parent class.

`inherited-all`

The generated `main` calls all public and protected methods including those inherited from a parent class.

`inherited-all-public`

The generated `main` calls all public methods including those inherited from a parent class.

unused

The generated `main` calls public and protected methods that are not called in the code.

unused-public

The generated `main` calls public methods that are not called in the code. It does not call methods inherited from a parent class.

inherited-unused

The generated `main` calls public and protected methods that are not called in the code including those inherited from a parent class.



inherited-unused-public

The generated `main` calls public methods that are not called in the code including those inherited from a parent class.

custom

The generated `main` calls the methods that you specify.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

You can use this option only if:

- Source code language (`-lang`) is set to CPP or C-CPP.
- Verify module or library (`-main-generator`) is selected.

Command-Line Information

Parameter: `-class-analyzer-calls`

Value: `all|all-public|inherited-all|inherited-all-public|unused|unused-public|inherited-unused|inherited-unused-public|custom=method1[,method2,...]`

Default: `unused`

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`

See Also

`Class (-class-analyzer)|Verify module or library (-main-generator)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C++ Classes”

Analyze class contents only (-class-only)

Do not analyze code other than class methods

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that Polyspace must verify only methods of classes that you specify using the option `Class` (-class-analyzer).

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-198 for other options that you must also enable.

Command line: Use the option `-class-only`. See “Command-Line Information” on page 2-199.

Why Use This Option

Use this option to restrict the analysis to certain class methods only.

You specify these methods through the options:

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

When you analyze a module or library, Code Prover generates a `main` function if one does not exist. The `main` function calls class methods using these two options and functions that are not class methods using other options. Code Prover analyzes these methods and functions for robustness to all inputs. If you use this option, Code Prover analyzes the methods only.

Settings

On

Polyspace verifies the class methods only. It stubs functions out of class scope even if the functions are defined in your code.

Off (default)

Polyspace verifies functions out of class scope in addition to class methods.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language` (-lang) is set to `CPP` or `C-CPP`.

- Verify module or library (-main-generator) is selected.

If you select this option, you must specify the classes using the Class (-class-analyzer) option.

Tips

Use this option:

- For robustness verification of class methods. Unless you use this option, Polyspace verifies methods that you call in your code only for your input combinations.
- In case of scaling.

Command-Line Information

Parameter: -class-only

Default: Off

Example (Code Prover): polyspace-code-prover -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only

See Also

Class (-class-analyzer) | Functions to call within the specified classes (-class-analyzer-calls) | Verify module or library (-main-generator)

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C++ Classes”

Skip member initialization check (`-no-constructors-init-check`)

Do not check if class constructor initializes class members

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must not check whether each class constructor initializes all class members.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-200 for other options that you must also enable.

Command line: Use the option `-no-constructors-init-check`. See “Command-Line Information” on page 2-201.

Why Use This Option

Use this option to disable checks for initialization of class members in constructors.

Settings

On

Polyspace does not check whether each class constructor initializes all class members.

Off (default)

Polyspace checks whether each class constructor initializes all class members. It uses the functions `check_NIV()` and `check_NIP()` in the generated `main` to perform these checks. It checks for initialization of:

- Integer types such as `int`, `char` and `enum`, both signed or unsigned.
- Floating-point types such as `float` and `double`.
- Pointers.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- Source code language (`-lang`) is set to `CPP` or `C-CPP`.
- Verify module or library (`-main-generator`) is selected.

If you select this option, you must specify the classes using the `theClass` (`-class-analyzer`) option.

Command-Line Information

Parameter: -no-constructors-init-check

Default: Off

Example (Code Prover): polyspace-code-prover -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -no-constructors-init-check

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -no-constructors-init-check

See Also

Class (-class-analyzer) | Verify module or library (-main-generator)

Topics

“Prepare Scripts for Polyspace Analysis”

“Verify C++ Classes”

Respect types in fields (-respect-types-in-fields)

Do not cast nonpointer fields of a structure to pointers

Description

This option affects a Code Prover analysis only.

Specify that structure fields not declared initially as pointers will not be cast to pointers later.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-respect-types-in-fields`. See “Command-Line Information” on page 2-203.

Why Use This Option

Use this option to identify and forbid casts from nonpointer structure fields to pointers.

Settings

On

The verification assumes that structure fields not declared initially as pointers will not be cast to pointers later.

Code with option off	Code with option on
<pre>struct { unsigned int x1; unsigned int x2; } S; void funct(void) { int var, *tmp; S.x1 = &var; tmp = (int*)S.x1; *tmp = 1; assert(var==1); }</pre> <p>In this example, the fields of S are declared as integers but S.x1 is cast to a pointer. With the option turned off, Polyspace allows the cast.</p>	<pre>struct { unsigned int x1; unsigned int x2; } S; void funct(void) { int var, *tmp; S.x1 = &var; tmp = (int*)S.x1; *tmp = 1; assert(var==1); }</pre> <p>In this example, the fields of S are declared as integers but S.x1 is cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of var through the pointer (int*)S.x1 and produces a red Non-initialized local variable error when var is read.</p>

Off (default)

The verification assumes that structure fields can be cast to pointers even when they are not declared as pointers.

Command-Line Information

Parameter: -respect-types-in-fields

Default: Off

Example (Code Prover): polyspace-code-prover -sources *file_name* -respect-types-in-fields

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -respect-types-in-fields

See Also

Non-initialized local variable | Respect types in global variables (-respect-types-in-globals)

Topics

“Prepare Scripts for Polyspace Analysis”

Respect types in global variables (-respect-types-in-globals)

Do not cast nonpointer global variables to pointers

Description

This option affects a Code Prover analysis only.

Specify that global variables not declared initially as pointers will not be cast to pointers later.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-respect-types-in-globals`. See “Command-Line Information” on page 2-205.

Why Use This Option

Use this option to identify and forbid casts from nonpointer global variables to pointers.

Settings

On

The verification assumes that global variables not declared initially as pointers will not be cast to pointers later.

Off (default)

The verification assumes that global variables can be cast to pointers even when they are not declared as pointers.

Tips

If you select this option, the number of checks in your code can change. You can use this option and the change in results to identify cases where you cast nonpointer variables to pointers.

For instance, in the following example, when you select the option, the results have one less orange check and one more red check.

Code with option off	Code with option on
<pre data-bbox="240 296 857 499">int global; void main(void) { int local; global = (int)&local; *(int*)global = 5; assert(local==5); }</pre> <p data-bbox="240 520 857 619">In this example, <code>global</code> is declared as an <code>int</code> variable but cast to a pointer. With the option turned off, Polyspace allows the cast.</p>	<pre data-bbox="857 296 1474 499">int global; void main(void) { int local; global = (int)&local; *(int*)global = 5; assert(local==5); }</pre> <p data-bbox="857 520 1474 745">In this example, <code>global</code> is declared as an <code>int</code> variable but cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of <code>local</code> through the pointer <code>(int*)global</code> and produces a red Non-initialized local variable error when <code>local</code> is read.</p>

Command-Line Information

Parameter: `-respect-types-in-globals`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -respect-types-in-globals`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -respect-types-in-globals`

See Also

Non-initialized local variable | Respect types in fields (`-respect-types-in-fields`)

Topics

“Prepare Scripts for Polyspace Analysis”

Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)

Specify that environment pointers can be unsafe to dereference unless constrained otherwise

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that the verification must consider environment pointers as unsafe unless otherwise constrained. Environment pointers are pointers that can be assigned values outside your code.

Environment pointers include:

- Global or extern pointers.
- Pointers returned from stubbed functions.

A function is stubbed if your code does not contain the function definition or you override a function definition by using the option `Functions to stub (-functions-to-stub)`.

- Pointer parameters of functions whose calls are generated by the software.

A function call is generated if you verify a module or library and the module or library does not have an explicit call to the function. You can also force a function call to be generated with the option `Functions to call (-main-generator-calls)`.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-stubbed-pointers-are-unsafe`. See “Command-Line Information” on page 2-208.

Why Use This Option

Use this option so that the verification makes more conservative assumptions about pointers from external sources.

If you specify this option, the verification considers that environment pointers can have a NULL value. If you read an environment pointer without checking for NULL, the **Illegally dereferenced pointer** check shows a potential error in orange. The message associated with the orange check shows the pointer can be NULL.

Settings

On

The verification considers that environment pointers can have a NULL value.

Off (default)

The verification considers that environment pointers:

- Cannot have a NULL value.
- Points within allowed bounds.

Tips

- Enable this option during the integration phase. In this phase, you provide complete code for verification. Even if an orange check originates from external sources, you are likely to place protections against unsafe pointers from such sources. For instance, if you obtain a pointer from an unknown source, you check the pointer for NULL value.

Disable this option during the unit testing phase. In this phase, you focus on errors originating from your unit.

- If you are verifying code implementation of AUTOSAR runnables, Code Prover assumes that pointer arguments to runnables and pointers returned from Rte_ functions are not NULL. You cannot use this option to change the assumption. See “Run Polyspace on AUTOSAR Code with Conservative Assumptions” (Polyspace Code Prover).
- If you enable this option, the number of orange checks in your code might increase.

Environment Pointers Safe	Environment Pointers Unsafe
<p>The Illegally dereferenced pointer check is green. The verification assumes that <code>env_ptr</code> is not NULL and any dereference is within allowed bounds. The verification assumes that the result of the dereference is full range. For instance, in this case, the return value has the full range of type <code>int</code>.</p> <pre>int func (int *env_ptr) { return *env_ptr; }</pre>	<p>The Illegally dereferenced pointer check is orange. The verification assumes that <code>env_ptr</code> can be NULL.</p> <pre>int func (int *env_ptr) { return *env_ptr; }</pre>

If you enable this option, the number of gray checks might decrease.

Environment Pointers Safe	Environment Pointers Unsafe
<p>The verification assumes that <code>env_ptr</code> is not NULL. The <code>if</code> condition is always true and the <code>else</code> block is unreachable.</p> <pre>#include <stdlib.h> int func (int *env_ptr) { if(env_ptr!=NULL) return *env_ptr; else return 0; }</pre>	<p>The verification assumes that <code>env_ptr</code> can be NULL. The <code>if</code> condition is not always true and the <code>else</code> block can be reachable.</p> <pre>#include <stdlib.h> int func (int *env_ptr) { if(env_ptr!=NULL) return *env_ptr; else return 0; }</pre>

- Instead of considering all environment pointers as safe or unsafe, you can individually constrain some of the environment pointers. See the description of **Initialize Pointer** in “External Constraints for Polyspace Analysis” (Polyspace Code Prover).

When you individually constrain a pointer, you first specify an **Init Mode**, and then specify through the **Initialize Pointer** option whether the pointer is `Null`, `Not Null`, or `Maybe Null`. Depending on the **Init Mode**, you can either override the global specification for all environment pointers or not.

- If you set the **Init Mode** of the pointer to `INIT` or `PERMANENT`, your selection for **Initialize Pointer** overrides your specification for this option. For instance, if you specify `Not NULL` for an environment pointer `ptr`, the verification assumes that `ptr` is not `NULL` even if you specify that environment pointers must be considered unsafe.
- If you set the **Init Mode** to `MAIN GENERATOR`, the verification uses your specification for this option.

For pointers returned from stubbed functions, the option `MAIN GENERATOR` is not available. If you override the global specification for such a pointer through the **Initialize Pointer** option in constraints, you cannot toggle back to the global specification without changing the **Initialize Pointer** option too.

- If you disable this option, the verification considers that dereferences at all pointer depths are valid.

For instance, all the dereferences are considered valid in this code:

```
int*** stub(void);

void func2() {
    int ***ptr = stub();
    int **ptr2 = *ptr;
    int *ptr3 = *ptr2;
}
```

Command-Line Information

Parameter: `-stubbed-pointers-are-unsafe`

Default: `Off`

Example (Code Prover): `polyspace-code-prover -sources file_name -stubbed-pointers-are-unsafe`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -stubbed-pointers-are-unsafe`

See Also

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

Topics

“Prepare Scripts for Polyspace Analysis”

“Specify External Constraints”

“External Constraints for Polyspace Analysis”

Introduced in R2016b

Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)

Assume that `volatile` qualified structure fields can have all possible values at any point in code

Description

This option affects a Code Prover analysis only.

Specify that the verification must take into account the `volatile` qualifier on fields of a structure.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-consider-volatile-qualifier-on-fields`. See “Command-Line Information” on page 2-211.

Why Use This Option

The `volatile` qualifier on a variable indicates that the variable value can change between successive operations even if you do not explicitly change it in your code. For instance, if `var` is a `volatile` variable, the consecutive operations `res = var;` `res =var;` can result in two different values of `var` being read into `res`.

Use this option so that the verification emulates the `volatile` qualifier for structure fields. If you select this option, the software assumes that a `volatile` structure field has a full range of values at any point in the code. The range is determined only by the data type of the structure field.

Settings

On

The verification considers the `volatile` qualifier on fields of a structure.

In the following example, the verification considers that the field `val1` can have all values allowed for the `int` type at any point in the code.

```
struct myStruct {
    volatile int val1;
    int val2;
};
```

Even if you write a specific value to `val1` and read the variable in the next operation, the variable read results in any possible value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion can fail
```

Off (default)

The verification ignores the `volatile` qualifier on fields of a structure.

In the following example, the verification ignores the qualifier on field `val1`.

```
struct myStruct {
    volatile int val1;
    int val2;
};
```

If you write a specific value to `val1` and read the variable in the next operation, the variable read results in that specific value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion passes
```

Tips

- If your volatile fields do not represent values read from hardware and you do not expect their values to change between successive operations, disable this option. You are using the `volatile` qualifier for some other reason and the verification does not need to consider full range for the field values.
- If you enable this option, the number of red, gray, and green checks in your code can decrease. The number of orange checks can increase.

In the following example, a red or green check changes to orange or a gray check goes away when the option is used. Considering the `volatile` qualifier changes the check color. These examples use the following structure definition:

```
struct myStruct {
    volatile int field1;
    int field2;
};
```

Color Without Option	Result Without Option	Result With Option
Green	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 == 1); }</pre>	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 ==1); }</pre>
Red	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 != 1); }</pre>	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 !=1); }</pre>

Color Without Option	Result Without Option	Result With Option
Gray	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; if (structVal.field1 != 1) { /* Perform operation */ } }</pre>	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; if (structVal.field1 != 1) { /* Perform operation */ } }</pre>

- In C++ code, the option also applies to class members.

Command-Line Information

Parameter: -consider-volatile-qualifier-on-fields

Default: Off

Example (Code Prover): polyspace-code-prover -sources *file_name* -consider-volatile-qualifier-on-fields

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -consider-volatile-qualifier-on-fields

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016b

Float rounding mode (`-float-rounding-mode`)

Specify rounding modes to consider when determining the results of floating point arithmetic

Description

This option affects a Code Prover analysis only.

Specify the rounding modes to consider when determining the results of floating-point arithmetic.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-float-rounding-mode`. See “Command-Line Information” on page 2-214.

Why Use This Option

The default verification uses the round-to-nearest mode.

Use the rounding mode `all` if your code contains routines such as `fesetround` to specify a rounding mode other than round-to-nearest. Although the verification ignores the `fesetround` specification, it considers all rounding modes including the rounding mode that you specified. Alternatively, for targets that can use extended precision (for instance, using the flag `-mfpmath=387`), use the rounding mode `all`. However, for your Polyspace analysis results to agree with run-time behavior, you must prevent use of extended precision through a flag such as `-ffloat-store`.

Otherwise, continue to use the default rounding mode `to-nearest`. Because all rounding modes are considered when you specify `all`, you can have many orange **Overflow** checks resulting from overapproximation.

Settings

Default: `to-nearest`

`to-nearest`

The verification assumes the round-to-nearest mode.

`all`

The verification assumes all rounding modes for each operation involving floating-point variables. The following rounding modes are considered: round-to-nearest, round-towards-zero, round-towards-positive-infinity, and round-towards-negative-infinity.

Tips

- The Polyspace analysis uses floating-point arithmetic that conforms to the IEEE® 754 standard. For instance, the arithmetic uses floating point instructions present in the SSE instruction set. The GNU C flag `-mfpmath=sse` enforces use of this instruction set. If you use the GNU C compiler

with this flag to compile your code, your Polyspace analysis results agree with your run-time behavior.

However, if your code uses extended precision, for instance using the GNU C flag `-mfpmath=387`, your Polyspace analysis results might not agree with your run-time behavior in some corner cases. See some examples of these corner cases in `codeprover_limitations.pdf` in `polyspaceroot\polyspace\verifier\code_prover_desktop`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

To prevent use of extended precision, on targets without SSE support, you can use a flag such as `-ffloat-store`. For your Polyspace analysis, use `all` for rounding mode to account for double rounding.

- The **Overflow** check uses the rounding modes that you specify. For instance, the following table shows the difference in the result of the check when you change your rounding modes.

Rounding mode: to-nearest	Rounding mode: all
<p>If results of floating-point operations are rounded to nearest values:</p> <ul style="list-style-type: none"> • In the first addition operation, <code>eps1</code> is just large enough that the value nearest to <code>FLT_MAX + eps1</code> is greater than <code>FLT_MAX</code>. The Overflow check is red. • In the second addition operation, <code>eps2</code> is just small enough that the value nearest to <code>FLT_MAX + eps2</code> is <code>FLT_MAX</code>. The Overflow check is green. 	<p>Besides to-nearest mode, the Overflow check also considers other rounding modes.</p> <ul style="list-style-type: none"> • In the first addition operation, in to-nearest mode, the value nearest to <code>FLT_MAX + eps1</code> is greater than <code>FLT_MAX</code>, so the addition overflows. But if rounded towards negative infinity, the result is <code>FLT_MAX</code>, so the addition does not overflow. Combining these two rounding modes, the Overflow check is orange. • In the second addition operation, in to-nearest mode, the value nearest to <code>FLT_MAX + eps2</code> is <code>FLT_MAX</code>, so the addition does not overflow. But if rounded towards positive infinity, the result is greater than <code>FLT_MAX</code>, so the addition overflows. Combining these two rounding modes, the Overflow check is orange.
<pre>#include <float.h> #define eps1 0x1p103 #define eps2 0x0.FFFFFFFp103 float func(int ch) { float left_op = FLT_MAX; float right_op_1 = eps1, \ right_op_2 = eps2; switch(ch) { case 1: return (left_op +\ right_op_1); case 2: return (left_op +\ right_op_2); default: return 0; } }</pre>	<pre>#include <float.h> #define eps1 0x1p103 #define eps2 0x0.FFFFFFFp103 float func(int ch) { float left_op = FLT_MAX; float right_op_1 = eps1, \ right_op_2 = eps2; switch(ch) { case 1: return (left_op +\ right_op_1); case 2: return (left_op +\ right_op_2); default: return 0; } }</pre>

If you set the rounding mode to `all` and obtain an orange **Overflow** check, to determine how the overflow can occur, consider all rounding modes.

Command-Line Information

Parameter: `-float-rounding-mode`

Value: `to-nearest|all`

Default: `to-nearest`

Example (Code Prover): `polyspace-code-prover -sources file_name -float-rounding-mode all`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -float-rounding-mode all`

See Also

Overflow

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016a

Allow negative operand for left shifts (`-allow-negative-operand-in-shift`)

Allow left shift operations on a negative number

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow left shift operations on a negative number.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-allow-negative-operand-in-shift`. See “Command-Line Information” on page 2-216.

Why Use This Option

According to the C99 standard (sec 6.5.7), the result of a left shift operation on a negative number is undefined. Following the standard, the verification produces a red check on left shifts of negative numbers.

If your compiler has a well-defined behavior for left shifts of negative numbers, set this option. Note that allowing left shifts of negative numbers can reduce the cross-compiler portability of your code.

Settings

On

The verification allows shift operations on a negative number, for instance, `-2 << 2`.

Off (default)

If a shift operation is performed on a negative number, the verification generates an error.

Command-Line Information

Parameter: `-allow-negative-operand-in-shift`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -allow-negative-operand-in-shift`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -allow-negative-operand-in-shift`

See Also

Invalid shift operations

Topics

“Prepare Scripts for Polyspace Analysis”

Overflow mode for signed integer (-signed-integer-overflows)

Specify whether result of overflow is wrapped around or truncated

Description

This option affects a Code Prover analysis only.

Specify whether Polyspace flags signed integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node under **Code Prover Verification**.

Command line: Use the option `-signed-integer-overflows`. See “Command-Line Information” (Polyspace Code Prover).

Why Use This Option

Use this option to specify whether to check for signed integer overflows and to specify the assumptions the analysis makes following an overflow.

Settings

Default: forbid

forbid

Polyspace flags signed integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:
 - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
 - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

This behavior conforms to the ANSI C (ISO C++) standard.

In the following code, `j` has values in the range $[1..2^{31}-1]$ before the orange overflow. Polyspace considers that `j` has even values in the range $[2..2147483646]$ after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // Result of * operation overflows
    i = i * 2;
    // Remaining code in current scope not analyzed
    printf("%d", i);
}

void func2()
{
    int j = getVal();
    if (j > 0) {
        // Range of j: [1..231-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 2147483646]
        printf("%d", j);
    }
}
```

allow

Polyspace does not flag signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In this code, the analysis does not flag any overflow in the code. However, the range of j wraps around to even values in the range $[-2^{31}..2]$ or $[2..2^{31}-2]$ and the value of i wraps around to -2^{31} .

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 230
    i = i * 2;
    // i = -231
    printf("%d", i);
}

void func2()
{
    int j = getVal();
    if (j > 0) {
        // Range of j: [1..231-1]
        j = j * 2;
        // Range of j: even values in [-231..2] or [2..231-2]
        printf("%d", j);
    }
}
```

warn-with-wrap-around

Polyspace flags signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In the following code, *j* has values in the range $[1..2^{31}-1]$ before the orange overflow. Polyspace considers that *j* has even values in the range $[-2^{31}..2]$ or $[2..2^{31}-2]$ after the overflow.

Similarly, *i* has value 2^{30} before the red overflow and value -2^{31} after it .


```

#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 230
    // Result of * operation overflows
    i = i * 2;
    // i = -231
    printf("%d", i);
}

void func2()
{
    int j = getVal();
    if (j > 0) {
        // Range of j: [1..231-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [-231..2] or [2..231-2]
        printf("%d", j);
    }
}

```

Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to forbid or warn-with-wrap-around. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.
- In Polyspace Code Prover, overflowing signed constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with signed constants, use the Polyspace Bug Finder checker Integer constant overflow.

Command-Line Information

Parameter: -signed-integer-overflows

Value: forbid|allow|warn-with-wrap-around

Default: forbid

Example (Code Prover): polyspace-code-prover -sources *file_name* -signed-integer-overflows allow

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -signed-integer-overflows allow

See Also

Overflow|Overflow mode for unsigned integer (-unsigned-integer-overflows)

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2018b

Overflow mode for unsigned integer (-unsigned-integer-overflows)

Specify whether result of overflow is wrapped around or truncated

Description

This option affects a Code Prover analysis only.

Specify whether Polyspace flags unsigned integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node under **Code Prover Verification**.

Command line: Use the option `-unsigned-integer-overflows`. See “Command-Line Information” (Polyspace Code Prover).

Why Use This Option

Use this option to specify whether to check for unsigned integer overflows and to specify the assumptions the analysis makes following an overflow.

Settings

Default: allow

forbid

Polyspace flags unsigned integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:
 - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
 - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

In the following code, `j` has values in the range $[1 \dots 2^{32} - 1]$ before the orange overflow. Polyspace considers that `j` has even values in the range $[2 \dots 4294967294]$ after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // Result of * operation overflows
    i = i * 2;
    // Remaining code in current scope not analyzed
    printf("%u", i);
}

void func2()
{
    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..232-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 4294967294]
        printf("%u", j);
    }
}
```

allow

Polyspace does not flag unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `MAX_INT + 1` wraps to `MIN_INT`. This behavior conforms to the ANSI C (ISO C++) standard.

In this code, the analysis does not flag any overflow in the code. However, the range of `j` wraps around to even values in the range `[0..232-2]` and the value of `i` wraps around to `0`.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 231
    i = i * 2;
    // i = 0
    printf("%u", i);
}

void func2()
{
    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..232-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}
```

warn-with-wrap-around

Polyspace flags unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `MAX_INT + 1` wraps to `MIN_INT`.

In the following code, `j` has values in the range `[1..232-1]` before the orange overflow. Polyspace considers that `j` has even values in the range `[0 .. 4294967294]` after the overflow.

Similarly, `i` has value `231` before the red overflow and value `0` after it.

```

#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 231
    i = i * 2;
    // i = 0
    printf("%u", i);
}
void func2()
{
    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..232-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}

```

Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to forbid or warn-with-wrap-around. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.
- In Polyspace Code Prover, overflowing unsigned constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with unsigned constants, use the Polyspace Bug Finder checker `Unsigned integer constant overflow`.

Command-Line Information

Parameter: `-unsigned-integer-overflows`

Value: `forbid|allow|warn-with-wrap-around`

Default: `allow`

Example (Code Prover): `polyspace-code-prover -sources file_name -unsigned-integer-overflows allow`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -unsigned-integer-overflows allow`

See Also

`Overflow|Overflow mode for signed integer (-signed-integer-overflows)`

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2018b

Disable checks for non-initialization (-disable-initialization-checks)

Disable checks for non-initialized variables and pointers

Description

This option affects a Code Prover analysis only.

Specify that Polyspace Code Prover must not check for non-initialization in your code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-disable-initialization-checks`. See “Command-Line Information” on page 2-228.

Why Use This Option

Use this option if you do not want to detect instances of non-initialized variables.

Settings

On

Polyspace Code Prover does not perform the following checks:

- `Non-initialized local variable`: Local variable is not initialized before being read.
- `Non-initialized variable`: Variable other than local variable is not initialized before being read.
- `Non-initialized pointer`: Pointer is not initialized before being read.
- `Return value not initialized`: C function does not return value when expected.

Polyspace assumes that, at declaration:

- Variables have full-range of values allowed by their type.
- Pointers can be NULL-valued or point to a memory block at an unknown offset.

Off (default)

Polyspace Code Prover checks for non-initialization in your code. The software displays red checks if, for instance, a variable is not initialized and orange checks if a variable is initialized only on some execution paths.

Tips

- If you select this option, the software does not report most violations of MISRA C:2004 rule 9.1, and MISRA C:2012 Rule 9.1.

- If you select this option, the number and type of orange checks in your code can change.

For instance, the following table shows an additional orange check with the option enabled.

Checks for Non-initialization Enabled	Checks for Non-initialization Disabled
<pre>void func(int flag) { int var1,var2; if(flag==0) { var1=var2; } else { var1=0; } var2=var1 + 1; }</pre>	<pre>void func(int flag) { int var1,var2; if(flag==0) { var1=var2; } else { var1=0; } var2=var1 + 1; }</pre>
<p>In this example, the software produces:</p> <ul style="list-style-type: none"> • A red Non-initialized local variable check on <code>var2</code> in the <code>if</code> branch. The verification continues as if only the <code>else</code> branch of the <code>if</code> statement exists. • A green Non-initialized local variable check on <code>var1</code> in the last statement. <code>var1</code> has the assigned value 0. • A green Overflow check on the <code>+</code> operation. 	<p>In this example, the software:</p> <ul style="list-style-type: none"> • Does not produce Non-initialized local variable checks. At initialization, the software assumes that <code>var2</code> has full range of <code>int</code> values. Following the <code>if</code> statement, because the software considers both <code>if</code> branches, it assumes that <code>var1</code> also has full range of <code>int</code> values. • Produces an orange Overflow check on the <code>+</code> operation. For instance, if <code>var1</code> has the maximum <code>int</code> value, adding 1 to it can cause an overflow.

Command-Line Information

Parameter: `-disable-initialization-checks`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -disable-initialization-checks`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -disable-initialization-checks`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Check that global variables are initialized after warm reboot (-check-globals-init)

Check that global variables are assigned values in designed initialization code

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must check whether all non-const global variables (and local static variables) are explicitly initialized at declaration or within a section of code marked as initialization code.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the main function (only once). The initialization code starts from the beginning of main and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option -check-globals-init. See “Command-Line Information” on page 2-232.

Why Use This Option

In a warm reboot, to save time, the bss segment of a program, which might hold variable values from a previous state, is not loaded. Instead, the program is supposed to explicitly initialize all non-const variables without default values before execution. You can use this option to delimit the initialization code and verify that all non-const global variables are indeed initialized in a warm reboot.

For instance, in this simple example, the global variable aVar is initialized in the initialization code section but the variable anotherVar is not.

```
int aVar;
const int aConst = -1;
int anotherVar;

int main() {
    aVar = aConst;
#pragma polyspace_end_of_init
    return 0;
}
```

Settings

On

Polyspace checks whether all global variables are initialized in the designated initialization code. The initialization code starts from the beginning from `main` and continues up to the pragma `polyspace_end_of_init`.

The results are reported using the check `Global variable not assigned a value in initialization code`.

Off (default)

Polyspace does not check for initialization of global variables in a designated code section.

However, the verification continues to check if a variable is initialized at the time of use. The results are reported using the check `Non-initialized variable`.

Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to `C`.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef-s`).

You cannot use this option with the following options:

- `Disable checks for non-initialization (-disable-initialization-checks)`
- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

Tips

- You can use this option along with the option `Verify initialization section of code only (-init-only-mode)` to check the initialization code before checking the remaining program.

This approach has the following benefits compared to checking the entire code in one run:

- Run-time errors in the initialization code can invalidate analysis of the remaining code. You can run a comparatively quicker check on the initialization code before checking the remaining program.
- You can review results of the checker `Global variable not assigned a value in initialization code` relatively easily.

Consider this example. There is an orange check on `var` because `var` might remain uninitialized when the `if` and `else if` statements are skipped.

```
int var;
```

```

int checkSomething(void);
int checkSomethingElse(void);

int main() {
    int local_var;
    if(checkSomething())
    {
        var=0;
    }
    else if(checkSomethingElse()) {
        var=1;
    }
    #pragma polyspace_end_of_init
    var=2;
    local_var = var;
    return 0;
}

```

To review this check and understand when `x` might be non-initialized, you have to browse through all instances of `x` on the **Variable Access** pane. If you check the initialization code alone, only the code in bold gets checked and you have to browse through only the instances in the initialization code.

- The check is only as good as your placement of the `pragma polyspace_end_of_init`. For instance:
 - Place the pragma only after initialization code ends.

Otherwise, a variable might appear falsely uninitialized.

- Try to place the pragma directly in the `main` function, that is, outside a block. If you place the pragma in a block, the check considers only those paths that end in the block.

All paths that end in the block might have a variable initialized but paths that skip the block might let the variable go uninitialized. If you do place the pragma in a block, make sure that it is okay if a variable stays uninitialized outside the block.

For instance, in this example, the variable `var` is initialized on all paths that end at the location of the pragma. The check is green despite the fact that the `if` block might be skipped, letting the variable go uninitialized.

```

int var;

int func();

int main() {
    int err = func();
    if(err) {
        var = 0;
    }
    #pragma polyspace_end_of_init
    int a = var;
    return 0;
}

```

The issue is detected by the checker if you place the pragma after the `if` block ends.

- Do not place the pragma in a loop.

If you place the pragma in a loop, you can see results that are difficult to interpret. For instance, in this example, both `aVar` and `anotherVar` are initialized in one iteration of the loop. However, the pragma only considers the first iteration of the loop when it shows a green check for initialization. If a variable is initialized on a later iteration, the check is orange.

```
int aVar;
int anotherVar;

void main() {
    for(int i=0; i<=1; i++) {
        if(i == 0)
            aVar = 0;
        else
            anotherVar = 0;
        #pragma polyspace_end_of_init
    }
}
```

The check is red if you verify initialization code alone and do not initialize a variable in the first loop iteration. To avoid these incorrect red or orange checks, do not place the pragma in a loop.

- To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

If you check initialization code only using the option `Verify initialization section of code only (-init-only-mode)`, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

Command-Line Information

Parameter: `-check-globals-init`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -check-globals-init`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -check-globals-init`

See Also

Global variable not assigned a value in initialization code | `Verify initialization section of code only (-init-only-mode)`

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2020a

Detect stack pointer dereference outside scope (-detect-pointer-escape)

Find cases where a function returns a pointer to one of its local variables

Description

This option affects a Code Prover analysis only.

Specify that the verification must detect cases where you access a variable outside its scope via pointers. Such an access can happen, for example, when a function returns a pointer to a local variable and you dereference the pointer outside the function. The dereference causes undefined behavior because the local variable that the pointer points to does not live outside the function.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-detect-pointer-escape`. See “Command-Line Information” on page 2-234.

Why Use This Option

Use this option to enable detection of pointer escape.

Settings

On

The **Illegally dereferenced pointer** check performs an additional task, besides its usual specifications. When you dereference a pointer, the check also determines if you are accessing a variable outside its scope through the pointer. The check is:

- Red, if all the variables that the pointer points to are accessed outside their scope.

For instance, you dereference a pointer `ptr` in a function `func` that is called twice in your code. In both calls, when you perform the dereference `*ptr`, `ptr` is pointing to variables outside their scope. Therefore, the **Illegally dereferenced pointer** check is red.
- Orange, if only some of the variables that the pointer points to are accessed outside their scope.
- Green, if none of the variables that the pointer points to are accessed outside their scope, and other requirements of the check are also satisfied.

In the following code, if you enable this option, Polyspace Code Prover produces a red **Illegally dereferenced pointer** check on `*ptr`. Otherwise, the **Illegally dereferenced pointer** check on `*ptr` is green.

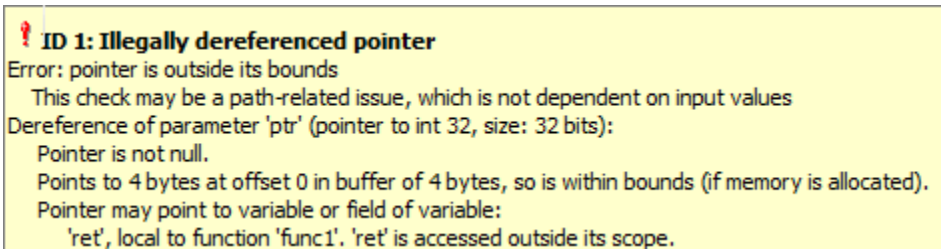
```
void func2(int *ptr) {  
    *ptr = 0;  
}
```

```

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}

```

The **Result Details** pane displays a message indicating that `ret` is accessed outside its scope.



Off (default)

When you dereference a pointer, the **Illegally dereferenced pointer** check does not check for whether you are accessing a variable outside its scope. The check is green even if the pointer dereference is outside the variable scope, as long as it satisfies these requirements:

- The pointer is not NULL.
- The pointer points within the memory buffer.

Tips

The detection of stack pointer dereference outside scope does not apply to certain types of pointers. For specific limitations, see “Limitations of Polyspace Verification” (Polyspace Code Prover).

Command-Line Information

Parameter: `-detect-pointer-escape`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -detect-pointer-escape`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -detect-pointer-escape`

See Also

Illegally dereferenced pointer

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2015a

Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)

Allow arithmetic on pointer to a structure field so that it points to another field

Description

This option affects a Code Prover analysis only.

Specify that a pointer assigned to a structure field can point outside its bounds as long as it points within the structure.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 2-236 for other options you must also enable.

Command line: Use the option `-allow-ptr-arith-on-struct`. See “Command-Line Information” on page 2-236.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. Once you assign a pointer to a structure field, you can perform pointer arithmetic and use the result to access another structure field.

Settings

On

A pointer assigned to a structure field can point outside the bounds imposed by the field as long as it points within the structure. For instance, in the following code, unless you use this option, the verification will produce a red **Illegally dereferenced pointer** check:

```
void main(void) {
  struct S {char a; char b; int c;} x;
  char *ptr = &x.b;
  ptr ++;
  *ptr = 1; // Red on the dereference, because ptr points outside x.b
}
```

Off (default)

A pointer assigned to a structure field can point only within the bounds imposed by the field.

Tips

- The verification does not allow a pointer with negative offset values. This behavior occurs irrespective of whether you choose the option **Enable pointer arithmetic across fields**.
- Using this option can slightly increase the number of orange checks. The option relaxes the constraint that a pointer to a structure field cannot point to other fields of the structure. In

exchange for relaxing this constraint, the verification loses precision on the boundary of fields within a structure and treats the structure as a whole. Pointer dereferences that were previously green can now turn orange.

Use this option if you follow a policy of reviewing red checks only and you need to work around red checks from pointer arithmetic within a structure.

- Before using this option, consider the costs of using pointer arithmetic across different fields of a structure.

Unlike an array, members of a structure can have different data types. For efficient storage, structures use padding to accommodate this difference. When you increment a pointer pointing to a structure member, you might not point to the next member. When you dereference this pointer, you cannot rely on what you are reading or writing to.

Dependency

This option is available only if you set `Source code language (-lang)` to C.

Command-Line Information

Parameter: `-allow-ptr-arith-on-struct`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -allow-ptr-arith-on-struct`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -allow-ptr-arith-on-struct`

See Also

Allow incomplete or partial allocation of structures (`-size-in-bytes`) |
Illegally dereferenced pointer

Topics

“Prepare Scripts for Polyspace Analysis”

Allow incomplete or partial allocation of structures (-size-in-bytes)

Allow a pointer with insufficient memory buffer to point to a structure

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow dereferencing a pointer that points to a structure but has a sufficient buffer for only some of the structure's fields.

This type of pointer results when a pointer to a smaller structure is cast to a pointer to a larger structure. The pointer resulting from the cast has sufficient buffer for only some fields of the larger structure.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-size-in-bytes`. See "Command-Line Information" on page 2-238.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. You can point to a structure even when the buffer allowed for the pointer is not sufficient for all the structure fields.

Settings

On

When a pointer with insufficient buffer is dereferenced, Polyspace does not produce an **Illegally dereferenced pointer** error, as long as the dereference occurs within allowed buffer.

For instance, in the following code, the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`. Therefore, with the option on, Polyspace considers that the first two dereferences are valid. The third dereference takes `p` outside its allowed buffer. Therefore, Polyspace produces an **Illegally dereferenced pointer** error on the third dereference.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
    BIG *p = malloc(sizeof(LITTLE));

    if (p!= ((void *) 0) ) {
        p->a = 0 ;
        p->b = 0 ;
        p->c = 0 ;    // Red IDP check
    }
}
```

```

    }
}

```

Off (default)

Polyspace does not allow dereferencing a pointer to a structure if the pointer does not have sufficient buffer for all fields of the structure. It produces an **Illegally dereferenced pointer** error the first time you dereference the pointer.

For instance, in the following code, even though the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`, Polyspace considers that dereferencing `p` is invalid.

```

#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
    BIG *p = malloc(sizeof(LITTLE));

    if (p!= ((void *) 0) ) {
        p->a = 0 ;    // Red IDP check
        p->b = 0 ;
        p->c = 0 ;
    }
}

```

Tips

- If you do not turn on this option, you cannot point to the field of a partially allocated structure.

For instance, in the preceding example, if you do not turn on the option and perform the assignment

```
int *ptr = &(p->a);
```

Polyspace considers that the assignment is invalid. If you dereference `ptr`, it produces an **Illegally dereferenced pointer** error.

- Using this option can slightly increase the number of orange checks.

Command-Line Information

Parameter: `-size-in-bytes`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -size-in-bytes`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -size-in-bytes`

See Also

Enable pointer arithmetic across fields (`-allow-ptr-arith-on-struct`) |
Illegally dereferenced pointer

Topics

“Prepare Scripts for Polyspace Analysis”

Permissive function pointer calls (-permissive-function-pointer)

Allow type mismatch between function pointers and the functions they point to

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow function pointer calls where the type of the function pointer does not match the type of the function.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 2-241 for other options you must also enable.

Command line: Use the option `-permissive-function-pointer`. See “Command-Line Information” on page 2-241.

Why Use This Option

By default, Code Prover does not recognize calls through function pointers when a type mismatch occurs. Fix the type mismatch whenever possible.

Use this option if:

- You cannot fix the type mismatch, and
- The analysis does not cover a significant portion of your code because calls via function pointers are not recognized.

Settings

On

The verification must allow function pointer calls where the type of the function pointer does not match the type of the function. For instance, a function declared as `int f(int*)` can be called by a function pointer declared as `int (*fptr)(void*)`.

Only type mismatches between pointer types are allowed. Type mismatches between nonpointer types cause compilation errors. For instance, a function declared as `int f(int)` cannot be called by a function pointer declared as `int (*fptr)(double)`.

Off (default)

The verification must require that the argument and return types of a function pointer and the function it calls are identical.

Type mismatches are detected with the check `Correctness` condition.

Tips

- With sources that use function pointers extensively, enabling this option can cause loss in performance. This loss occurs because the verification has to consider more execution paths.
- Using this option can increase the number of orange checks. Some of these orange checks can reveal a real issue with the code.

Consider these examples where a type mismatch occurs between the function pointer type and the function that it points to:

- In this example, the function pointer `obj_fptr` has an argument that is a pointer to a three-element array. However, it points to a function whose corresponding argument is a pointer to a four-element array. In the body of `foo`, four array elements are read and incremented. The fourth element does not exist and the `++` operation reads a meaningless value.

```
typedef int array_three_elements[3];
typedef void (*fptr)(array_three_elements*);

typedef int array_four_elements[4];
void foo(array_four_elements*);

void main() {
    array_three_elements arr[3] = {0,0,0};
    array_three_elements *ptr;
    fptr obj_fptr;

    ptr = &arr;
    obj_fptr = &foo;

    //Call via function pointer
    obj_fptr(&ptr);
}

void foo(array_four_elements* x) {
    int i = 0;
    int *current_pos;

    for(i = 0; i < 4; i++) {
        current_pos = (*x) + i;
        (*current_pos)++;
    }
}
```

Without this option, an orange `Correctness condition` check appears on the call `obj_fptr(&ptr)` and the function `foo` is not verified. If you use this option, the body of `foo` contains several orange checks. Review the checks carefully and make sure that the type mismatch does not cause issues.

- In this example, the function pointer has an argument that is a pointer to a structure with three `float` members. However, the corresponding function argument is a pointer to an unrelated structure with one array member. In the function body, the `strlen` function is used assuming the array member. Instead the `strlen` call reads the `float` members and can read meaningless values, for instance, values stored in the structure padding.

```

#include <string.h>
struct point {
    float x;
    float y;
    float z;
};
struct message {
    char msg[10] ;
};
void foo(struct message*);

void main() {
    struct point pt = {3.14, 2048.0, -1.0} ;
    void (*obj_fptr)(struct point *) ;

    obj_fptr = &foo;

    //Call via function pointer
    obj_fptr(&pt);
}

void foo(struct message* x) {
    int y = strlen(x->msg) ;
}

```

Without this option, an orange `Correctness` condition check appears on the call `obj_fptr(&pt)` and the function `foo` is not verified. If you use this option, the function contains an orange check on the `strlen` call. Review the check carefully and make sure that the type mismatch does not cause issues.

Dependency

This option is available only if you set `Source code language (-lang)` to `C`.

Command-Line Information

Parameter: `-permissive-function-pointer`

Default: `Off`

Example (Code Prover): `polyspace-code-prover -sources file_name -lang c -permissive-function-pointer`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -lang c -permissive-function-pointer`

See Also

`Correctness condition`

Topics

“Prepare Scripts for Polyspace Analysis”

Consider non finite floats (-allow-non-finite-floats)

Enable an analysis mode that incorporates infinities and NaNs

Description

Enable an analysis mode that incorporates infinities and NaNs for floating point operations.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-allow-non-finite-floats`. See “Command-Line Information” on page 2-244.

Why Use This Option

Code Prover

By default, the analysis does not incorporate infinities and NaNs. For instance, the analysis terminates the execution thread where a division by zero occurs and does not consider that the result could be infinite.

If you use functions such as `isinf` or `isnan` and account for infinities and NaNs in your code, set this option. When you set this option and a division by zero occurs for instance, the execution thread continues with infinity as the result of the division.

Set this option alone if you are sure that you have accounted for infinities and NaNs in your code. Using the option alone effectively disables many numerical checks on floating point operations. If you have generally accounted for infinities and NaNs, but you are not sure that you have considered all situations, set these additional options:

- **Infinities** (`-check-infinite`): Use `warn-first`.
- **NaNs** (`-check-nan`): Use `warn-first`.

Bug Finder

If the analysis flags comparisons using `isinf` or `isnan` as dead code, use this option. By default, a Bug Finder analysis does not incorporate infinities and NaNs.

Settings

On

The analysis allows infinities and NaNs. For instance, in this mode:

- The analysis assumes that floating-point operations can produce results such as infinities and NaNs.

By using options `Infinities` (`-check-infinite`) and `NaNs` (`-check-nan`), you can choose to highlight operations that produce nonfinite results and stop the execution threads where the nonfinite results occur. These options are not available for a Bug Finder analysis.

- The analysis assumes that floating-point variables with unknown values can have any value allowed by their type, including infinite or NaN. Floating-point variables with unknown values include volatile variables and return values of stubbed functions.

Off (default)

The analysis does not allow infinities and NaNs. For instance, in this mode:

- The Code Prover analysis produces a red check on a floating-point operation that produces an infinity or a NaN as the only possible result on all execution paths. The verification produces an orange check on a floating-point operation that can potentially produce an infinity or NaN.
- The Code Prover analysis assumes that floating-point variables with unknown values are full-range but finite.
- The Bug Finder analysis shows comparisons with infinity using `isinf` as dead code.

Tips

- The IEEE 754 Standard allows special quantities such as infinities and NaN so that you can handle certain numerical exceptions without aborting the code. Some implementations of the C standard support infinities and NaN.
- If your compiler supports infinities and NaNs and you account for them explicitly in your code, use this option so that the verification also allows them.

For instance, if a division results in infinity, in your code, you specify an alternative action. Therefore, you do not want the verification to highlight division operations that result in infinity.

- If your compiler supports infinities and NaNs but you are not sure if you account for them explicitly in your code, use this option so that the verification incorporates infinities and NaNs. Use the options `-check-nan` and `-check-infinite` with argument `warn` so that the verification highlights operations that result in infinities and NaNs, but does not stop the execution thread. These options are not available for a Bug Finder analysis.
- If you run a Code Prover analysis and use this option, checkers for overflow, division by zero and other numerical run-time errors are disabled. See “Numerical Checks” (Polyspace Code Prover Access).

If you run a Bug Finder analysis and use this option:

- The checkers for overflow and division by zero are disabled. See “Numerical Defects” (Polyspace Bug Finder Access).
- The checker `Floating point comparison with equality operators` can show false positives.
- If you select this option, the number and type of Code Prover checks in your code can change.

For instance, in the following example, when you select the option, the results have one less red check and three more green checks.

Infinities and NaNs Not Allowed	Infinities and NaNs Allowed
<p>Code Prover produces a Division by zero error and stops verification.</p> <pre data-bbox="324 388 625 556">double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z; }</pre>	<p>If you select this option, Code Prover does not check for a Division by zero error.</p> <pre data-bbox="901 388 1201 556">double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z; }</pre> <p>The analysis assumes that dividing by zero results in:</p> <ul data-bbox="901 672 1226 787" style="list-style-type: none"> • Value of x equal to Inf • Value of y equal to 0.0 • Value of z equal to NaN <p>In your analysis results in the Polyspace user interface, if you place your cursor on y and z, you can see the nonfinite values Inf and NaN respectively in the tooltip.</p>

- You cannot run the Automatic Orange Tester in Code Prover if you incorporate non-finites in your analysis.

Command-Line Information

Parameter: -allow-non-finite-floats

Default: Off

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -allow-non-finite-floats

Example (Code Prover): polyspace-code-prover -sources *file_name* -allow-non-finite-floats

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -allow-non-finite-floats

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -allow-non-finite-floats

See Also

“Numerical Defects” (Polyspace Bug Finder Access) | “Numerical Checks” (Polyspace Code Prover Access) | Infinities (-check-infinite) | NaNs (-check-nan)

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016a

Infinities (-check-infinite)

Specify how to handle floating-point operations that result in infinity

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in infinities.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 2-246 for other options you must also enable.

Command line: Use the option `-check-infinite`. See “Command-Line Information” on page 2-246.

Why Use This Option

Use this option to enable detection of floating-point operations that result in infinities.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: allow

allow

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Overflow** check.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

warn-first

The verification produces a check on the operation. The check determines if the result of the operation is infinite when the operands themselves are not infinite. The verification does not terminate the execution thread that produces infinity.

If the verification detects an operation that produces infinity as the only possible result on all execution paths and the operands themselves are never infinite, the check is red. If the operation can potentially result in infinity, the check is orange.

For instance, in the following code, there is a nonblocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
```

```
    return x;
}
```

Even though the **Overflow** check on the / operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on x in the return statement.

forbid

The verification produces a check on the operation and terminates the execution thread that produces infinity.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced infinity.

For instance, in the following code, there is a blocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

The verification stops because the **Overflow** check on the / operation is red. For instance, a **Non-initialized local variable** check does not appear on x in the return statement.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`.

Command-Line Information

Parameter: -check-infinite

Value: allow|warn-first|forbid

Default: allow

Example (Code Prover): `polyspace-code-prover -sources file_name -check-infinite forbid`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -check-infinite forbid`

See Also

Polyspace Analysis Options

`Consider non finite floats (-allow-non-finite-floats)` | `NaNs (-check-nan)`

Polyspace Results

Overflow

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016a

NaNs (-check-nan)

Specify how to handle floating-point operations that result in NaN

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in NaN.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 2-248 for other options you must also enable.

Command line: Use the option `-check-nan`. See “Command-Line Information” on page 2-248.

Why Use This Option

Use this option to enable detection of floating-point operations that result in NaN-s.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: allow

allow

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Invalid operation on floats** check.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

warn-first

The verification produces a check on the operation. The check determines if the result of the operation is NaN when the operands themselves are not NaN. For instance, the check flags the operation `val1 + val2` only if the result can be NaN when *both* `val1` and `val2` are not NaN. The verification does not terminate the execution thread that produces NaN.

If the verification detects an operation that produces NaN as the only possible result on all execution paths and the operands themselves are never NaN, the check is red. If the operation can potentially result in NaN, the check is orange.

For instance, in the following code, there is a nonblocking **Invalid operation on floats** check for NaN.

```
double func(void) {  
    double x=1.0/0.0;  
    double y=x-x;  
    return y;  
}
```

Even though the **Invalid operation on floats** check on the `-` operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on `y` in the `return` statement.

forbid

The verification produces a check on the operation and terminates the execution thread that produces NaN.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced a NaN.

For instance, in the following code, there is a blocking **Invalid operation on floats** check for NaN.

```
double func(void) {  
    double x=1.0/0.0;  
    double y=x-x;  
    return y;  
}
```

The verification stops because the **Invalid operation on floats** check on the `-` operation is red. For instance, a **Non-initialized local variable** check does not appear on `y` in the `return` statement.

The **Invalid operation on floats** check for NaN also appears on the `/` operation and is green.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`.

Command-Line Information

Parameter: `-check-nan`

Value: `allow|warn-first|forbid`

Default: `allow`

Example (Code Prover): `polyspace-code-prover -sources file_name -check-nan forbid`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -check-nan forbid`

See Also

Polyspace Analysis Options

`Consider non finite floats (-allow-non-finite-floats)|Infinities (-check-infinite)`

Polyspace Results

Invalid operation on floats

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016a

Subnormal detection mode (-check-subnormal)

Detect operations that result in subnormal floating-point values

Description

This option affects a Code Prover analysis only.

Specify that the verification must check floating-point operations for subnormal results.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-check-subnormal`. See “Command-Line Information” on page 2-252.

Why Use This Option

Use this option to detect floating-point operations that result in subnormal values.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

Settings

Default: allow

allow

The verification does not check operations for subnormal results.

forbid

The verification checks for subnormal results.

The verification stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.

warn-all

The verification checks for subnormal results and highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.

The verification continues even if the check is red.

warn-first

The verification checks for subnormal results but only highlights first occurrences of subnormal values. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.

The verification continues even if the check is red.

For details of the result colors in each mode, see `Subnormal float`.

Tips

- If you want to see only those operations where a subnormal value originates from non-subnormal operands, use the `warn-first` mode.

For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results from certain operations. If you use the `warn-first` mode, the first operation causing the subnormal result is highlighted.

warn-all	warn-first
<pre>void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p>In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.</p>	<pre>void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p>In this example, <code>difference1</code> and <code>difference2</code> can be subnormal if <code>arg1</code> and <code>arg2</code> are sufficiently close. The first two checks for subnormal results are orange. <code>val1</code> and <code>val2</code> cannot be subnormal unless <code>difference1</code> and <code>difference2</code> are subnormal. The last two checks for subnormal results are green.</p> <p>Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.</p>

- If you want to see where a subnormal value originates and do not want to see subnormal results arising from the same cause more than once, use the `forbid` mode.

For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results for `arg1 - arg2`. If you use the `forbid` mode and perform the operation `arg1 - arg2` twice in succession, only the first operation is highlighted. The second operation is not highlighted because the subnormal result for the second operation arises from the same cause as the first operation.

warn-all	forbid
<pre>void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p>In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.</p>	<pre>void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p>In this example, <code>difference1</code> can be subnormal if <code>arg1</code> and <code>arg2</code> are sufficiently close. The first check for subnormal results is orange. Following this check, the verification excludes from consideration:</p> <ul style="list-style-type: none"> The close values of <code>arg1</code> and <code>arg2</code> that led to the subnormal value of <code>difference1</code>. <p>In the subsequent operation <code>arg1 - arg2</code>, the check is green and <code>difference2</code> is not subnormal. The result of the check on <code>difference2 * 2</code> is green for the same reason.</p> <ul style="list-style-type: none"> The subnormal value of <code>difference1</code>. <p>In the subsequent operation <code>difference1 * 2</code>, the check is green.</p>

- You cannot run the Automatic Orange Tester if you check for subnormals in your verification.

Command-Line Information

Parameter: `-check-subnormal`

Value: `allow` | `warn-first` | `warn-all` | `forbid`

Default: `allow`

Example (Code Prover): `polyspace-code-prover -sources file_name -check-subnormal forbid`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -check-subnormal forbid`

See Also

Polyspace Results

Subnormal float

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016b

Detect uncalled functions (-uncalled-function-checks)

Detect functions that are not called directly or indirectly from `main` or another entry point function

Description

This option affects a Code Prover analysis only.

Detect functions that are not called directly or indirectly from `main` or another entry point function during run-time.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-uncalled-function-checks`. See “Command-Line Information” on page 2-253.

Why Use This Option

Typically, after verification, the **Dashboard** pane shows functions that are not called during verification. However, you do not see them in your analysis results or reports. You cannot comment on them or justify them.

If you want to see these uncalled functions in your analysis results and reports, use this option.

Settings

Default: none

none

The verification does not generate checks for uncalled functions.

never-called

The verification generates checks for functions that are defined but not called.

called-from-unreachable

The verification generates checks for functions that are defined and called from an unreachable part of the code.

all

The verification generates checks for functions that are:

- Defined but not called
- Defined and called from an unreachable part of the code.

Command-Line Information

Parameter: `-uncalled-function-checks`

Value: none | never-called | called-from-unreachable | all

Default: none

Example (Code Prover): polyspace-code-prover -sources *file_name* -uncalled-function-checks all

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -uncalled-function-checks all

See Also

Function not reachable | Function not called

Topics

“Prepare Scripts for Polyspace Analysis”

Precision level (-0)

Specify a precision level for the verification

Description

This option affects a Code Prover analysis only.

Specify the precision level that the verification must use.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-0#`, for instance, `-00` or `-01`. See “Command-Line Information” on page 2-256.

Why Use This Option

Higher precision leads to greater number of proven results but also requires more verification time. Each precision level corresponds to a different algorithm used for verification.

In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: 2

0

This option corresponds to a static interval verification.

1

This option corresponds to a more complex static interval verification.

2

This option corresponds to a complex polyhedron model of domain values with additional precision for interprocedural analysis depending on the option `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

3

This option is only suitable for code having less than 1000 lines. Using this option, the percentage of proven results can be very high.

Tips

- For best results in reasonable time, use the default level 2. If the verification takes a long time, reduce precision. However, the number of unproven checks can increase. Likewise, to reduce orange checks, you can improve your precision. But the verification can take significantly longer time.

- The precision levels 2 and below begin to take effect only from verification levels higher than Software Safety Analysis level 0. See also Verification level (-to).

For instance, to reduce analysis time, you might have reduced the verification level to Software Safety Analysis level 0. Do not try to reduce the precision level below 2 to lower the analysis time further.

Note that algorithms used in precision level 3 can also apply to the verification level Software Safety Analysis level 0.

Command-Line Information

Parameter: -00 | -01 | -02 | -03

Default: -02

Example (Code Prover): `polyspace-code-prover -sources file_name -01`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -01`

See Also

Specific precision (-modules-precision) | Verification level (-to)

Topics

“Prepare Scripts for Polyspace Analysis”

Verification level (-to)

Specify number of times the verification process runs on your code

Description

This option affects a Code Prover analysis only.

Specify the number of times the Polyspace verification process runs on your source code. Each run can lead to greater number of proven results but also requires more verification time.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-to`. See “Command-Line Information” on page 2-259.

Why Use This Option

There are many reasons you might want to increase or decrease the verification level. For instance:

- Coding rules are checked early during the compilation phase, with some exception only. If you check for coding rules alone, you can lower the verification level. See “Check for Coding Standard Violations”.
- If you see many orange checks after verification, try increasing the verification level. However, increasing the verification level also increases verification time.

In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: Software Safety Analysis level 2

Source Compliance Checking

Polyspace checks for compilation errors only. Most coding rule violations are also found in this phase.

Software Safety Analysis level 0

The verification process performs some simple analysis. The analysis is designed to reach completion despite complexities in the code.

If the verification gets stuck at a higher level, try running to this level and review the results.

Software Safety Analysis level 1

The verification process analyzes each function once with algorithms whose complexity depends on the precision level. See **Precision level (-0)**. The analysis starts from the top of the function call hierarchy (an actual or generated `main` function) and propagates to the leaves of the call hierarchy.

Software Safety Analysis level 2

The verification process analyzes each function twice. In the first pass, the analysis propagates from the top of the function call hierarchy to the leaves. In the second pass, the analysis

propagates from the leaves back to the top. Each pass uses information gathered from the previous pass.

Use this option for most accurate results in reasonable time.

Software Safety Analysis level 3

The verification process runs three times on each function: from the top of the function call hierarchy to the leaves, from the leaves to the top, and from the top to the leaves again. Each pass uses information gathered from the previous pass.

Software Safety Analysis level 4

The verification process runs four passes on each function: from the top of the function call hierarchy to the leaves twice. Each pass uses information gathered from the previous pass.

other

If you use this option, Polyspace verification will make 20 passes unless you stop it manually.

Tips

- Use a higher verification level for fewer orange checks.

In some cases, if the verification can detect that results of maximum precision are available after an earlier level, the verification stops and does not proceed to the level that you specify.

Difference between Level 0 and 1

The following example illustrates the difference between **Software Safety Analysis level 0** and **Software Safety Analysis level 1**. In level 1, Code Prover can establish the success of the final assertion that involves a relation between two array elements even without knowing the actual elements of the array.

Software Safety Analysis Level 0	Software Safety Analysis Level 1
<pre>extern int tab[]; int main() { int i = tab[3]; int j = tab[1]; if (i > j) { int l = i-j; assert(l > 0); } }</pre>	<pre>extern int tab[]; int main() { int i = tab[3]; int j = tab[1]; if (i > j) { int l = i-j; assert(l > 0); } }</pre>

In the table, verification produces an orange **Division by Zero** check during level 0 verification. The check turns green during level 1. The verification acquires more precise knowledge of x in the higher level.

If a higher verification level fails because the verification runs out of memory, but results are available at a lower level, Polyspace displays the results from the lower level.

- For best results, use the option **Software Safety Analysis level 2**. If the verification takes too long, use a lower **Verification level**. Fix red errors and gray code before rerunning the verification with higher verification levels.

- Use the option `Other` sparingly since it can increase verification time by an unreasonable amount. Using `Software Safety Analysis level 2` provides optimal verification of your code in most cases.
- If the **Verification Level** is set to `Source Compliance Checking`, do not run verification on a remote server. The source compliance checking, or compilation, phase takes place on your local computer anyway. Therefore, if you are running verification only to the end of compilation, run verification on your local computer.
- If you want to see global variable sharing and usage only use `Show global variable sharing and usage only (-shared-variables-mode)` to run a less extensive analysis.

Command-Line Information

Parameter: `-to`

Value: `compile | pass0 | pass1 | pass2 | pass3 | pass4 | other`

Default: `pass2`

Example (Code Prover): `polyspace-code-prover -sources file_name -to pass2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -to pass2`

You can also use these additional values not available in the user interface:

- C projects: `c-to-il` (C to intermediate language conversion phase)
- C++ projects: `cpp-to-il` (C++ to intermediate language conversion phase), `cpp-normalize` (C++ normalization phase), `cpp-link` (C++ link phase)

Use these values only if you have specific reasons to do so. For instance, to generate a blank constraints (DRS) template for C++ projects, you have to run an analysis up to the `cpp-normalize` phase.

See Also

Precision level (-0) | Show global variable sharing and usage only (-shared-variables-mode)

Topics

“Prepare Scripts for Polyspace Analysis”

Verification time limit (-timeout)

Specify a time limit on your verification

Description

This option affects a Code Prover analysis only.

Specify a time limit for the verification in hours. If the verification does not complete within that limit, it stops.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-timeout`. See “Command-Line Information” on page 2-260.

Why Use This Option

Use this option to impose a time limit on the verification.

By default, if an internal step in the verification lasts for more than 24 hours, the verification stops. You can use this option to reduce the time limit even further. Note that you can have verification results despite the verification timing out. For instance, if a step in Software Safety Analysis level 1 times out, you still get the results from level 0. See `Verification level (-to)`.

The option is useful only in very specific cases. Suppose your code has certain constructs that might slow down the verification. To check this, you can impose a time limit on the verification so that the verification stops if it takes too long.

Typically, Technical Support asks you to use this option as needed.

Settings

Enter the time in hours. For fractions of an hour, specify decimal form.

Command-Line Information

Parameter: `-timeout`

Value: *time*

Example (Code Prover): `polyspace-code-prover -sources file_name -timeout 5.75`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -timeout 5.75`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Sensitivity context (-context-sensitivity)

Store call context information to identify function call that caused errors

Description

This option affects a Code Prover analysis only.

Specify the functions for which the verification must store call context information. If the function is called multiple times, using this option helps you to distinguish between the different calls.

Set Option



User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-context-sensitivity`. See “Command-Line Information” (Polyspace Code Prover).

Why Use This Option

Suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use this option.

For instance, if a function contains a red or orange check and a green check on the same operation for two different calls, the software combines the contexts and displays an orange check on the operation. If you use this option, the check turns dark orange and the result details show the color of the check for each call.

 **Division by Zero** 

Warning (probable error): scalar division by zero may occur

operator / on type int 32

left: full-range [-2³¹ .. 2³¹-1]

right: full-range [-2³¹ .. 2³¹-1]

result: full-range [-2³¹ .. 2³¹-1]

Calling context	File	Scope	Line
operator / on type int 32 left: 1 right: 0	file.c	main	9
operator / on type int 32 left: 1 right: 11 result: 0	file.c	main	8

Settings

Default: none

none


The software does not store call context information for functions.

auto

The software stores call context information for checks in:

- Functions that form the leaves of the call tree. These functions are called by other functions, but do not call functions themselves.
- Small functions. The software uses an internal threshold to determine whether a function is small.

custom

The software stores call context information for functions that you specify. To enter the name of a function, click .

Tips

- If you select this option, you do not see tooltips in the body of the functions that benefit from this option (and keep the call contexts separate).
- If you select this option, the analysis can show some code operations in grey (unreachable code) even when you can identify execution paths leading to the operations. In this case, the grey code indicates operations that might be unreachable only in a particular call context.

For instance, suppose this function is called with the arguments -1 and 1 :

```
int isPositive (int num) {
    if(num < 0)
        return 0;
    return 1;
}
```

If you use the option with this function as argument, there are two unreachable code checks:

- The check on `if` is grey because when the function is called with argument -1, the `if` condition is always true.
- The check on the code inside the `if` branch is grey because when the function is called with argument 1, the `if` condition is always false.

Each unreachable code check indicates code that is unreachable only in a particular call context. You see the call context in the result details.

Command-Line Information

Parameter: `-context-sensitivity`

Value: `function1[,function2,...]`

Default: none

Example (Code Prover): `polyspace-code-prover -sources file_name -context-sensitivity myFunc1,myFunc2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -context-sensitivity myFunc1,myFunc2`

To allow the software to determine which functions receive call context storage, use the option `-context-sensitivity-auto`.

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Improve precision of interprocedural analysis (-path-sensitivity-delta)

Avoid certain verification approximations for code with fewer lines

Description

This option affects a Code Prover analysis only.

For smaller code, use this option to improve the precision of cross-functional analysis.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-path-sensitivity-delta`. See “Command-Line Information” on page 2-264.

Why Use This Option

Use this option to avoid certain software approximations on execution paths. Avoiding these approximations results in fewer orange checks but a much longer verification time.

For instance, for deep function call hierarchies or nested conditional statements, to complete verification in a reasonable amount of time, the software combines many execution paths and stores less information at each stage of verification. If you use this option, the software stores more information about the execution paths, resulting in a more precise verification.

Settings

Default: Off

Enter a positive integer to turn on this option.

Entering a higher value leads to a greater number of proven results, but also increases verification time exponentially. For instance, a value of 10 can result in very long verification times.

Tips

Use this option only when you have less than 1000 lines of code.

Command-Line Information

Parameter: `-path-sensitivity-delta`

Value: Positive integer

Example (Code Prover): `polyspace-code-prover -sources file_name -path-sensitivity-delta 1`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -path-sensitivity-delta 1`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Specific precision (`-modules-precision`)

Specify source files you want to verify at higher precision than the remaining verification

Description

This option affects a Code Prover analysis only.

Specify source files that you want to verify at a precision level higher than that for the entire verification.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node. See “Dependency” on page 2-266 for other options you must also enable.

Command line: Use the option `-modules-precision`. See “Command-Line Information” on page 2-266.


Why Use This Option

If a specific file is verified imprecisely leading to many orange checks in the file and elsewhere, you can improve the precision for that file.

Note that increasing precision also increases verification time.

Settings

Default: All files are verified with the precision you specified using **Precision > Precision level**.

Click  to enter the name of a file without the extension `.c` and the corresponding precision level.

Dependency

This option is available only if you set `Source code language (-lang)` to C or C-CPP.

Command-Line Information

Parameter: `-modules-precision`

Value: `file:00 | file:01 | file:02 | file:03`

Example (Code Prover): `polyspace-code-prover -sources file_name -01 -modules-precision My_File:02`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -01 -modules-precision My_File:02`

See Also

Precision level (-0)

Topics

“Prepare Scripts for Polyspace Analysis”

Inline (-inline)

Specify functions that must be cloned internally for each function call

Description

This option affects a Code Prover analysis only.

Specify the functions that the verification must clone internally for every function call.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Scaling** node.

Command line: Use the option `-inline`. See “Command-Line Information” on page 2-269.

Why Use This Option



Use this option sparingly. Sometimes, using the option helps to work around scaling issues during verification. If your verification takes too long, Technical Support can ask you to use this option for certain functions.

Do not use this option to understand results. For instance, suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use the option `Sensitivity context (-context-sensitivity)`.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

The verification internally clones the function for each call. For instance, if you specify the function `func` for inlining and `func` is called twice, the software creates two copies of `func` for verification. The copies are named using the convention `func_pst_inlined_ver` where `ver` is the version number. You see both copies on the **Call Hierarchy** pane.

However, for each run-time check in the function body, you see only one color in your verification results. The semantics of the check color is different from the normal specification.

Red checks:

- Normally, if a function is called twice and an operation causes a definite error only in one of the calls, the check color is orange.
- If you use this option, the color changes to dark orange (shown with an orange exclamation mark in the results list).

Gray checks:

- Normally, if a function is called twice and an `if` statement branch is unreachable in only one of the calls, the branch is shown as reachable.
- If you use this option, the worst color is shown for the check. Therefore, the `if` branch appears gray.

Do not use this option to understand results. Use this option only if a certain function causes scaling issues.

Tips

- Use this option to identify the cause of a **Non-terminating call** error.
 - **Situation:** Sometimes, a red **Non-terminating call** check can appear on a function call though a red check does not appear in the function body. The function body represents all calls to the function. Therefore, if some calls to a function do not cause an error, an orange check appears in the function body.
 - **Action:** If you use this option, for every function call, there is a corresponding function body. Therefore, you can trace a red check on a function call to a red check in the function body.
- Using this option can sometimes duplicate a lot of code and lead to scaling problems. Therefore choose functions to inline carefully.
- Choose functions to inline based on hints provided by the alias verification.
- Do not use this option for entry point functions, including `main`.
- Using this option can increase the number of gray **Unreachable code** checks.

For example, in the following code, if you enter `max` for **Inline**, you obtain two **Unreachable code** checks, one for each call to `max`.

```
int max(int a, int b) {
    return a > b ? a : b;
}

void main() {
    int i=3, j=1, k;
    k=max(i,j);
    i=0;
    k=max(i,j);
}
```

- If you use the keyword `inline` before a function definition, place the definition in a header file and call the function from multiple source files, you have the same result as using the option **Inline**.
- For C++ code, this option applies to all overloaded methods of a class.

Command-Line Information

Parameter: `-inline`

Value: `function1[,function2[,...]]`

No Default

Example (Code Prover): `polyspace-code-prover -sources file_name -inline func1,func2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -inline func1,func2`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Depth of verification inside structures (-k-limiting)

Limit the depth of analysis for nested structures

Description

This option affects a Code Prover analysis only.

Specify a limit to the depth of analysis for nested structures.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Scaling** node.

Command line: Use the option `-k-limiting`. See “Command-Line Information” on page 2-271.

Why Use This Option

Use this option if the analysis is slow because your code has a structure that is many levels deep.

Typically, you see a warning message when a structure with a deep hierarchy is slowing down the verification.

Settings

Default: Full depth of nested structures is analyzed.

Enter a number to specify the depth of analysis for nested structures. For instance, if you specify 0, the analysis does not verify a structure inside a structure.

If you specify a number less than 2, the verification could be less precise.

Command-Line Information

Parameter: `-k-limiting`

Value: *positive integer*

Example (Code Prover): `polyspace-code-prover -sources file_name -k-limiting 3`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -k-limiting 3`

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Generate report

Specify whether to generate a report after the analysis

Description

Specify whether to generate a report along with analysis results.

Depending on the format you specify, you can view this report using an external software. For example, if you specify the format PDF, you can view the report in a pdf reader.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Reporting** node.

Command line: See “Command-Line Information” on page 2-273.

Why Use This Option

You can generate a report from your analysis results for archiving purposes. You can provide this report to your management or clients as proof of code quality.

Using other analysis options, you can tailor the report content and format for your specific needs. See **Bug Finder and Code Prover report** (-report-template) and **Output format** (-report-output-format).

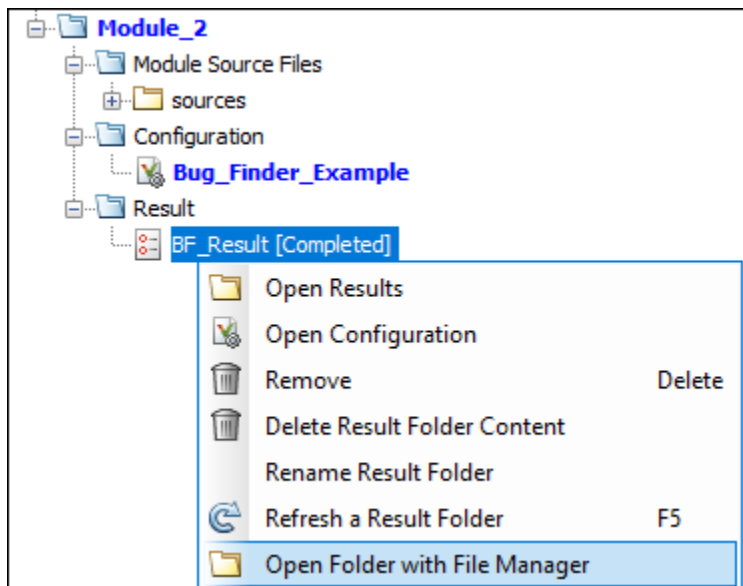
Settings

On

Polyspace generates an analysis report using the template and format you specify.

The report is stored in the Polyspace-Doc subfolder of your results folder.

In Polyspace desktop products, to open your results folder from the user interface, on the **Project Browser** pane, right-click the results node and select **Open Folder with File Manager**.



To change the results folder location, see “Project and Results Folder Contents” (Polyspace Code Prover).

On the command-line, the results folder is the argument of the option `-results-dir`.

Off (default)

Polyspace does not generate an analysis report. You can still view your results in the Polyspace interface.

Tips

This option allows you to specify report generation before starting an analysis.

To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

Command-Line Information

There is no command-line option to solely turn on the report generator. However, using the options `-report-template` for template and `-report-output-format` for output format automatically turns on the report generator.

See Also

Bug Finder and Code Prover report (`-report-template`) | Output format (`-report-output-format`) | `polyspace-report-generator`

Topics

“Prepare Scripts for Polyspace Analysis”

Bug Finder and Code Prover report (-report-template)

Specify template for generating analysis report

Description

Specify template for generating analysis report.

.rpt files for the report templates are available in *polyspaceroot*\toolbox\polyspace\psrptgen\templates\. Here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2019a.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Reporting** node. You have separate options for Bug Finder and Code Prover analysis. See “Dependencies” on page 2-279 for other options you must also enable.

Command line: Use the option -report-template. See “Command-Line Information” on page 2-280.

Why Use This Option

Depending on the template that you use, the report contains information about certain types of results from the **Results List** pane. The template also determines what information is presented in the report and how the information is organized. See the template descriptions below.

Settings - Bug Finder

Default: BugFinderSummary

BugFinder

The report lists:

- **Polyspace Bug Finder Summary:** Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics:** Summary of the various code complexity metrics. For more information, see “Code Metrics” (Polyspace Code Prover Access).
- **Coding Rules:** Coding rule violations in the source code. For each rule violation, the report lists the:
 - Rule number and description.
 - Function containing the rule violation.
 - Review information, such as **Severity**, **Status** and comments.
- **Defects:** Defects found in the source code. For each defect, the report lists the:

- Function containing the defect.
- Defect information on the **Result Details** pane.
- Review information, such as **Severity**, **Status** and comments.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

BugFinderSummary

The report lists:

- **Polyspace Bug Finder Summary**: Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics**: Summary of the various code complexity metrics. For more information, see “Code Metrics” (Polyspace Code Prover Access).
- **Coding Rules Summary**: Coding rules along with number of violations.
- **Defect Summary**: Defects that Polyspace Bug Finder looks for. For each defect, the report lists the:
 - Defect group.
 - Defect name.
 - Number of instances of the defect found in the source code.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. For more information, see “Analysis Options” (Polyspace Bug Finder). If your project has source files with compilation errors, these files are also listed.

If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

CodeMetrics

The report lists the following:

- **Code Metrics Summary**: Various quantities related to the source code. For more information, see “Code Metrics” (Polyspace Code Prover Access).
- **Code Metrics Details**: Various quantities related to the source code with the information broken down by file and function.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

CodingStandards

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, CERT C, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File:** Graph showing each file with number of rule violations.
- **Summary - Violations by Rule:** Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files:** Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules:** Table showing each guideline or rule with number of violations.
- **Violations:** Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

SecurityCWE

The report contains the same information as the BugFinder report. However, in the **Defects** chapter, an additional column lists the CWE™ rules mapped to each defect. The **Configuration Settings** appendix also includes a **Security Standard to Polyspace Result Map**.

Metrics

Only available for results downloaded from the Polyspace Metrics interface.

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives
- Time taken in each phase of analysis
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

Settings - Code Prover

Default: Developer

CodeMetrics

The report contains a summary of code metrics, followed by the complete metrics for an application.

CodingStandards

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File:** Graph showing each file with number of rule violations.
- **Summary - Violations by Rule:** Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files:** Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules:** Table showing each guideline or rule with number of violations.
- **Violations:** Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

Developer

The report lists information useful to developers, including:

- Summary of results
- Coding rule violations
- List of proven run-time errors or red checks
- List of unproven run-time errors or orange checks
- List of unreachable procedures or gray checks
- Global variable usage in code. See “Global Variables” (Polyspace Code Prover Access).

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

DeveloperReview

The report lists the same information as the **Developer** report. However, the reviewed results are sorted by severity and status, and unreviewed results are sorted by file location.

Developer_withGreenChecks

The report lists the same information as the **Developer** report. In addition, the report lists code proven to be error-free or green checks.

Quality

The report lists information useful to quality engineers, including:

- Summary of results
- Statistics about the code
- Graphs showing distributions of checks per file

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

VariableAccess

The report displays the global variable access in your source code. The report first displays the number of global variables of each type. For information on the types, see “Global Variables” (Polyspace Code Prover Access). For each global variable, the report displays the following information:

- Variable name.

The entry for each variable is denoted by |.

- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:

- File and function containing the operation in the form *file_name.function_name*.

The entry for each read or write operation is denoted by ||. Write operations are denoted by < and read operations by >.

- Line and column number of the operation.

This report captures the information available on the **Variable Access** pane in the Polyspace user interface.

CallHierarchy

The report displays the call hierarchy in your source code. For each function call in your source code, the report displays the following information:

- Level of call hierarchy, where the function is called.

Each level is denoted by |. If a function call appears in the table as ||| -> *file_name.function_name*, the function call occurs at the third level of the hierarchy. Beginning from `main` or an entry point, there are three function calls leading to the current call.

- File containing the function call.

In addition, the line and column is also displayed.

- File containing the function definition.

In addition, the line and column where the function definition begins is also displayed.

In addition, the report also displays uncalled functions.

This report captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

SoftwareQualityObjectives

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives
- Time taken in each phase of verification
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface or from results uploaded to the Polyspace Metrics web interface (and then downloaded to the Polyspace user interface) . In each case, you have to set the objectives explicitly in the web interface and then generate the reports.

SoftwareQualityObjectives_Summary

The report contains the same information as the `SoftwareQualityObjectives` report. However, it does not have the supporting appendices with details of code metrics, coding rule violations and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface or from results uploaded to the Polyspace Metrics web interface (and then downloaded to the Polyspace user interface). In each case, you have to set a quality objective level explicitly in the web interface and then generate the reports.

Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the `Generate report` option.

Tips

- This option allows you to specify report generation before starting an analysis.

To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting > Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

- In Bug Finder, the report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user interface or your IDE if you are using a Polyspace plugin.
- If you use the `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives` templates to generate reports, the pass/fail status depends on whether you set the quality objectives level in Polyspace Metrics or Polyspace Access:
 - In Polyspace Access, the pass/fail status is determined based on all results. For instance, if you use the level SQA-4 which sets a threshold of 60% on orange overflow checks, your project has a **FAIL** status if the percentage of green and justified orange overflow checks is less than 60% of *all green and orange overflow checks*.
 - In Polyspace Metrics, the pass/fail status is determined based on a file-by-file basis. The overall status is **FAIL** if one of the files have a **FAIL** status. For instance, if you use the level SQA-4 which sets a threshold of 60% on orange overflow checks, your project has a **FAIL** status if the percentage of green and justified orange overflow checks *in any file* is less than 60% of green and orange overflow checks in that file.

Command-Line Information

Parameter: -report-template

Value: Full path to *template.rpt*

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -report-template *polyspaceroot\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt*

Example (Code Prover): polyspace-code-prover -sources *file_name* -report-template *polyspaceroot\toolbox\polyspace\psrptgen\templates\Developer.rpt*

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -report-template *polyspaceroot\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt*

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -report-template *polyspaceroot\toolbox\polyspace\psrptgen\templates\Developer.rpt*

See Also

Generate report | Output format (-report-output-format) | polyspace-report-generator

Topics

“Prepare Scripts for Polyspace Analysis”

Output format (- report-output-format)

Specify output format of generated report

Description

Specify output format of analysis report.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Reporting** node. See “Dependencies” on page 2-281 for other options you must also enable.

Command line: Use the option `-report-output-format`. See “Command-Line Information” on page 2-282.

Why Use This Option

Use this option to specify whether you want a report in PDF, HTML or another format.

Settings

Default: Word

HTML

Generate report in `.html` format

PDF

Generate report in `.pdf` format

Word

Generate report in `.docx` format.

Tips

- This option allows you to specify report generation before starting an analysis.

To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting > Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

- If the table of contents or graphics in a `.docx` report appear outdated, select the content of the report and refresh the document. Use keyboard shortcuts **Ctrl+A** to select the content and **F9** to refresh it.

Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the **Generate report** option.

Command-Line Information

Parameter: -report-output-format

Value: html | pdf | word

Default: word

Example (Bug Finder): polyspace-bug-finder -sources *file_name* -report-output-format pdf

Example (Code Prover): polyspace-code-prover -sources *file_name* -report-output-format pdf

Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -report-output-format pdf

Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -report-output-format pdf

See Also

Bug Finder and Code Prover report (-report-template) | Generate report | polyspace-report-generator

Topics

“Prepare Scripts for Polyspace Analysis”

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

Enable batch remote analysis

Description

This option applies to the Polyspace desktop products only. The option is used to send the analysis from a desktop to a server (where the analysis runs using the Polyspace server products).

Specify that the analysis must be offloaded to a remote server.

To offload a Polyspace analysis, you need:

- Polyspace Bug Finder Server and/or Polyspace Code Prover Server, and MATLAB Parallel Server™ on the server.
- Polyspace Bug Finder and/or Polyspace Code Prover on the desktop.

See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Set Option

User interface: In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis.

Command line: Use the option `-batch`. See “Command-Line Information” on page 2-284.

Why Use This Option

Use this option if you want the analysis to run on a remote cluster instead of your local desktop.

For instance, you can run remote analysis when:

- You want to shut down your local machine but not interrupt the analysis.
- You want to free execution time on your local machine.
- You want to transfer the analysis to a more powerful computer.

Settings

On

Run batch analysis on a remote computer. In this remote analysis mode, the analysis is queued on a cluster after the compilation phase. Therefore, on your local computer, after the analysis is queued:

- If you are running the analysis from the Polyspace user interface, you can close the user interface.
- If you are running the analysis from the command line, you can close the command-line window.

You can manage the queue from the Polyspace Job Monitor. To use the Polyspace Job Monitor:

- In the Polyspace user interface, select **Tools > Open Job Monitor**. See “Send Polyspace Analysis from Desktop to Remote Servers”.
- On the DOS or UNIX® command line, use the `polyspace-jobs-manager` command. For more information, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”.
- On the MATLAB command line, use the `polyspaceJobsManager` function.

After the analysis, you might have to manually download the results from the cluster.

Off (default)

Do not run batch analysis on a remote computer.

Dependencies

- If you use a third-party scheduler instead of the MATLAB Job Scheduler, add the option `-no-credentials-check`. The credentials check performed in the product is only compatible with the MATLAB Job Scheduler. In the Polyspace user interface, add this option to the **Other** field.
- Do not run a Code Prover analysis on a remote cluster if you run up to the **Verification Level** of **Source Compliance Checking**. For both local and remote analysis, the source compliance checking or compilation phase takes place on your local computer. Therefore, if you are running only up to this phase, run on your local computer.

Command-Line Information

To run a remote analysis from the command line, use with the `-scheduler` option.

Parameter: `-batch`

Value: `-scheduler host_name` if you have not set the **Job scheduler host name** in the Polyspace user interface

Default: Off

Example (Bug Finder): `polyspace-bug-finder -batch -scheduler NodeHost`

`polyspace-bug-finder -batch -scheduler MJSName@NodeHost`

Example (Code Prover): `polyspace-code-prover -batch -scheduler NodeHost`

`polyspace-code-prover -batch -scheduler MJSName@NodeHost`

See Also

`-scheduler`

Topics

“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”

“Prepare Scripts for Polyspace Analysis”

“Send Polyspace Analysis from Desktop to Remote Servers”

“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”

“Send Analysis from Client to Server”

Upload results to Polyspace Metrics (-add-to-results-repository)

Upload analysis results for viewing on Polyspace Metrics web dashboard

Description

This option applies to the Polyspace desktop products only.

Specify upload of analysis results to the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics.

Set Option

User interface: In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis. See “Dependencies” on page 2-285 for other options that you must also enable.

Command line: Use the option `-add-to-results-repository`. See “Command-Line Information” on page 2-286.

Why Use This Option

Polyspace Metrics is a web dashboard that generates code quality metrics from your analysis results. Using this dashboard, you can:

- Provide your management a high-level overview of your code quality.
- Compare your code quality against predefined standards.
- Establish a process where you review in detail only those results that fail to meet standards.
- Track improvements or regression in code quality over time.

See “Generate Code Quality Metrics” (Polyspace Code Prover).

Settings

On

Analysis results are stored in the Polyspace Metrics results repository. This allows you to use a Web browser to view results and code metrics.

The results are not downloaded automatically to your desktop.

Off (default)

Analysis results are stored locally.

Dependencies

The option to upload to Polyspace Metrics is available only if you select Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`).

If you perform a local analysis on your desktop, you can later upload your results to Polyspace Metrics. Right-click your results file and select **Upload to Metrics**.

Command-Line Information

Parameter: `-add-to-results-repository`

Default: Off

Example (Bug Finder): `polyspace-bug-finder -batch -scheduler NodeHost -add-to-results-repository -password passwordName`

Example (Code Prover): `polyspace-code-prover -batch -scheduler NodeHost -add-to-results-repository -password passwordName`

The password is optional.

The upload uses the Polyspace Metrics server that you set up in the Polyspace user interface. See “Set Up Polyspace Metrics” (Polyspace Code Prover). If you want to explicitly specify the Polyspace Metrics server during upload, use the option `-polyspace-metrics-server serverName:portNumber`. For instance:

```
-add-to-results-repository -polyspace-metrics-server localhost:12427
```

See Also

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

Topics

“Set Up Polyspace Metrics” (Polyspace Code Prover)

“Generate Code Quality Metrics” (Polyspace Code Prover)

Command/script to apply after the end of the code verification (-post-analysis-command)

Specify command or script to be executed after analysis

Description

Specify a command or script to be executed after the analysis.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node.

Command line: Use the option `-post-analysis-command`. See “Command-Line Information” on page 2-288.

Why Use This Option

Create scripts for tasks that you want performed after the Polyspace analysis.

For instance, you want to be notified by email that the Polyspace analysis is over. Create a script that sends an email and use this option to execute the script after the Polyspace analysis.

Settings

No Default

Enter full path to the command or script, or click  to navigate to the location of the command or script. After the analysis, this script is executed.

The script is executed in the Polyspace results folder. In your script, consider the results folder as the current folder for relative paths to other files.

For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script. For example, to specify a Perl script `send_email.pl` that sends an email once the analysis is over, enter `polyspaceroot\sys\perl\win32\bin\perl.exe <absolute_path>\send_email.pl`. Here, *polyspaceroot* is the location of the current Polyspace installation, such as `C:\Program Files\Polyspace\R2019a\`, and *<absolute_path>* is the location of the Perl script.

Tips

Running post analysis commands on the server

If you perform verification on a remote server, after verification, the software executes your command on the server, not on the client desktop. If your command executes a script, the script must be present on the server.

For instance, if you specify the command, `/local/utills/send_mail.sh`, the Shell script `send_email.sh` must be present on the server in `/local/utills/`. The software does not copy the

script `send_email.sh` from your desktop to the server before executing the command. If the script is not present on the server, you encounter an error. Sometimes, there are multiple servers that the MATLAB Job Scheduler can run the verification on. Place the script on each of the servers because you do not control which server eventually runs your verification.

Running post analysis commands in the Polyspace user interface

To test the use of this option, run the following Perl script from a folder containing a Polyspace project (`.psprj` file). The script parses the latest Polyspace log file in the folder `Module_1\CP_Result` and writes the current project name and date to a file `report.txt`. The file is saved in `Module_1\CP_Result`.

```
foreach my $file (`ls Module_1\CP_Result\Polyspace_*.log`) {
    open (FH, $file);

    while ($line = <FH>) {
        if ($line =~ m/Ending at: (.*)/) {
            $date=$1;
        }
        if ($line =~ m/-prog=(.*)/) {
            $project=$1;
        }
    }
}

my $filename = 'report.txt';
open(my $fh, '>', $filename) or die "Could not open file '$filename' $!";

print $fh "date=$date\n";
print $fh "project=$project\n";

close $fh;
```

In Linux, you can specify the Perl script for this option.

In Windows, instead of specifying the Perl script directly, specify a `.bat` file that invokes Perl and runs this script. For instance, the `.bat` file can contain the following line (assuming that the `.bat` file and `.pl` file are in the Polyspace project folder). Depending on your MATLAB installation, change the path to `perl.exe` appropriately.

```
"C:\Program Files\MATLAB\R2018b\sys\perl\win32\bin\perl.exe" command.pl
```

Run Code Prover. Check that the folder `Module_1\CP_Result` contains the file `report.txt` with the project name and date.

Command-Line Information

Parameter: `-post-analysis-command`

Value: Path to executable file or command in quotes

No Default

Example in Linux (Bug Finder): `polyspace-bug-finder -sources file_name -post-analysis-command `pwd`/send_email.pl`

Example in Linux (Code Prover): `polyspace-code-prover -sources file_name -post-analysis-command `pwd`/send_email.pl`

Example in Linux (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -post-analysis-command `pwd`/send_email.pl

Example in Linux (Code Prover Server): polyspace-code-prover-server -sources *file_name* -post-analysis-command `pwd`/send_email.pl

Example in Windows: polyspace-bug-finder -sources *file_name* -post-analysis-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"

Note that in Windows, you use the full path to the Perl executable.

See Also

Command/script to apply to preprocessed files (-post-preprocessing-command)

Topics

"Prepare Scripts for Polyspace Analysis"

Automatic Orange Tester (-automatic-orange-tester)

Specify that Automatic Orange Tester must be executed after verification

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify that the Automatic Orange Tester must be executed at the end of the verification.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependency” on page 2-290 for other options you must also enable.

Command line: Use the option -automatic-orange-tester. See “Command-Line Information” on page 2-291.

Why Use This Option

The Automatic Orange Tester runs dynamic tests on your code. The dynamic tests help you determine if an orange check represents a real run-time error or an imprecision of Polyspace analysis. For a tutorial, see “Test Orange Checks for Run-Time Errors” (Polyspace Code Prover).

To run the Automatic Orange Tester after verification, you must select this option *before verification*. During verification, Polyspace generates additional source code to test each orange check for errors. When you run the Automatic Orange Tester later, the software uses this instrumented code for testing.

Settings

On

After verification, when you run the Automatic Orange Tester, Polyspace creates tests for unproven code and runs them.

Off (default)

You cannot launch the Automatic Orange Tester after verification.

Dependency

This option is available only if you set Source code language (-lang) to C or C-CPP.

Tips

- To launch the Automatic Orange Tester, after verification, open your results. Select **Tools > Automatic Orange Tester**.

- When using the automatic orange tester, you cannot:
 - Select **Division round down** under **Target & Compiler**.
 - Select the options `c18`, `tms320c3c`, `x86_64` or `sharc21x61` for **Target & Compiler > Target processor type**.
 - Specify the type `char` as 16-bit or `short` as 8-bit using the option `mcpu...` (Advanced) for **Target & Compiler > Target processor type**. For the same option, you must specify the type `pointer` as 32-bit.
 - Specify global asserts in the code, having the form `Pst_Global_Assert(A,B)`. In global assert mode, you cannot use **Constraint setup** under **Inputs & Stubbing**.
 - Select these options related to floating-point verification: **Subnormal detection mode** and **Consider non finite floats**.

Command-Line Information

Parameter: `-automatic-orange-tester`

Default: Off

Example (Code Prover): `polyspace-code-prover -sources file_name -lang c -automatic-orange-tester`

See Also

Maximum loop iterations (`-automatic-orange-tester-loop-max-iteration`) | Maximum test time (`-automatic-orange-tester-timeout`) | Number of automatic tests (`-automatic-orange-tester-tests-number`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)

“Limitations of Automatic Orange Tester” (Polyspace Code Prover)

Number of automatic tests (-automatic-orange-tester-tests-number)

Specify number of tests that Automatic Orange Tester must run

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify number of tests that you want the Automatic Orange Tester to run. The more the number of tests, the greater the possibility of finding a run-time error, but longer it takes to complete.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 2-292 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-tests-number`. See “Command-Line Information” on page 2-292.

Settings

Default: 500

Enter number of tests up to a maximum of 100,000.

Dependencies

This option is enabled only if you set the following options:

- Set Source code language (`-lang`) to C or C-CPP.
- Specify the option Automatic Orange Tester (`-automatic-orange-tester`).

Command-Line Information

Parameter: `-automatic-orange-tester-tests-number`

Value: *positive integer*

Default: 500

Example (Code Prover): `polyspace-code-prover -sources file_name -lang c -automatic-orange-tester -automatic-orange-tester-tests-number 500`

See Also

Automatic Orange Tester (`-automatic-orange-tester`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)

Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)

Specify number of loop iterations after which Automatic Orange Tester considers infinite loop

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify number of loop iterations after which the Automatic Orange Tester considers the loop to be infinite. Specifying a large number decreases the possibility of identifying an infinite loop incorrectly, but takes more time to complete.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 2-293 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-loop-max-iteration`. See “Command-Line Information” on page 2-293.

Settings

Default: 1000

Enter number of loop iterations. The maximum value that the software supports is 1000.

Dependencies

This option is enabled only if you set the following options:

- Set Source code language (`-lang`) to C or C-CPP.
- Specify the option Automatic Orange Tester (`-automatic-orange-tester`).

Command-Line Information

Parameter: `-automatic-orange-tester-loop-max-iteration`

Value: *positive integer*

Default: 1000

Example (Code Prover): `polyspace-code-prover -sources file_name -lang c -automatic-orange-tester -automatic-orange-tester-loop-max-iteration 500`

See Also

Automatic Orange Tester (`-automatic-orange-tester`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)

Maximum test time (-automatic-orange-tester-timeout)

Specify time in seconds allowed for a single test in Automatic Orange Tester

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify time in seconds allowed for a single test. After this time is over, the Automatic Orange Tester proceeds to the next test. Increasing this time reduces number of tests that do not complete, but increases total verification time.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 2-294 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-timeout`. See “Command-Line Information” on page 2-294.

Settings

Default: 5

Enter time in seconds. The maximum value that the software supports is 60.

Dependencies

This option is enabled only if you set the following options:

- Set Source code language (`-lang`) to C or C-CPP.
- Specify the option Automatic Orange Tester (`-automatic-orange-tester`).

Command-Line Information

Parameter: `-automatic-orange-tester-timeout`

Value: *time*

Default: 5

Example (Code Prover): `polyspace-code-prover -sources file_name -lang c -automatic-orange-tester -automatic-orange-tester-test-timeout 10`

See Also

Automatic Orange Tester (`-automatic-orange-tester`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)

Other

Specify additional flags for analysis

Description

This option is useful only if you run an analysis in the user interface of the Polyspace desktop products.

Enter command-line-style flags such as `-max-processes`.

Set Option

In your project configuration, the option is on the **Advanced Settings** node. You can enter multiple options in this field. If you enter the same option multiple times with different arguments, the analysis uses your last argument.

Why Use This Option

Use this option to add nonofficial or command-line only options to the analyzer.

If you have to add several command line options, you can save them in a text file and specify the file using the option `-options-file`. You can reuse the options file across projects.

Tip

Nonofficial options: In rare circumstances, to work around very specific issues, MathWorks Technical Support might provide you some undocumented options. If you are running verification from the user interface, you use the **Other** field in the **Configuration** pane to enter the options. Sometimes, the options and their arguments have to be preceded by extra flags. When providing you the option, Technical Support will let you know if the extra flags are required.

Possible Flags: `-extra-flags` | `-c-extra-flags` | `-cpp-extra-flags` | `-cfe-extra-flags` | `-il-extra-flags`

Example (Bug Finder): `polyspace-bug-finder -extra-flags -option-name -extra-flags option_param`

Example (Code Prover): `polyspace-code-prover -extra-flags -option-name -extra-flags option_param`

Example (Bug Finder Server): `polyspace-bug-finder-server -extra-flags -option-name -extra-flags option_param`

Example (Code Prover Server): `polyspace-code-prover-server -extra-flags -option-name -extra-flags option_param`

Analysis Options, Command-Line Only

-asm-begin -asm-end

Exclude compiler-specific asm functions from analysis

Syntax

```
-asm-begin "mark1[,mark2,...]" -asm-end "mark1[,mark2,...]"
```

Description

`-asm-begin "mark1[,mark2,...]" -asm-end "mark1[,mark2,...]"` excludes compiler-specific assembly language source code functions from the analysis. You must use these two options together.

Polyspace recognizes most inline assemblers by default. Use the option only if compilation errors occur due to introduction of assembly code. For more information, see “Assembly Code” (Polyspace Code Prover).

Mark the offending code block by two `#pragma` directives, one at the beginning of the assembly code and one at the end. In the command usage, give these marks in the same order for `-asm-begin` as they are for `-asm-end`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

A block of code is delimited by `#pragma start1` and `#pragma end1`. These names must be in the same order for their respective options. Either:

```
-asm-begin "start1" -asm-end "end1"
```

or

```
-asm-begin "mark1,...markN,start1" -asm-end "mark1,...markN,end1"
```

The following example marks two functions for exclusion, `foo_1` and `foo_2`.

Code:

```
#pragma asm_begin_foo
int foo(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_foo

#pragma asm_begin_bar
void bar(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_bar
```

Polyspace Command:

- Bug Finder:

```
polyspace-bug-finder -lang c -asm-begin "asm_begin_foo,asm_begin_bar"  
-asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover:

```
polyspace-code-prover -lang c -asm-begin "asm_begin_foo,asm_begin_bar"  
-asm-end "asm_end_foo,asm_end_bar"
```

- Bug Finder Server:

```
polyspace-bug-finder-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"  
-asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover Server:

```
polyspace-code-prover-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"  
-asm-end "asm_end_foo,asm_end_bar"
```

asm_begin_foo and asm_begin_bar mark the beginning of the assembly source code sections to be ignored. asm_end_foo and asm_end_bar mark the end of those respective sections.

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-author

Specify project author

Syntax

```
-author "value"
```

Description

-author "value" assigns an author to the Polyspace project. The name appears as the project owner in Polyspace Metrics and on generated reports.

The default value is the user name of the current user, given by the DOS or UNIX command `whoami`.

In the user interface of the Polyspace desktop products, select  to specify the Project name, Version, and Author parameters in the **Polyspace Project - Properties** dialog box.

Examples

Assign a project author to your Polyspace Project.

- Bug Finder:

```
polyspace-bug-finder -author "John Smith"
```

- Code Prover:

```
polyspace-code-prover -author "John Smith"
```

- Bug Finder Server:

```
polyspace-bug-finder-server -author "John Smith"
```

- Code Prover Server:

```
polyspace-code-prover-server -author "John Smith"
```

See Also

-date | -prog

Topics

"Prepare Scripts for Polyspace Analysis"

-code-behavior-specifications

Map imprecisely analyzed function to standard function for precise analysis

Syntax

```
-code-behavior-specifications file1[, file2, [...]]
```

Description

`-code-behavior-specifications file1[, file2, [...]]` specifies XML files that allow you to associate behaviors with elements of your code. For instance, you can:

- Map your library functions to corresponding standard functions that Polyspace recognizes. Mapping to standard library functions can help with precision improvement or automatic detection of new threads.
- Specify a function as forbidden.

If you run verification from the command line, specify the absolute path to the XML files or path relative to the folder from which you run the command. If you run verification from the user interface (desktop products only), specify the option along with an absolute path to the XML file in the **Other** field. See **Other**.

A sample template file `code-behavior-specifications-sample.xml` shows the XML syntax. The file is in `polyspaceroot\polyspace\verifier\cxx\` where `polyspaceroot` is the Polyspace installation folder.

Using Option for Precision Improvement

XML Syntax: `<function name="custom_function" std="std_function"> </function>`

Use this entry in the XML file to reduce the number of orange checks from imprecise Code Prover analysis of your function (or false negatives from an imprecise Bug Finder analysis). Sometimes, the verification does not analyze certain kinds of functions precisely because of inherent limitations in static verification. In those cases, if you find a standard function that is a close analog of your function, use this mapping. Though your function itself is not analyzed, the analysis is more precise at the locations where you call the function. For instance, if the verification cannot analyze your function `cos32` precisely and considers full range for its return value, map it to the `cos` function for a return value in `[-1,1]`.

The verification ignores the body of your function. However, the verification emulates your function behavior in the following ways:

- The verification assumes the same return values for your function as the standard function.

For instance, if you map your function `cos32` to the standard function `cos`, the verification assumes that `cos32` returns values in `[-1,1]`.

- The verification checks for the same issues as it checks with the standard function.

For instance, if you map your function `acos32` to the standard function `acos`, the `Invalid use of standard library routine` check determines if the argument of `acos32` is in `[-1,1]`.

The functions that you can map to include:

- Standard library functions from `math.h`.
- Memory management functions from `string.h`.
- `__ps_meminit`: A function specific to Polyspace that initializes a memory area.

Sometimes, the verification does not recognize your memory initialization function and produces an orange `Non-initialized local variable` check on a variable that you initialized through this function. If you know that your memory initialization function initializes the variable through its address, map your function to `__ps_meminit`. The check turns green.

- `__ps_lookup_table_clip`: A function specific to Polyspace that returns a value within the range of the input array.

Sometimes, the verification considers full range for the return values of functions that look up values in large arrays (look-up table functions). If you know that the return value of a look-up table function must be within the range of values in its input array, map the function to `__ps_lookup_table_clip`.

In code generated from models, the verification by default makes this assumption for look-up table functions. To identify if the look-up table uses linear interpolation and no extrapolation, the verification uses the function names. Use the mapping only for handwritten functions, for instance, functions in a C/C++ S-Function block. The names of those functions do not follow specific conventions. You must explicitly specify them.

See also “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” (Polyspace Bug Finder Server).

Using Option for Concurrency Detection

XML Syntax: `<function name="custom_function" std="std_function"> </function>`

Use this entry in the XML file for automatic detection of thread-creation functions and functions that begin and end critical sections. Polyspace supports automatic detection for certain families of multitasking primitives only. Extend the support using this XML entry.

If your thread-creation function, for instance, does not belong to one of the supported families, map your function to a supported concurrency primitive.

See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” (Polyspace Bug Finder Server).

Using Option for Blacklisting Functions

This section applies only to a Bug Finder analysis.

XML Syntax: `<function name="function_name" behavior="FORBIDDEN_FUNC"> </function>`

Use this entry in the XML file to specify a list of functions that you want to prohibit from your source code.

See “Flag Deprecated or Unsafe Functions Using Bug Finder Checkers” (Polyspace Bug Finder Server).

Examples

The examples in the next sections refer to a Code Prover analysis. For Bug Finder examples, see:

- “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” (Polyspace Bug Finder Server)
- “Flag Deprecated or Unsafe Functions Using Bug Finder Checkers” (Polyspace Bug Finder Server)
- “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” (Polyspace Bug Finder Server)

Specify Mapping to Standard Function

You can adapt the sample mapping XML file provided with your Polyspace installation and map your function to a standard function.

Suppose the default verification produces an orange `User` assertion check on this code:

```
double x = acos32(1.0) ;
assert(x <= 2.0);
```

Suppose you know that the function `acos32` behaves like the function `acos` and the return value is 0. You expect the check on the `assert` statement to be green. However, the verification considers that `acos32` returns any value in the range of type `double` because `acos32` is not precisely analyzed. The check is orange. To map your function `acos32` to `acos`:

- 1 Copy the file `code-behavior-specifications-sample.xml` from *polyspaceroot* \polyspace\verifier\cxx\ to another location, for instance, "C:\Polyspace_projects\Common\Config_files". Change the write permissions on the file.
- 2 To map your function to a standard function, modify the contents of the XML file. To map your function `acos32` to the standard library function `acos`, change the following code:

```
<function name="my_lib_cos" std="acos"> </function>
```

To:

```
<function name="acos32" std="acos"> </function>
```

- 3 Specify the location of the file for verification:

- Code Prover:

```
polyspace-code-prover -code-behavior-specifications
"C:\Polyspace_projects\Common\Config_files
\code-behavior-specifications-sample.xml"
```

- Code Prover Server:

```
polyspace-code-prover-server -code-behavior-specifications
"C:\Polyspace_projects\Common\Config_files
\code-behavior-specifications-sample.xml"
```

Specify Mapping to Standard Function with Argument Remapping

Sometimes, the arguments of your function do not map one-to-one with arguments of the standard function. In those cases, remap your function argument to the standard function argument. For instance:

- `__ps_lookup_table_clip`:

This function specific to Polyspace takes only a look-up table array as argument and returns values within the range of the look-up table. Your look-up table function might have additional arguments besides the look-up table array itself. In this case, use argument remapping to specify which argument of your function is the look-up table array.

For instance, suppose a function `my_lookup_table` has the following declaration:

```
double my_lookup_table(double u0, const real_T *table,
                        const double *bp0);
```

The second argument of your function `my_lookup_table` is the look-up table array. In the file `code-behavior-specifications-sample.xml`, add this code:

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
  <mapping std_arg="1" arg="2"></mapping>
</function>
```

When you call the function:

```
res = my_lookup_table(u, table10, bp);
```

The verification interprets the call as:

```
res = __ps_lookup_table_clip(table10);
```

The verification assumes that the value of `res` lies within the range of values in `table10`.

- `__ps_meminit`:

This function specific to Polyspace takes a memory address as the first argument and a number of bytes as the second argument. The function assumes that the bytes in memory starting from the memory address are initialized with a valid value. Your memory initialization function might have additional arguments. In this case, use argument remapping to specify which argument of your function is the starting address and which argument is the number of bytes.

For instance, suppose a function `my_meminit` has the following declaration:

```
void my_meminit(enum InitKind k, void* dest, int is_aligned,
                unsigned int size);
```

The second argument of your function is the starting address and the fourth argument is the number of bytes. In the file `code-behavior-specifications-sample.xml`, add this code:

```
<function name="my_meminit" std="__ps_meminit">
  <mapping std_arg="1" arg="2"></mapping>
  <mapping std_arg="2" arg="4"></mapping>
</function>
```

When you call the function:

```
my_meminit(INIT_START_BY_END, &buffer, 0, sizeof(buffer));
```

The verification interprets the call as:

```
__ps_meminit(&buffer, sizeof(buffer));
```

The verification assumes that `sizeof(buffer)` number of bytes starting from `&buffer` are initialized.

- `memset`: Variable number of arguments.

If your function has variable number of arguments, you cannot map it directly to a standard function without explicit argument remapping. For instance, say your function is declared as:

```
void* my_memset(void*, int, size_t, ...)
```

To map the function to the `memset` function, use the following mapping:

```
<function name="my_memset" std="memset">
  <mapping std_arg="1" arg="1"></mapping>
  <mapping std_arg="2" arg="2"></mapping>
  <mapping std_arg="3" arg="3"></mapping>
</function>
```

Effect of Mapping on Precision

These examples show the result of mapping certain functions to standard functions:

- `my_acos` → `acos`:

If you use the mapping, the `User assertion` check turns green. The verification assumes that the return value of `my_acos` is 0.

- *Before mapping:*

```
double x = my_acos(1.0);
assert(x <= 2.0);
```

- *Mapping specification:*

```
<function name="my_acos" std="acos">
</function>
```

- *After mapping:*

```
double x = my_acos(1.0);
assert(x <= 2.0);
```

- `my_sqrt` → `sqrt`:

If you use the mapping, the `Invalid use of standard library routine` check turns red. Otherwise, the verification does not check whether the argument of `my_sqrt` is nonnegative.

- *Before mapping:*

```
res = my_sqrt(-1.0);
```

- *Mapping specification:*

```
<function name="my_sqrt" std="sqrt">
</function>
```

- *After mapping:*

```
res = my_sqrt(-1.0);
```

- `my_lookup_table` (argument 2) → `__ps_lookup_table_clip` (argument 1):

If you use the mapping, the `User assertion` check turns green. The verification assumes that the return value of `my_lookup_table` is within the range of the look-up table array `table`.

- *Before mapping:*

```
double table[3] = {1.1, 2.2, 3.3}
.
.
double res = my_lookup_table(u, table, bp);
assert(res >= 1.1 && res <= 3.3);
```

- *Mapping specification:*

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
  <mapping std_arg="1" arg="2"></mapping>
</function>
```

- *After mapping:*

```
double table[3] = {1.1, 2.2, 3.3}
.
.
res_real = my_lookup_table(u, table9, bp);
assert(res_real >= 1.1 && res_real <= 3.3);
```

- `my_meminit` → `__ps_meminit`:

If you use the mapping, the `Non-initialized local variable` check turns green. The verification assumes that all fields of the structure `x` are initialized with valid values.

- *Before mapping:*

```
struct X {
  int field1 ;
  int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(struct X));
return x.field1;
```

- *Mapping specification:*

```
<function name="my_meminit" std="__ps_meminit">
  <mapping std_arg="1" arg="1"></mapping>
  <mapping std_arg="2" arg="2"></mapping>
</function>
```

- *After mapping:*

```
struct X {
  int field1 ;
  int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(struct X));
return x.field1;
```

- `my_meminit` → `__ps_meminit`:

If you use the mapping, the `Non-initialized local variable` check turns red. The verification assumes that only the field `field1` of the structure `x` is initialized with valid values.

- *Before mapping:*

```
struct X {
    int field1 ;
    int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(int));
return x.field2;
```

- *Mapping specification:*

```
<function name="my_meminit" std="__ps_meminit">
</function>
```

- *After mapping:*

```
struct X {
    int field1 ;
    int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(int));
return x.field2;
```

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016b

-consider-external-array-access-unsafe

(C++ only) Remove the default assumption that external arrays of unspecified size can be safely accessed at any index

Syntax

`-consider-external-array-access-unsafe`

Description

`-consider-external-array-access-unsafe` removes the default Code Prover assumption in C++ that external arrays of unspecified size can be safely accessed at any index. By default, because of this assumption, Code Prover shows green **Out of bounds array index** checks on external array accesses in C++ code despite their size being unknown. If you use this option, the same check is orange indicating that the access is not proven safe and requires manual inspection.

Note that a Code Prover analysis of C code assumes by default that external array accesses are unsafe.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

Run Code Prover on this example (using a C++ file) with and without the option.

```
extern int arr[];

int getFifthElement(void) {
    return arr[5];
}
```

The array access shows a green **Out of bounds array index** check without the option but an orange check with the option.

See Also

Generic target options | Target processor type (`-target`)

Topics

“Prepare Scripts for Polyspace Analysis”

-custom-target

Create a custom target processor with specific data type sizes

Syntax

-custom-target *target_sizes*

Description

-custom-target *target_sizes* defines a custom target processor for the Polyspace analysis. The target processor definition includes sizes in bytes of fundamental data types, signedness of plain char, alignment of structures and underlying types of standard typedef-s such as `size_t`, `ptrdiff_t` and `wchar_t`.

target_sizes is a comma-separated list specifying these values. From left to right, the values are the following. If a data type is not supported, -1 is used for its size.

Specification	Possible Values
Whether plain char is signed	true or false
Size of char in bits	Number
Other sizes are in bytes.	
Size of short	Number
Size of int	Number
Size of short long	Number
Size of long	Number
Size of long long	Number
Size of float	Number
Size of double	Number
Size of long double	Number
Size of pointer	Number
Maximum alignment of all integer types	Number
Maximum alignment of variables of type struct or union	Number
Endianness	little or big
Underlying type of <code>size_t</code>	unknown, unsigned_char, unsigned_short, unsigned_int, unsigned_long, or unsigned_long_long
Underlying type of <code>ptrdiff_t</code>	unknown, signed_char, short, int, long, or long_long
Underlying type of <code>wchar_t</code>	unknown, short, unsigned_short, int, unsigned_int, long, or unsigned_long

Typically, this option is used when the `polyspace-configure` command creates an options file for the subsequent Polyspace analysis. However, you can directly enter this option when manually writing options files. This option is useful in situations where your target specifications are not covered by one of the predefined target processors. See [Target processor type \(-target\)](#).

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

Examples

An usage of the option looks like this:

```
-custom-target false,8,2,4,-1,4,8,4,8,8,4,8,1,little,unsigned_int,int,unsigned_int
```

The option argument translates to the following target specification.

Specification	Possible Values
Whether plain char is signed	false
Size of char	8 bits
Size of short	2 bytes
Size of int	4 bytes
Size of short long	short long is not supported.
Size of long	4 bytes
Size of long long	8 bytes
Size of float	4 bytes
Size of double	8 bytes
Size of long double	8 bytes
Size of pointer	4 bytes
Maximum alignment of all integer types	8 bytes
Maximum alignment of variables of type struct or union	1 byte
Endianness	little
Underlying type of size_t	unsigned_int
Underlying type of ptrdiff_t	int
Underlying type of wchar_t	unsigned_int

See Also

[Generic target options](#) | [Target processor type \(-target\)](#)

Topics

“Prepare Scripts for Polyspace Analysis”

-date

Specify date of analysis

Syntax

`-date "date"`

Description

`-date "date"` specifies the date stamp for the analysis in the format dd/mm/yyyy. By default the value is the date the analysis starts.

Examples

Assign a date to your Polyspace Project:

- Bug Finder:
`polyspace-bug-finder -date "15/03/2012"`
- Code Prover:
`polyspace-code-prover -date "15/03/2012"`
- Bug Finder Server:
`polyspace-bug-finder-server -date "15/03/2012"`
- Code Prover Server:
`polyspace-code-prover-server -date "15/03/2012"`

See Also

`-author` | `-date`

Topics

"Prepare Scripts for Polyspace Analysis"

-doc | -documentation

Display Polyspace documentation in help browser

Syntax

```
-doc  
-documentation
```

Description

`-doc` and `-documentation` opens Polyspace documentation in a help browser. You can see information such as getting started, workflows and reference pages for commands and analysis options. You can also search through the documentation in the help browser.

Examples

Display Polyspace documentation in a help browser:

- Bug Finder:

```
polyspace-bug-finder -doc  
polyspace-bug-finder -documentation
```

- Code Prover:

```
polyspace-code-prover -doc  
polyspace-code-prover -documentation
```

- Bug Finder Server:

```
polyspace-bug-finder-server -doc  
polyspace-bug-finder-server -documentation
```

- Code Prover Server:

```
polyspace-code-prover-server -doc  
polyspace-code-prover-server -documentation
```

See Also

```
-h[elp]
```

-dump-preprocessing-info

Show all macros implicitly defined during a particular analysis

Syntax

-dump-preprocessing-info

Description

-dump-preprocessing-info prints all the macros implicitly defined (or undefined) during a particular Polyspace analysis. The macro definitions come from:

- Your specification for the option `Compiler` (-compiler)
Polyspace emulates a compiler by defining the compiler-specific macros.
- Macros defined (or undefined) in the Polyspace implementation of Standard Library headers
- Macros that you explicitly define (or undefine) using the options `Preprocessor definitions` (-D) and `Disabled preprocessor definitions` (-U)

Use this option only if you want to know how Polyspace defines a specific macro. In case you want to use a different definition for the macro, you can then override the current definition.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**. On the **Output Summary** pane, you can see each macro definition on a separate line. You can search for the macro name in the user interface and click the line with the macro name to see further details in the **Detail** pane.

Examples

Suppose that you use the ARM v6 compiler for building your source code. For the Polyspace analysis, you use the value `armclang` for the option `Compiler` (-compiler). Suppose that you want to know what Polyspace uses as definition for the macro `__ARM_ARCH`.

- 1 Enter the following command and pipe the console output to a file that you can search later:

```
polyspace-bug-finder -sources aFile.c -compiler armclang -dump-preprocessing-info
```

`aFile.c` can be a simple C file. You can also replace `polyspace-bug-finder` with `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.

- 2 Search for `__ARM_ARCH` in the file containing the console output. You can see the line with the macro definition:

```
Remark: Definition of macro __ARM_ARCH (pre-processing __polyspace_stdstubs.c)
|#define __ARM_ARCH 8
|defined by syntax extension xml file
|predefined macro
```

In this example, the macro is set to the value 8.

- To override this macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine this macro, use the option `Disabled preprocessor definitions (-U)`.

See Also

`Compiler (-compiler)`

Topics

"Prepare Scripts for Polyspace Analysis"

-generate-launching-script-for

Extract information from project file

Syntax

`-generate-launching-script-for` *PRJFILE*

Description

`-generate-launching-script-for` *PRJFILE* extracts information from a project file *PRJFILE* (created in the user interface of the Polyspace desktop products) so that you can run an analysis from the command line. For each project module and each configuration in each module, a folder is created containing the following files::

- `source_command.txt` — List of source files for the `-sources-list-file` option.
- `options_command.txt` — List of the analysis options for the `-options-file` option.
- `temporal_exclusions.txt` — List of temporal exclusions, generated only if you specify the Temporarily exclusive tasks (`-temporal-exclusions-file`) option.
- `.polyspace_conf.psprj` — A copy of the project file Polyspace used to generate the scripting files.
- `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — shell script that calls the correct commands. The script also calls any options that cannot be given to the `-options-file` command, such as `-batch` or `-add-to-results-repository`. You can give this file additional analysis options as parameters.

Note The script that Polyspace generates runs the same analysis that Polyspace runs from the user interface. If your project runs in the Polyspace user interface, the script will run from the command line.

Examples

Extract information to run `myproject` from the command line. Use this option with the desktop binary `polyspace`:

- Bug Finder:


```
polyspace -generate-launching-script-for myproject.psprj -bug-finder
```
- Code Prover:


```
polyspace -generate-launching-script-for myproject.psprj
```

See Also

Topics

“Configure Polyspace Analysis Options in User Interface and Generate Scripts”

-h | -help

Display list of possible options

Syntax

-h
-help

Description

-h and -help display the list of possible options in the command window along with option argument syntax.

Examples

Display the command-line help:

- Bug Finder:

```
polyspace-bug-finder -h  
polyspace-bug-finder -help
```

- Code Prover:

```
polyspace-code-prover -h  
polyspace-code-prover -help
```

- Bug Finder Server:

```
polyspace-bug-finder-server -h  
polyspace-bug-finder-server -help
```

- Code Prover Server:

```
polyspace-code-prover-server -h  
polyspace-code-prover-server -help
```

-doc | -documentation

-I

Specify include folder for compilation

Syntax

`-I folder`

Description

`-I folder` specifies a folder that contains include files required for compiling your sources. You can specify only one folder for each instance of `-I`. However, you can specify this option multiple times.

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

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

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

The analysis automatically includes the `./sources` folder (if it exists) after the include folders that you specify.

Examples

Include two folders with the analysis:

- Bug Finder:

```
polyspace-bug-finder -I /com1/inc -I /com1/sys/inc
```

- Code Prover:

```
polyspace-code-prover -I /com1/inc -I /com1/sys/inc
```

- Bug Finder Server:

```
polyspace-bug-finder-server -I /com1/inc -I /com1/sys/inc
```

- Code Prover Server:

```
polyspace-code-prover-server -I /com1/inc -I /com1/sys/inc
```

The source folder is implicitly included. Include files in the source folder can be found automatically without explicit inclusion of the source folder with the `-I` option.

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-import-comments

Import review information from previous analysis

Syntax

```
-import-comments resultsFolder
```

Description

`-import-comments resultsFolder` imports the review information (status, severity and additional notes) from a previous analysis, as specified by the results folder.

You can import review information from the same type of results only. For instance:

- You cannot import review information from a results of a Bug Finder checker to a Code Prover run-time check. Even when the checker names sound similar, the underlying semantics of Bug Finder and Code Prover can be different. The only exception is checkers for coding rules. You can import comments between Bug Finder and Code Prover for coding rule violations.
- You cannot import review information from results of a file-by-file verification in Code Prover to results of a regular Code Prover verification.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

Import review information from the previous results:

- Bug Finder:

```
polyspace-bug-finder -sources filename  
-import-comments C:\Results\myProj\1.2
```

- Code Prover:

```
polyspace-code-prover -sources filename  
-import-comments C:\Results\myProj\1.2
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources filename  
-import-comments C:\Results\myProj\1.2
```

- Code Prover Server:

```
polyspace-code-prover-server -sources filename  
-import-comments C:\Results\myProj\1.2
```

See Also

`-v[ersion] | polyspace-comments-import`

Topics

“Import Review Information from Previous Polyspace Analysis”

-list-all-values

Display valid option arguments for a given command-line option

Syntax

```
-list-all-values option
```

Description

`-list-all-values option` displays all the valid option arguments for the command-line option *option*.

Examples

Display the valid option arguments for option `-misra3`:

- Polyspace Bug Finder:

```
polyspace-bug-finder -list-all-values -misra3
```
- Polyspace Code Prover:

```
polyspace-code-prover -list-all-values -misra3
```
- Polyspace Bug Finder Server:

```
polyspace-bug-finder-server -list-all-values -misra3
```
- Polyspace Code Prover Server:

```
polyspace-code-prover-server -list-all-values -misra3
```

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2020a

-max-processes

Specify maximum number of processors for analysis

Syntax

```
-max-processes num
```

Description

`-max-processes num` specifies the maximum number of processes that you want the analysis to use. On a multicore system, the software parallelizes the analysis and creates the specified number of processes to speed up the analysis. The valid range of *num* is 1 to 128.

Unless you specify this option, a Code Prover verification uses up to four processes. If you have fewer than four processes, the verification uses the maximum available number. To increase or restrict the number of processes, use this option.

Unless you specify this option, a Bug Finder analysis uses the maximum number of available processes. Use this option to restrict the number of processes used.

To use this option effectively, determine the number of processors available for use. If the number of processes you create is greater than the number of processors available, the analysis does not benefit from the parallelization. Check the system information in your operating system.

Note that when you start a verification, a message states the number of logical processors detected on your system. However, the analysis is parallelized to the physical processor cores on a machine. Multithreading implementations such as hyper-threading is not taken into account.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

Disable parallel processing during the analysis:

- Bug Finder:

```
polyspace-bug-finder -max-processes 1
```
- Code Prover:

```
polyspace-code-prover -max-processes 1
```
- Bug Finder Server:

```
polyspace-bug-finder-server -max-processes 1
```
- Code Prover Server:

```
polyspace-code-prover-server -max-processes 1
```

Tips

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 processes.

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-no-assumption-on-absolute-addresses

Remove assumption that absolute address usage is valid

Syntax

-no-assumption-on-absolute-addresses

Description

This option affects a Code Prover analysis only.

-no-assumption-on-absolute-addresses removes the default assumption that absolute addresses used in your code are valid. If you use this option, the verification produces an orange **Absolute address usage** check when you assign an absolute address to a pointer. Otherwise, the check is green by default.

The type of the pointer to which you assign the address determines the initial value stored in the address. For instance, if you assign the address to an `int*` pointer, following this check, the verification assumes that the memory zone that the address points to is initialized with an `int` value. The value can be anything allowed for the data type `int`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

The use of option -no-assumption-on-absolute-addresses can increase the number of orange checks in your code. For instance, the following table shows an additional orange check with the option enabled.

Absolute Address Usage Green	Absolute Address Usage Orange
<pre>void main() { int *p = (int *)0x32; int x; x=*p; }</pre>	<pre>void main() { int *p = (int *)0x32; int x; x=*p; }</pre>
<p>In this example, the software produces:</p> <ul style="list-style-type: none"> A green Absolute address usage check when the address 0x32 is assigned to a pointer p. A green Illegally dereferenced pointer check when the pointer p is read. <p>x potentially has all values allowed for an <code>int</code> variable.</p>	<p>In this example, the software produces:</p> <ul style="list-style-type: none"> An orange Absolute address usage check when the address 0x32 is assigned to a pointer p. A green Illegally dereferenced pointer check when the pointer p is read. <p>x potentially has all values allowed for an <code>int</code> variable.</p>

For best use of the **Absolute address usage** check, leave this check green by default during initial stages of development. During integration stage, use the option -no-assumption-on-absolute-

addresses and detect all uses of absolute memory addresses. Browse through them and make sure that the addresses are valid.

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

Introduced in R2016a

-non-preemptable-tasks

Specify functions that represent nonpreemptable tasks

Syntax

```
-non-preemptable-tasks function1[,function2[,...]]
```

Description

This option affects a Bug Finder analysis only.

`-non-preemptable-tasks function1[,function2[,...]]` specifies functions that represent nonpreemptable tasks.

The functions cannot be interrupted by other noncyclic tasks and cyclic tasks but can be interrupted by interrupts, preemptable or nonpreemptable. Noncyclic tasks are specified with the option `Tasks (-entry-points)`, cyclic tasks with the option `Cyclic tasks (-cyclic-tasks)` and interrupts with the option `Interrupts (-interrupts)`. For examples, see “Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server).

To specify a function as a nonpreemptable cyclic task, you must first specify the function as a cyclic or noncyclic task. The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Critical section details (-critical-section-begin -critical-section-end)` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”

“Protections for Shared Variables in Multitasking Code”

“Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server)

“Concurrency Defects” (Polyspace Bug Finder Access)

Introduced in R2016b

-options-for-sources

Specify analysis options specific to a source file

Syntax

`-options-for-sources filename options`

Description

`-options-for-sources filename options` associates a semicolon-separated list of Polyspace analysis options with the source file specified by *filename*.

This option is primarily used when the `polyspace-configure` command creates an options file for the subsequent Polyspace analysis. The option `-options-for-sources` associates a group of analysis options such as include folders and macro definitions with specific source files.

However, you can directly enter this option when manually writing options files. This option is useful in situations where you want to associate a group of options with a specific source file without applying it to other files.

In the user interface of the Polyspace desktop products, you can create a Polyspace project from your build command. The project uses the option `-options-for-sources` to associate specific Polyspace analysis options with specific files. However, when you open the project in the user interface, you cannot see the use of this option. Open the project in a text editor to see this option.

Examples

In this sample options file, the include folder `/usr/lib/gcc/x86_64-linux-gnu/6/include` and the macros `__STDC_VERSION__` and `__GNUC__` are associated only with the source file `file.c` and not `fileAnother.c`.

```
-options-for-sources file.c;-I /usr/lib/gcc/x86_64-linux-gnu/6/include;  
-options-for-sources file.c;-D __STDC_VERSION__=201112L;-D __GNUC__=6;  
-sources file.c  
-sources fileAnother.c
```

For the options used in this example, see:

- `-sources`
- `-I`
- Preprocessor definitions (`-D`)

See Also

`-options-file` | `polyspace-configure`

Topics

“Prepare Scripts for Polyspace Analysis”

-preemptable-interrupts

Specify functions that represent preemptable interrupts

Syntax

```
-preemptable-interrupts function1[,function2[,...]]
```

Description

This option affects a Bug Finder analysis only.

`-preemptable-interrupts function1[,function2[,...]]` specifies functions that represent preemptable interrupts.

The function acts as an interrupt in every way except that it can be interrupted by other interrupts, preemptable or nonpreemptable. Interrupts are specified with the option `Interrupts (-interrupts)`. For examples, see “Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server).

To specify a function as a preemptable interrupt, you must first specify the function as an interrupt. The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | Critical section details (`-critical-section-begin` `-critical-section-end`) | Cyclic tasks (`-cyclic-tasks`) | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | Temporally exclusive tasks (`-temporal-exclusions-file`)

Topics

“Prepare Scripts for Polyspace Analysis”

“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”

“Protections for Shared Variables in Multitasking Code”

“Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder Server)

“Concurrency Defects” (Polyspace Bug Finder Access)

Introduced in R2016b

-options-file

Run Polyspace using list of options

Syntax

-options-file *file*

Description

-options-file *file* specifies a file which lists your analysis options. The file must be a text file with each option on a separate line. Use # to add comments to this file.

Examples

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

- Bug Finder or Bug Finder Server:

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

- Code Prover or Code Prover Server:

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

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

- Bug Finder:

```
polyspace-bug-finder -options-file listofoptions.txt
```

- Code Prover:

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

- Bug Finder Server:

```
polyspace-bug-finder-server -options-file listofoptions.txt
```

- Code Prover Server:

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

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-prog

Specify name of project

Syntax

`-prog projectName`

Description

`-prog projectName` specifies a name for your Polyspace project. This name must use only letters, numbers, underscores (`_`), dashes (`-`), or periods (`.`).

The name appears in the analysis log and a few other places.

Examples

Assign a name to your Polyspace project:

- Bug Finder:
`polyspace-bug-finder -prog MyApp`
- Code Prover:
`polyspace-code-prover -prog MyApp`
- Bug Finder Server:
`polyspace-bug-finder-server -prog MyApp`
- Code Prover Server:
`polyspace-code-prover-server -prog MyApp`

See Also

`-author` | `-date`

Topics

“Prepare Scripts for Polyspace Analysis”

-regex-replace-rgx -regex-replace-fmt

Make replacements in preprocessor directives

Syntax

```
-regex-replace-rgx matchFileName -regex-replace-fmt replacementFileName
```

Description

`-regex-replace-rgx matchFileName -regex-replace-fmt replacementFileName` replaces tokens in preprocessor directives for the purposes of Polyspace analysis. The original source code is unchanged. You match a token using a regular expression in the file *matchFileName* and replace the token using a replacement in the file *replacementFileName*.

Use this option only to replace or remove tokens in the preprocessor directives *before preprocessing*. If a token in your source code causes a compilation error, you can typically replace or remove the token from the preprocessed code. Use the more convenient option `Command/script to apply to preprocessed files (-post-preprocessing-command)`. You cannot make the replacements in preprocessed code only for tokens in preprocessor directives.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

In the user interface, specify absolute paths to the text files with the search and replace patterns.

Examples

Suppose you want to replace `_rom_beg` in this `#define` directive:

```
#define ROM_BEG_ADDR (uint16*)&_rom_beg
```

and modify the directive to:

```
#define ROM_BEG_ADDR (0x4000u)
```

Specify this regular expression in a file `match.txt`:

```
^\s*#define\s+ROM_BEG_ADDR\s+\(uint16_t\*\)\(\&_rom_beg\)
```

These elements are used in the regular expression:

- `^` asserts position at the start of a line.
- `\s*` represents zero or more whitespace characters.
- `\s+` represents one or more whitespace characters.

The characters `*`, `&`, `(` and `)` in the original expression are escaped with `\`. For a complete list of regular expressions, see Perl documentation.

Specify the replacement in a file `replace.txt`.

```
#define ROM_BEG_ADDR \ (0x4000u\)
```

Specify the two text files during analysis with the options `-regex-replace-rgx` and `-regex-replace-fmt`:

- Bug Finder:

```
polyspace-bug-finder -sources filename
                    -regex-replace-rgx match.txt
                    -regex-replace-fmt replace.txt
```

- Code Prover:

```
polyspace-code-prover -sources filename
                    -regex-replace-rgx match.txt
                    -regex-replace-fmt replace.txt
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources filename
                            -regex-replace-rgx match.txt
                            -regex-replace-fmt replace.txt
```

- Code Prover Server:

```
polyspace-code-prover-server -sources filename
                             -regex-replace-rgx match.txt
                             -regex-replace-fmt replace.txt
```

See Also

Command/script to apply to preprocessed files (`-post-preprocessing-command`)

Topics

“Prepare Scripts for Polyspace Analysis”

-report-output-name

Specify name of report

Syntax

-report-output-name *reportName*

Description

-report-output-name *reportName* specifies the name of an analysis report.

The default name for a report is *Prog_Template.Format*:

- *Prog* is the name of the project specified by -prog.
- *TemplateName* is the type of report template specified by -report-template.
- *Format* is the file extension for the report specified by -report-output-format.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

Specify the name of the analysis report:

- Bug Finder:

```
polyspace-bug-finder -report-template Developer
                    -report-output-name Airbag_v3.doc
```

- Code Prover:

```
polyspace-code-prover -report-template Developer
                    -report-output-name Airbag_v3.doc
```

- Bug Finder Server:

```
polyspace-bug-finder-server -report-template Developer
                          -report-output-name Airbag_v3.doc
```

- Code Prover Server:

```
polyspace-code-prover-server -report-template Developer
                          -report-output-name Airbag_v3.doc
```

See Also

Bug Finder and Code Prover report (-report-template) | Output format (-report-output-format)

Topics

“Prepare Scripts for Polyspace Analysis”

-results-dir

Specify the results folder

Syntax

```
-results-dir resultsFolder
```

Description

`-results-dir resultsFolder` specifies where to save the analysis results. The default location at the command line is the current folder.

If you are running analysis in the user interface of the Polyspace desktop products, see “Run Polyspace Analysis on Desktop” (Polyspace Code Prover).

Examples

Specify to store your results in the RESULTS folder:

- Bug Finder:

```
polyspace-bug-finder -results-dir RESULTS
```

- Code Prover:

```
polyspace-code-prover -results-dir RESULTS
```

- Bug Finder Server:

```
polyspace-bug-finder-server -results-dir RESULTS
```

- Code Prover Server:

```
polyspace-code-prover-server -results-dir RESULTS
```

You can create the name of the results folder based on the verification date and time. For instance, in a Bash shell, enter these commands to create a variable `RESULTS` that begins with `results_` and contains the current date and time:

```
export DATETIME=$(date +%d%B_%HH%M_%A)
export RESULTS=results_$DATE
```

You can then use the variable `RESULTS` as argument of the option `-results-dir`:

```
-results-dir $RESULTS
```

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-scheduler

Specify cluster or job scheduler

Syntax

`-scheduler schedulingOption`

Description

`-scheduler schedulingOption` specifies the head node of the MATLAB Parallel Server cluster that manages Polyspace analysis submissions from multiple clients and allocates the analysis to worker nodes. You use this option along with the option `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)` to offload an analysis from a desktop to a remote cluster. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Examples

Run a batch analysis on a remote server using one of these syntaxes for the job scheduler:

- Bug Finder:

```
polyspace-bug-finder -batch -scheduler NodeHost
polyspace-bug-finder -batch -scheduler 192.168.1.124:12400
polyspace-bug-finder -batch -scheduler MJSName@NodeHost
```

- Code Prover:

```
polyspace-code-prover -batch -scheduler NodeHost
polyspace-code-prover -batch -scheduler 192.168.1.124:12400
polyspace-code-prover -batch -scheduler MJSName@NodeHost
```

For details, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”.

You can track the status of the job using the `polyspace-jobs-manager` command:

```
polyspace-jobs-manager listjobs -scheduler NodeHost
```

See Also

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

Topics

“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”

“Send Code Prover Analysis from Desktop to Locally Hosted Server”

“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”

-sources

Specify source files

Syntax

```
-sources file1[,file2,...]
-sources file1 -sources file2
```

Description

`-sources file1[,file2,...]` or `-sources file1 -sources file2` specifies the list of source files that you want to analyze. You can use standard UNIX wildcards with this option to specify your sources.

The source files are compiled in the order in which they are specified.

Examples

Analyze the files `mymain.c`, `funAlgebra.c`, and `funGeometry.c`.

- Bug Finder:

```
polyspace-bug-finder -sources mymain.c
                    -sources funAlgebra.c -sources funGeometry.c
```

- Code Prover:

```
polyspace-code-prover -sources mymain.c
                    -sources funAlgebra.c -sources funGeometry.c
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources mymain.c
                            -sources funAlgebra.c -sources funGeometry.c
```

- Code Prover Server:

```
polyspace-code-prover-server -sources mymain.c
                             -sources funAlgebra.c -sources funGeometry.c
```

See Also

`-sources-list-file` | `polyspace-configure`

Topics

“Prepare Scripts for Polyspace Analysis”

-sources-list-file

Specify file containing list of sources

Syntax

`-sources-list-file file_path`

Description

`-sources-list-file file_path` specifies the absolute path to a text file that lists each file name that you want to analyze.

To specify your sources in the text file, on each line, specify the path to a source file. You can specify an absolute path or a path relative to the folder from which you are running the analysis. For example:

```
C:\Sources\myfile.c
C:\Sources2\myfile2.c
```

Examples

Run analysis on files listed in `files.txt`:

- Bug Finder:

```
polyspace-bug-finder -sources-list-file "C:\Analysis\files.txt"
polyspace-bug-finder -sources-list-file "/home/polyspace/files.txt"
```

- Code Prover:

```
polyspace-code-prover -sources-list-file "C:\Analysis\files.txt"
polyspace-code-prover -sources-list-file "/home/polyspace/files.txt"
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources-list-file "C:\Analysis\files.txt"
polyspace-bug-finder-server -sources-list-file "/home/polyspace/files.txt"
```

- Code Prover Server:

```
polyspace-code-prover-server -sources-list-file "C:\Analysis\files.txt"
polyspace-code-prover-server -sources-list-file "/home/polyspace/files.txt"
```

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-submit-job-from-previous-compilation-results

Specify that the analysis job must be resubmitted without recompilation

Syntax

```
-submit-job-from-previous-compilation-results
```

Description

`-submit-job-from-previous-compilation-results` specifies that the Polyspace analysis must start after the compilation phase with compilation results from a previous analysis. The option is primarily useful when offloading a Polyspace analysis from desktops to remote servers. If a remote analysis stops after compilation, for instance because of communication problems between the server and client computers, use this option. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

When you perform a remote analysis:

- 1 On the local host computer, the Polyspace software performs code compilation and coding rule checking.
- 2 The analysis job is then submitted to the MATLAB job scheduler on the head node of the MATLAB Parallel Server cluster.
- 3 The head node of the MATLAB Parallel Server cluster assigns the verification job to a worker node, where the remaining phases of the Polyspace analysis occur.

If an analysis stops after completing the first step and you restart the analysis, use this option to reuse compilation results from the previous analysis. You thereby avoid restarting the analysis from the compilation phase.

If previous compilation results do not exist in the current folder, an error occurs. Remove the option and restart analysis from the compilation phase.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

Specify remote analysis with compilation results from a previous analysis:

- Bug Finder:

```
polyspace-bug-finder -batch -scheduler localhost  
-submit-job-from-previous-compilation-results
```

- Code Prover:

```
polyspace-code-prover -batch -scheduler localhost  
-submit-job-from-previous-compilation-results
```

See Also

Topics

“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”

“Send Code Prover Analysis from Desktop to Locally Hosted Server”

“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”

-tmp-dir-in-results-dir

Keep temporary files in results folder

Syntax

```
-tmp-dir-in-results-dir
```

Description

`-tmp-dir-in-results-dir` specifies that temporary files must be stored in a subfolder of the results folder. Use this option only when the standard temporary folder does not have enough disk space. If the results folder is mounted on a network drive, this option can slow down your processor.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files” (Polyspace Code Prover).

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See **Other**.

Examples

Store temporary files in the results folder:

- Bug Finder:

```
polyspace-bug-finder -tmp-dir-in-results-dir
```

- Code Prover:

```
polyspace-code-prover -tmp-dir-in-results-dir
```

- Bug Finder Server:

```
polyspace-bug-finder-server -tmp-dir-in-results-dir
```

- Code Prover Server:

```
polyspace-code-prover-server -tmp-dir-in-results-dir
```

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-v | -version

Display Polyspace version number

Syntax

-v
-version

Description

-v or -version displays the version number of your Polyspace product.

Examples

Display the version number and release of your Polyspace product:

- Bug Finder:
`polyspace-bug-finder -v`
- Code Prover:
`polyspace-code-prover -v`
- Bug Finder Server:
`polyspace-bug-finder-server -v`
- Code Prover Server:
`polyspace-code-prover-server -v`

-verif-version

Assign a version identifier

Syntax

```
-verif-version id
```

Description

`-verif-version id` assigns an identifier, *id*, to identify the analysis. You can use this identifier to refer to different analyses at the command line. For example, you can import comments from a previous analysis using the identifier.

Examples

Assign a verification identifier:

- Bug Finder:

```
polyspace-bug-finder -verif-version 1.3
```
- Code Prover:

```
polyspace-code-prover -verif-version 1.3
```
- Bug Finder Server:

```
polyspace-bug-finder-server -verif-version 1.3
```
- Code Prover Server:

```
polyspace-code-prover-server -verif-version 1.3
```

See Also

Topics

“Prepare Scripts for Polyspace Analysis”

-xml-annotations-description

Apply custom code annotations to Polyspace analysis results

Syntax

```
-xml-annotations-description file_path
```

Description

`-xml-annotations-description file_path` uses the annotation syntax defined in the XML file located in *file_path* to interpret code annotations in your source files. You can use the XML file to specify an annotation syntax and map it to the Polyspace annotation syntax. When you run an analysis by using this option, you can justify and hide results with annotations that use your syntax. If you run Polyspace at the command line, *file_path* is the absolute path or path relative to the folder from which you run the command. If you run Polyspace through the user interface, *file_path* is the absolute path.

If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See **Other**.

Why Use This Option

If you have existing annotations from previous code reviews, you can import these annotations to Polyspace. You do not have to review and justify results that you have already annotated. Similarly, if your code comments must adhere to a specific format, you can map and import that format to Polyspace.

Examples

Import Existing Annotations for Coding Rule Violations

Suppose that you have previously reviewed source file `zero_div.c` containing the following code, and justified certain MISRA C: 2012 violations by using custom annotations.

```
#include <stdio.h>

/* Violation of Misra C:2012
rules 8.4 and 8.7 on the next
line of code. */

int func(int p) //My_rule 50, 51
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

/* Violation of Misra C:2012
rule 8.4 on the next line of
code */

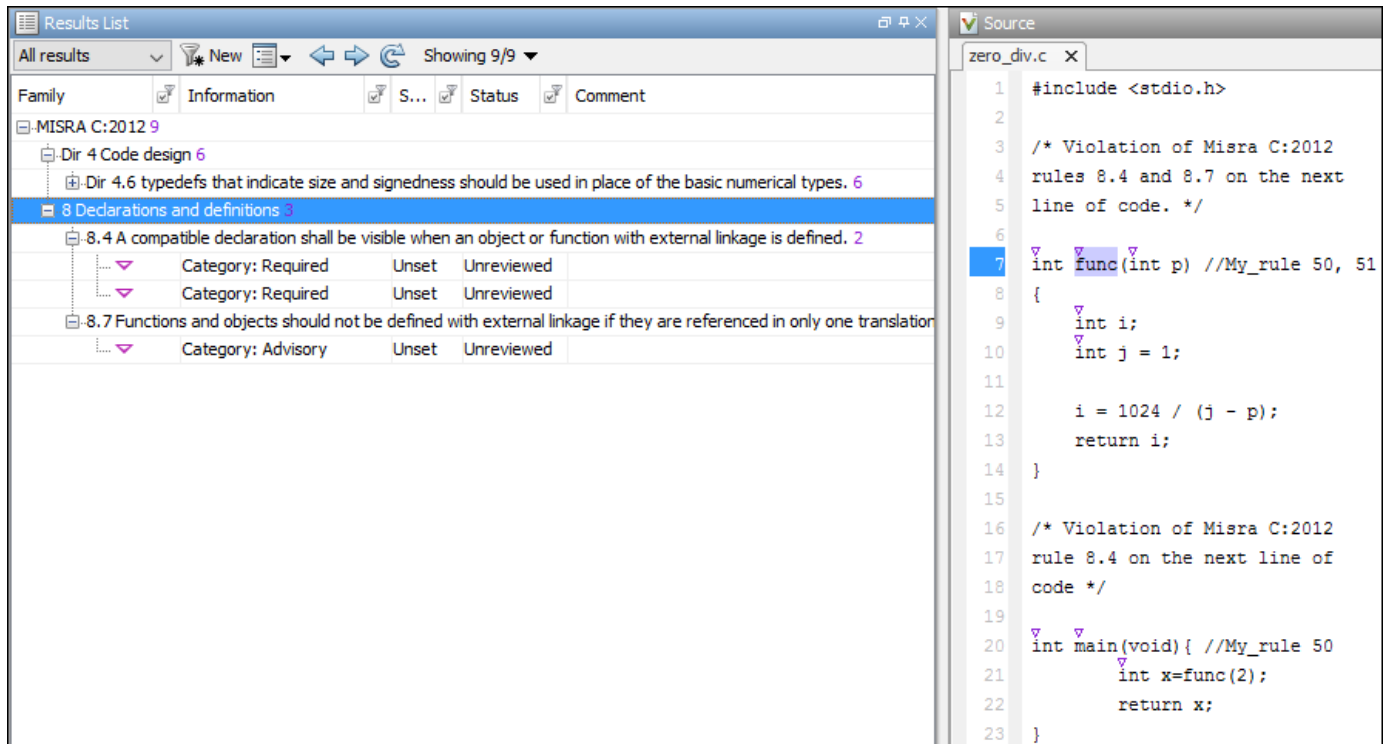
int main(void){ //My_rule 50
    int x=func(2);
    return x;
}
```

The code comments **My_rule 50, 51** and **My_rule 50** do not use the Polyspace annotation syntax. Instead, you use a convention where you place all MISRA rules in a single numbered list. In this list, rules 8.4 and 8.7 correspond to the numbers 50 and 51. You can check this code for MISRA C: 2012 violations by typing the command:

- Bug Finder:
`polyspace-bug-finder -sources source_path -misra3 all`
- Code Prover:
`polyspace-code-prover -sources source_path -misra3 all`
- Bug Finder Server:
`polyspace-bug-finder-server -sources source_path -misra3 all`
- Code Prover Server:
`polyspace-code-prover-server -sources source_path -misra3 all`

source_path is the path to `zero_div.c`.

The annotated violations appear in the **Results List** pane. You must review and justify them again.



This XML example defines the annotation format used in `zero_div.c` and maps it to the Polyspace annotation syntax:

- The format of the annotation is the keyword `My_rule`, followed by a space and one or more comma-separated alphanumeric rule identifiers.
- Rule identifiers 50 and 51 are mapped to MISRA C: 2012 rules 8.4 and 8.7 respectively. The mapping uses the Polyspace annotation syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="example annotation">
```

```
  <Expressions Search_For_Keywords="My_rule"
    Separator_Result_Name="," >
```

```
    <!-- This section defines the annotation syntax format -->
    <Expression Mode="SAME_LINE"
      Regex="My_rule\s(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />
```

```
</Expressions>
```

```
  <!-- This section maps the user annotation to the Polyspace
  annotation syntax -->
```

```
  <Mapping>
```

```
  <Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
```

```
  <Result_Name_Mapping Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
```

```
</Mapping>  
</Annotations>
```

To import the existing annotations and apply them to the corresponding Polyspace results:

- 1 Copy the preceding code example to a text editor and save it on your machine as `annotations_description.xml`, for instance in `C:\Polyspace_workspace\annotations\`.
- 2 Rerun the analysis on `zero_div.c` by using the command:

- Bug Finder:

```
polyspace-bug-finder -sources source_path -misra3 all ^  
-xml-annotations-description ^  
C:\Polyspace_workspace\annotations\annotations_description.xml
```

- Code Prover:

```
polyspace-code-prover -sources source_path -misra3 all ^  
-xml-annotations-description ^  
C:\Polyspace_workspace\annotations\annotations_description.xml
```

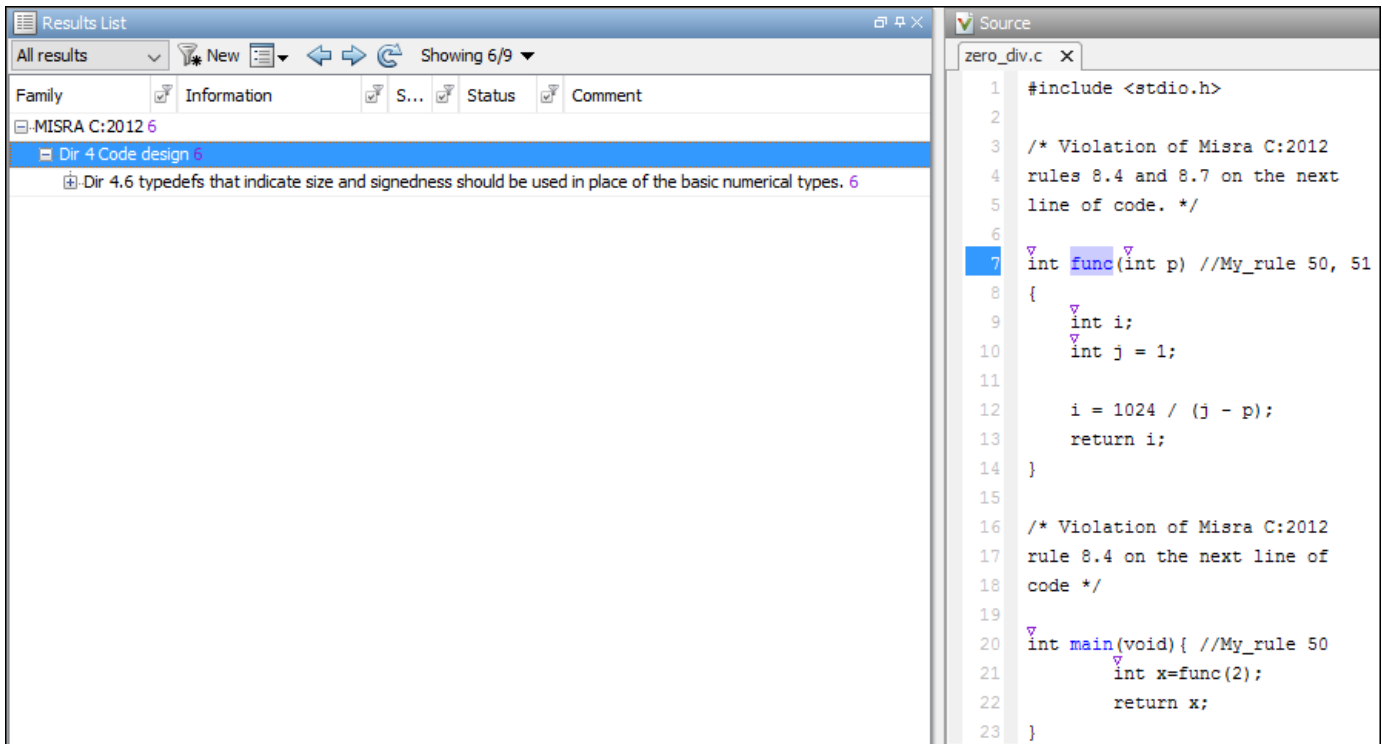
- Bug Finder Server:

```
polyspace-bug-finder-server -sources source_path -misra3 all ^  
-xml-annotations-description ^  
C:\Polyspace_workspace\annotations\annotations_description.xml
```

- Code Prover Server:

```
polyspace-code-prover-server -sources source_path -misra3 all ^  
-xml-annotations-description ^  
C:\Polyspace_workspace\annotations\annotations_description.xml
```

Polyspace considers the annotated results justified and hides them in the **Results List** pane.



See Also

Topics

"Prepare Scripts for Polyspace Analysis"

"Define Custom Annotation Format" (Polyspace Code Prover)

"Annotation Description Full XML Template" (Polyspace Code Prover)

Introduced in R2017b

