Polyspace® Code Prover™

User's Guide

MATLAB® & SIMULINK®

R2015a

MathWorks®

How to Contact MathWorks

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

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

Polyspace® Code Prover™ User's Guide

Trademarks

Patents

Revision History

September 2013	Online Only	Revised for Version 9.0 (Release 2013b)
March 2014	Online Only	Revised for Version 9.1 (Release 2014a)
October 2014	Online Only	Revised for Version 9.2 (Release 2014b)
March 2015	Online Only	Revised for Version 9.3 (Release 2015a)

Contents

1 Introduction to Polyspace Products

2 How to Use Polyspace Software

3 Setting Up Project in User Interface

4 Setting Up Polyspace User Interface

5 Emulating Your Runtime Environment

6 Preparing Source Code for Verification

7 Running a Verification

8

Troubleshooting Verification Problems

9 Reviewing Verification Results

10 Reviewing Checks

11 Managing Orange Checks

12 Coding Rule Sets and Concepts

13 Checking Coding Rules

14 Software Quality with Polyspace Metrics

15 Configure Model for Code Analysis

16 Model Link for Polyspace Code Prover

17 Configure Code Analysis Options

18 Run Polyspace on Generated Code

19 Using Polyspace Software in the Eclipse IDE

20 Using Polyspace Software in Visual Studio

Glossary

Introduction to Polyspace Products

Polyspace Verification

In this section...

Polyspace Verification

Polyspace products verify C, C++, and Ada code by detecting run-time errors before code is compiled and executed.

To verify the source code, you set up verification parameters in a project, run the verification, and review the results. A graphical user interface helps you to efficiently review verification results. The software assigns a color to operations in the source code as follows:

- **Green** – Indicates that the operation is proven to not have certain kinds of error.
- **Red** – Indicates that the operation is proven to have at least one error.
- **Gray** – Indicates unreachable code.
- **Orange** – Indicates that the operation can have an error along some execution paths.

The color-coding helps you to quickly identify errors and find the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

Value of Polyspace Verification

Polyspace verification can help you to:

- "Enhance Software Reliability" on page 1-2
- "Decrease Development Time" on page 1-3
- "Improve the Development Process" on page 1-4

Enhance Software Reliability

Polyspace software enhances the reliability of your C/C++ applications by proving code correctness and identifying run-time errors. Using advanced verification techniques, Polyspace software performs an exhaustive verification of your source code.

Because Polyspace software verifies all executions of your code, it can identify code that:

- Never has an error
- Always has an error
- Is unreachable
- Might have an error

With this information, you know how much of your code does not contain run-time errors, and you can improve the reliability of your code by fixing errors.

You can also improve the quality of your code by using Polyspace verification software to check that your code complies with established coding standards, such as the MISRA C®, MISRA® C++ or JSF® C++ standards.[1]

Decrease Development Time

Polyspace software reduces development time by automating the verification process and helping you to efficiently review verification results. You can use it at any point in the development process. However, using it during early coding phases allows you to find errors when it is less costly to fix them.

You use Polyspace software to verify source code before compile time. To verify the source code, you set up verification parameters in a project, run the verification, and review the results. This process takes significantly less time than using manual methods or using tools that require you to modify code or run test cases.

Color-coding of results helps you to quickly identify errors. You will spend less time debugging because you can see the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

Polyspace verification software helps you to use your time effectively. Because you know the parts of your code that do not have errors, you can focus on the code with proven (red code) or potential errors (orange code).

Reviewing code that might have errors (orange code) can be time-consuming, but Polyspace software helps you with the review process. You can use filters to focus on certain types of errors or you can allow the software to identify the code that you should review.

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

Improve the Development Process

Polyspace software makes it easy to share verification parameters and results, allowing the development team to work together to improve product reliability. Once verification parameters have been set up, developers can reuse them for other files in the same application.

Polyspace verification software supports code verification throughout the development process:

- An individual developer can find and fix run-time errors during the initial coding phase.
- Quality assurance engineers can check overall reliability of an application.
- Managers can monitor application reliability by generating reports from the verification results.

How Polyspace Verification Works

Polyspace software uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as run-time debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the Polyspace verification are true for all executions of the software.

What is Static Verification

Static verification is a broad term, and is applicable to any tool that derives dynamic properties of a program without executing the program. However, most static verification tools only verify the complexity of the software, in a search for constructs that may be potentially erroneous. Polyspace verification provides deep-level verification identifying almost all run-time errors and possible access conflicts with global shared data.

Polyspace verification works by approximating the software under verification, using representative approximations of software operations and data.

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
{    tab[i] = foo(i);
}
```

To check that the variable `i` never overflows the range of `tab`, a traditional approach would be to enumerate each possible value of `i`. One thousand checks would be required.

Using the static verification approach, the variable `i` is modelled by its domain variation. For instance, the model of `i` is that it belongs to the static interval [0..999]. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborate models are also used for this purpose).

By definition, an approximation leads to information loss. For instance, the information that `i` is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the domain variation of `i` is smaller than the range of `tab`. Only one check is required to establish that — and hence the gain in efficiency compared to traditional approaches.

Static code verification has an exact solution. However, this exact solution is not practical, as it would require the enumeration of all possible test cases. As a result, approximation is required for a usable tool.

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of a program variable is a superset of its actual variation domain. As a result, Polyspace verifies run-time error items that require checking.

Related Products

In this section...

Polyspace Bug Finder

For information about Polyspace Bug Finder™ , see http://www.mathworks.com/products/polyspace-bug-finder/.

Polyspace Products for Verifying Ada Code

For information about Polyspace products that verify Ada code, see the following:

http://www.mathworks.com/products/polyspaceclientada/

http://www.mathworks.com/products/polyspaceserverada/

Tool Qualification and Certification

You can use the DO Qualification Kit and IEC Certification Kit products to qualify Polyspace Products for C/C++ for DO and IEC Certification.

To view the artifacts available with these kits, use the Certification Artifacts Explorer. Artifacts included in the kits are not accessible from the MathWorks® web site.

For more information on the IEC Certification Kit, see IEC Certification Kit (for ISO 26262 and IEC 61508).

For more information on the DO Qualification Kit, see DO Qualification Kit (for DO-178).

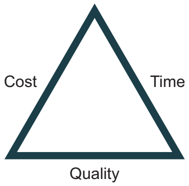
How to Use Polyspace Software

Polyspace Verification and the Software Development Cycle

In this section...
"Software Quality and Productivity" on page 2-2
"Best Practices for Verification Workflow" on page 2-3

Software Quality and Productivity

The goal of most software development teams is to maximize both quality and productivity. However, when developing software, there are three related variables to consider: cost, quality, and time.



Changing the requirements for one of these variables affects the other two.

Generally, the criticality of your application determines the balance between these three variables – your quality model. With classical testing processes, development teams generally try to achieve their quality model by testing all modules in an application until each module meets the required quality level. Unfortunately, this process often ends before quality requirements are met, because the available time or budget has been exhausted.

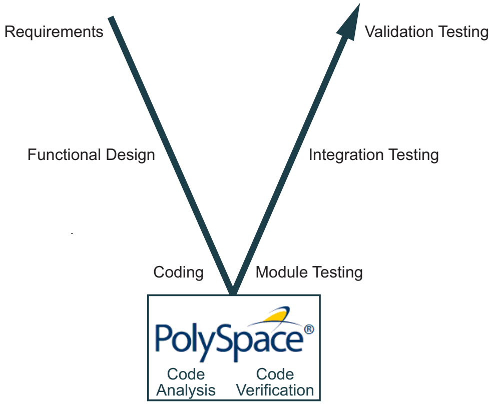
Polyspace verification allows a different process. Polyspace verification can support both productivity improvement and quality improvement at the same time. However, you must balance the aims of these activities.

You should not perform code verification at the end of the development process. To achieve maximum quality and productivity, integrate verification into your development process, considering time and cost restrictions.

This section describes how to integrate Polyspace verification into your software development cycle. It explains both how to use Polyspace verification in your current development process, and how to change your process to get more out of verification.

Best Practices for Verification Workflow

Polyspace verification can be used throughout the software development cycle. However, to maximize both quality and productivity, the most efficient time to use it is early in the development cycle.



Polyspace Verification in the Development Cycle

Typically, verification is conducted in two stages. First, you verify code as it is written, to check coding rules and quickly identify obvious defects. Once the code is stable, you verify it again before module/unit testing, with more stringent verification and review criteria.

Using verification early in the development cycle improves both quality and productivity, because it allows you to find and manage defects soon after the code is written. This saves time because each user is familiar with their own code, and can quickly determine why the code contains defects. In addition, defects are cheaper to fix at this stage, since they can be addressed before the code is integrated into a larger system.

Implement Process for Verification

In this section...

Overview of the Polyspace Process

Polyspace verification cannot magically produce quality code at the end of the development process. However, if you integrate Polyspace verification into your development process, Polyspace verification helps you to measure the quality of your code, identify issues, and ultimately achieve your own quality goals.

To implement Polyspace verification within your development process, you must perform each of the following steps:

1. Define your quality goals.
2. Define a process to match your quality goals.
3. Apply the process to assess the quality of your code.
4. Improve the process.

Define Process to Meet Your Goals

Once you have defined your quality goals, you must define a process that allows you to meet those goals. Defining the process involves actions both within and outside Polyspace software.

These actions include:

- Communicating coding standards (coding rules) to your development team.
- Setting Polyspace analysis options. For more information, see "Specify Analysis Options".
- Setting review criteria for consistent review of results. For more information, see "Limit Display of Orange Checks".

Apply Process to Assess Code Quality

Once you have defined a process that meets your quality goals, it is up to your development and testing teams to apply it consistently to all software components.

This process includes:

1. Running a Polyspace verification on each software component as it is written.
2. Reviewing verification results consistently. See "Add Review Comments to Results".
3. Saving review comments for each component, so they are available for future review. See "Import Review Comments from Previous Verifications".
4. Performing additional verifications on each component, as defined by your quality goals.

Improve Your Verification Process

Once you review initial verification results, you can assess both the overall quality of your code, and how well the process meets your requirements for software quality, development time, and cost restrictions.

Based on these factors, you may want to take actions to modify your process. These actions may include:

- Reassessing your quality goals.
- Changing your development process to produce code that is easier to verify.
- Changing Polyspace analysis options to improve the precision of the verification.
- Changing Polyspace options to change how verification results are reported.

For more information, see "Reduce Orange Checks".

Sample Workflows for Polyspace Verification

In this section...

Overview of Verification Workflows

Polyspace verification supports two goals at the same time:

- Reducing the cost of testing and validation
- Improving software quality

You can use Polyspace verification in different ways depending on your development context and quality model.

This section provides sample workflows that show how to use Polyspace verification in a variety of development contexts.

Software Developers and Testers – Standard Development Process

User Description

This workflow applies to software developers and test groups using a standard development process, where coding rules are not used or followed consistently.

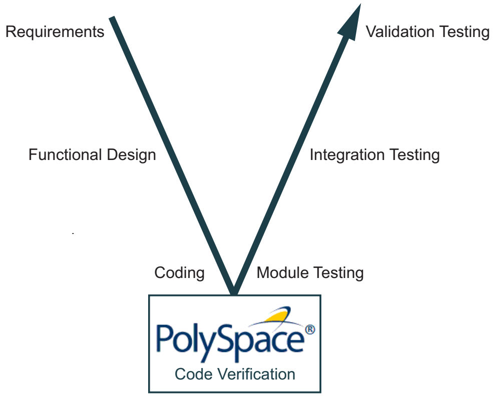
Quality

The main goal of Polyspace verification is to improve productivity while maintaining or improving software quality. Verification helps developers and testers find and fix bugs more quickly than other processes. It also improves software quality by identifying bugs that otherwise might remain in the software.

In this process, the goal is not to completely prove the absence of errors. The goal is to deliver code of equal or better quality that other processes, while optimizing productivity to provide a predictable time frame with minimal delays and costs.

Verification Workflow

This process involves file-by-file verification immediately after coding, and again just before functional testing.



The verification workflow consists of the following steps:

1. The project leader configures a Polyspace project to perform robustness verification, using default Polyspace options.

   Note: This means that verification uses the automatically generated “main” function. This main will call unused procedures and functions with full range parameters.

2. Each developer performs file-by-file verification as they write code, and reviews verification results.
3. The developer fixes **red** errors and examines **gray** code identified by the verification.
4. Until coding is complete, the developer repeats steps 2 and 3 as required..
5. Once a developer considers a file complete, they perform a final verification.
6. The developer fixes **red** errors, examines **gray** code, and performs a selective orange review.

Note: The goal of the selective orange review is to find as many bugs as possible within a limited period of time.

Using this approach, it is possible that some bugs may remain in unchecked oranges. However, the verification process represents a significant improvement from other testing methods.

Costs and Benefits

When using verification to detect bugs:

- **Red and gray checks** – Reviewing red and gray checks provides a quick method to identify real run-time errors in the code.
- **Orange checks** – Selective orange review provides a method to identify potential run-time errors as quickly as possible. The time required to find one bug varies from 5 minutes to 1 hour, and is typically around 30 minutes. This represents an average of two minutes per orange check review, and a total of 20 orange checks per package in Ada and 60 orange checks per file in C.

Disadvantages to this approach:

- **Number of orange checks –** If you do not use coding rules, your verification results will contain more orange checks.
- **Unreviewed orange checks** – Some bugs may remain in unchecked oranges.

Software Developers and Testers – Rigorous Development Process

User Description

This workflow applies to software developers and test engineers working within development groups. These users are often developing software for embedded systems, and typically use coding rules.

These users typically want to find bugs early in the development cycle using a tool that is fast and iterative.

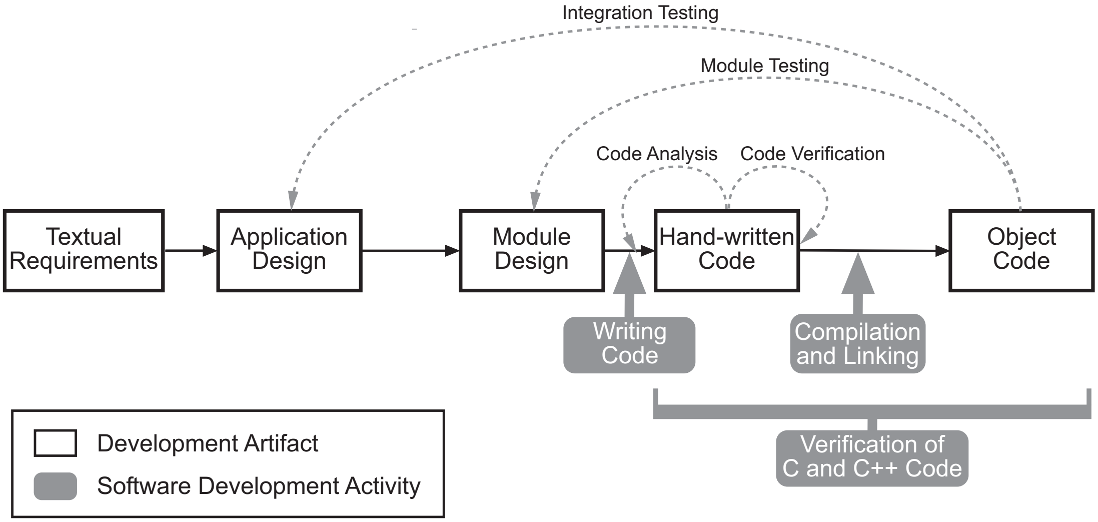
Quality

The goal of Polyspace verification is to improve software quality with equal or increased productivity.

Verification can prove the absence of run-time errors, while helping developers and testers to find and fix defects efficiently.

Verification Workflow

This process involves both code analysis and code verification during the coding phase, and thorough review of verification results before module testing. It may also involve integration analysis before integration testing.



Workflow for Code Verification

Note: Solid arrows in the figure indicate the progression of software development activities.

The verification workflow consists of the following steps:

1. The project leader configures a Polyspace project to perform contextual verification. This involves:

   - Using Data Range Specifications (DRS) to define initialization ranges for input data. For example, if a variable "x" is read by functions in the file, and if x can be initialized to any value between 1 and 10, this information should be included in the DRS file.

- Creates a "main" program to model call sequence, instead of using the automatically generated main.
- Sets options to check the properties of some output variables. For example, if a variable "y" is returned by a function in the file and should always be returned with a value in the range 1 to 100, then Polyspace can flag instances where that range of values might be breached.

2. The project leader configures the project to check the required coding rules.
3. Each developer performs file-by-file verification as they write code, and reviews both coding rule violations and verification results.
4. The developer fixes coding rule violations and **red** errors, examines **gray** code, and performs a selective orange review.
5. Until coding is complete, the developer repeats steps 2 and 3 as required.
6. Once a developer considers a file complete, they perform a final verification.
7. The developer or tester performs an exhaustive orange review on the remaining orange checks.

Note: The goal of the exhaustive orange review is to examine orange checks that are not reviewed as part of selective reviews. When you fix coding rule violations, the total number of orange checks is reduced, and the remaining orange checks are likely to reveal problems with the code.

Optionally, an additional verification can be performed during the integration phase. The purpose of this additional verification is to track integration bugs, and review:

- Red and gray integration checks;
- The remaining orange checks with a selective review: *Integration bug tracking*.

Costs and Benefits

With this approach, Polyspace verification typically provides the following benefits:

- Fewer orange checks in the verification results (improved selectivity). The number of orange checks is typically reduced to 3–5 per file, yielding an average of 1 bug. Often, several of the orange checks represent the same bug.
- Fewer gray checks in the verification results.

- Typically, each file requires two verifications before it can be checked-in to the configuration management system.
- The average verification time is about 15 minutes.

Note: If the development process includes data rules that determine the data flow design, the benefits might be greater. Using data rules reduces the potential of verification finding integration bugs.

If performing the optional verification to find integration bugs, you may see the following results. On a typical 50,000 line project:

- A selective orange review may reveal **one integration bug per hour** of code review.
- Selective orange review takes about 6 hours to complete. This is long enough to review orange checks throughout the whole application and represents a step towards an exhaustive orange check review. Spending more time is unlikely to be efficient.
- An exhaustive orange review would take between 4 and 6 days, assuming that 50,000 lines of code contains approximately 400–800 orange checks. Exhaustive orange review is typically recommended only for high-integrity code, where the consequences of a potential error justify the cost of the review.

Quality Engineers – Code Acceptance Criteria

User Description

This workflow applies to quality engineers who work outside of software development groups, and are responsible for independent verification of software quality and adherence to standards.

These users generally receive code late in the development cycle, and may even be verifying code that is written by outside suppliers or other external companies. They are concerned with not just detecting bugs, but measuring quality over time, and developing processes to measure, control, and improve product quality going forward.

Quality

The main goal of Polyspace verification is to control and evaluate the safety of an application.

The criteria used to evaluate code can vary widely depending on the nature of the application. For example:

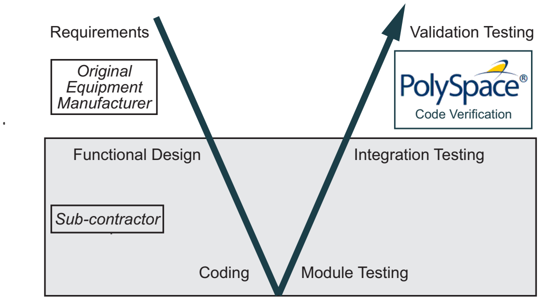
- You may be satisfied with zero red checks.
- In addition to zero red checks, you may want to conduct an exhaustive orange check review.

Typically, these criteria become increasingly stringent as a project advances from early, to intermediate, and eventually to final delivery.

For more information on defining these criteria, see "Customize Software Quality Objectives".

Verification Workflow

This process usually involves both code analysis and code verification before validation phase, and thorough review of verification results based on defined quality goals.



Note: Verification is often performed multiple times, as multiple versions of the software are delivered.

The verification workflow consists of the following steps:

1. Quality engineering group defines clear quality goals for the code to be written, including specific quality levels for each version of the code to be delivered (first, intermediate, or final delivery) For more information, see "Customize Software Quality Objectives".
2. Development group writes code according to established standards.
3. Development group delivers software to the quality engineering group.
4. The project leader configures the Polyspace project to meet the defined quality goals, as described in "Define Process to Meet Your Goals" on page 2-4.

5. Quality engineers perform verification on the code.
6. Quality engineers review **red** errors, **gray** code, and the number of orange checks defined in the process.

   Note: The number of orange checks reviewed often depends on the version of software being tested (first, intermediate, or final delivery). This can be defined by quality level (see "Define Broad Requirements for Verification").

7. Quality engineers create reports documenting the results of the verification, and communicate those results to the supplier.
8. Quality engineers repeat steps 5–7 for each version of the code delivered.

Costs and Benefits

The benefits of code verification at this stage are the same as with other verification processes, but the cost of correcting faults is higher, because verification takes place late in the development cycle.

It is possible to perform an exhaustive orange review at this stage, but the cost of doing so can be high. If you want to review all orange checks at this phase, it is important to use development and verification processes that minimize the number of orange checks. This includes:

- Developing code using strict coding and data rules.
- Providing accurate manual stubs for unresolved function calls.
- Using DRS to provide accurate data ranges for input variables.

Taking these steps will minimize the number of orange checks reported by the verification, and make it more likely that remaining orange checks represent real issues with the software.

Quality Engineers – Certification/Qualification

User Description

This workflow applies to quality engineers who work with applications requiring outside quality certification, such as IEC 61508 certification or DO-178 qualification.

These users must perform a set of activities to meet certification requirements.

You can use the “IEC Certification Kit (for ISO 26262 and IEC 61508)” to help qualify Polyspace products within an IEC 61508, ISO 26262, EN 50128, or other related functional-safety standard certification environment.

You can use the “DO Qualification Kit (for DO-178)” to help qualify Polyspace products within an DO-178 qualification environment.

Model-Based Design Users — Verifying Generated Code

User Description

This workflow applies to users who have adopted model-based design to generate code for embedded application software.

These users generally use Polyspace software in combination with several other MathWorks products, including Simulink®, Embedded Coder® , and Simulink Design Verifier™ products. In many cases, these customers combine application components that are manually written code with those created using generated code.

Quality

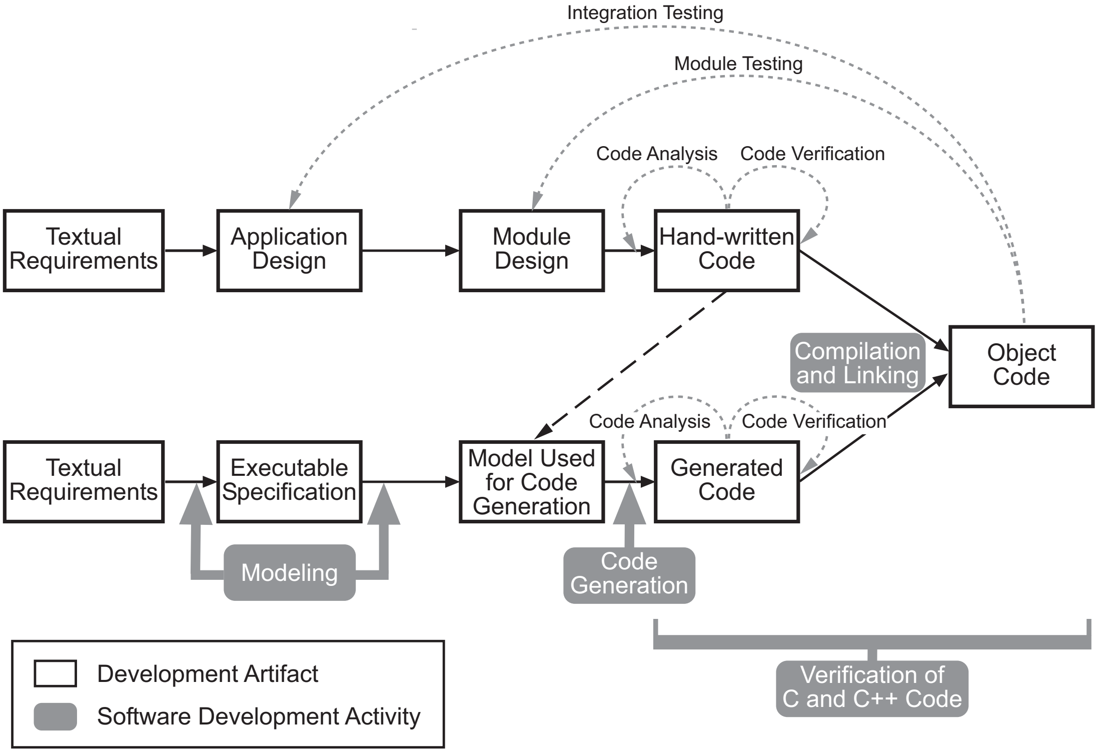
The goal of Polyspace verification is to improve the quality of the software by identifying implementation issues in the code, and proving that the code is both semantically and logically correct.

Polyspace verification allows you to find run-time errors:

- In hand-coded portions within the generated code
- In the model used for production code generation
- In the integration of manually written and generated code

Verification Workflow

The workflow is different for manually written code, generated code, and mixed code. Polyspace products can perform code verification as part of any of these workflows. The following figure shows a suggested verification workflow for manually written and mixed code.



Workflow for Verification of Generated and Mixed Code

Note: Solid arrows in the figure indicate the progression of software development activities.

The verification workflow consists of the following steps:

1. The project leader configures a Polyspace project to meet defined quality goals.
2. Developers manually code sections of the application.
3. Developers or testers perform **Polyspace verification** of manually coded sections within the generated code, and review verification results according to the established quality goals.
4. Developers create Simulink model based on requirements.

5 Developers validate model to prove it is logically correct (using tools such as Simulink Model Advisor, and the Simulink Verification and Validation™ and Simulink Design Verifier products).

6 Developers generate code from the model.

7 Developers or testers perform **Polyspace verification** on the entire software component, including both manually written and generated code.

8 Developers or testers review verification results according to the established quality goals.

Note: Polyspace Code Prover allows you to quickly track issues identified by the verification back to the block in the Simulink model.

Costs and Benefits

Simulink Design Verifier verification can identify errors in textual designs or executable models that are not identified by other methods. The following table shows how errors in textual designs or executable models can appear in the resulting code.

Examples of Common Run-Time Errors

Type of Error	Design or Model Errors	Code Errors
Arithmetic errors	• Incorrect Scaling • Unknown calibrations • Untested data ranges	• Overflows/Underflows • Division by zero • Square root of negative numbers
Memory corruption	• Incorrect array specification in state machines • Incorrect legacy code (look-up tables)	• Out of bound array indexes • Pointer arithmetic
Data truncation	• Unexpected data flow	• Overflows/Underflows • Wrap-around
Logic errors	• Unreachable states • Incorrect Transitions	• Non initialized data • Dead code

Project Managers — Integrating Polyspace Verification with Configuration Management Tools

User Description

This workflow applies to project managers responsible for establishing check-in criteria for code at different development stages.

Quality

The goal of Polyspace verification is to test that code meets established quality criteria before being checked in at each development stage.

Verification Workflow

The verification workflow consists of the following steps:

1. Project manager defines quality goals, including individual quality levels for each stage of the development cycle.
2. Project leader configures a Polyspace project to meet quality goals.
3. Developers or testers run verification at the following stages:
   - Daily check-in — On the files currently under development. Compilation must complete without the permissive option.
   - Pre-unit test check-in — On the files currently under development.
   - Pre-integration test check-in — On the whole project, ensuring that compilation can complete without the permissive option. This stage differs from daily check-in because link errors are highlighted.
   - Pre-build for integration test check-in — On the whole project, with multitasking aspects accounted for as required.
   - Pre-peer review check-in — On the whole project, with multitasking aspects accounted for as required.
4. Developers or testers review verification results for each check-in activity to confirm the code meets the required quality level. For example, the transition criterion could be: “No defect found in 20 minutes of selective orange review”

Define Your Requirements

Before launching verification, define your requirements from the verification process. Defining your requirements helps decide which analysis options and results are relevant for you.

In this section...
"Define Broad Requirements for Verification" on page 2-18
"Define Specific Requirements for Verification" on page 2-19

Define Broad Requirements for Verification

This example shows how to define your broad requirements before you begin a Polyspace Code Prover verification, and then implement them in your verification process.

1 Prepare a set of quality levels for your application. A quality level chart can be like this:

Software Quality Levels

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
Enforce MISRA C coding rules in `SQO-subset1`	X	X	X	X
Review all red checks	X	X	X	X
Review all gray checks	X	X	X	X
Review critical orange checks		X	X	X
Review all orange checks			X	X
Enforce MISRA C coding rules in `SQO-subset2`			X	X
Analyze dataflow			X	X

2 Depending on the quality level that you want to implement, choose your verification options. The options appear on the **Configuration** pane in the Polyspace user interface.

For instance, if you want to implement level QL1, under **Coding Rules**, select `SQO-subset1` for **Check MISRA C:2004**.

3 Depending on the quality level that you want to implement, plan your review process for the verification results. Your review process involves options in the Polyspace interface.

For instance, if you want to implement level QL1, on the **Results Summary** pane, filter only red and gray checks.

Define Specific Requirements for Verification

This example shows how to define specific requirements before you begin a Polyspace Code Prover verification, and then implement them in your verification process.

Specify Code Constructs

1 Prepare a list of constructs that you want to retain in your code or remove from it.

2 On the **Configuration** pane, specify the verification options corresponding to your requirements.

For instance, you can have the following requirements and choose the corresponding options.

Requirement	Option
Detect overflows only on signed integer computations.	Under **Check Behavior**, for **Detect overflows**, select `signed`.
Allow a pointer to one structure field to point to another field of the same structure.	Under **Check Behavior**, select **Enable pointer arithmetic across fields**.
Do not allow global variables to be initialized by default.	Under **Inputs & Stubbing**, select **Ignore default initialization of global variables**.

Specify Coding Rules

1 Prepare a list of coding rules for your code.

2 On the **Configuration** pane, under the **Coding Rules** node, specify your coding rules. For more information, see “Set Up Coding Rules Checking”.

Specify Results to Review

1 Prepare a list of files or list of checks that you want to review.

2 After you run your verification, apply appropriate filters to focus your review on those files or checks. For more information, see “Filter and Group Results”.

Setting Up Project in User Interface

Create Project Automatically

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

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

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

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

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

4 Click ▷ Run.

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

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

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

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

Note:

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.
- By default, Polyspace assigns the latest dialect for your compiler. If you have compilation errors in your project, check the dialect. If it does not apply to you, change it to a more appropriate one.
  For an example, see "Compilation Error After Creating Project from Visual Studio Build".

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

Related Examples

- “Trace Visual Studio Build”
- “Create Project Manually”
- “Update Project”

More About

- “Requirements for Project Creation from Build Systems”
- “Your Compiler Is Not Supported”

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet the following requirements:

- Your compiler must be called locally.

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

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

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

  If your compiler configuration is not available to Polyspace:

  - Write a compiler configuration file for your compiler in a specific format. For more information, see “Your Compiler Is Not Supported”.
  - Contact MathWorks Technical Support. For more information, see “Contact Technical Support”.
- In Linux®, only UNIX® shell (sh) commands must be used. If your build uses advanced commands such as commands supported only by bash, tcsh or zsh, Polyspace cannot trace your build.

  In Windows®, only DOS commands must be used. If your build uses advanced commands such as commands supported only by Powershell, Polyspace cannot trace your build.

- Your build command must not use aliases.

  The `alias` command is used in Linux to create an alternate name for commands. If your build command uses those alternate names, Polyspace cannot recognize them.
- Your build command must be executable completely on the current machine and must not require privileges of another user.

  If your build uses `sudo` to change user privileges or `ssh` to remotely login to another machine, Polyspace cannot trace your build.
- If you use Cygwin™ to build your source code, Polyspace cannot trace your build. You can either use MinGW to build your source and have the software trace your build, or do the following:

  1. Build your source code using the process that you usually follow. Copy the command lines that executed during the build to a file.

     For instance, on `make` systems, use the flag `-B` or `-W` *`makefileName`* to build your entire source and `-n` to view the commands. For more information, see make options.
  2. Save the file as a Windows batch file. A batch file is a file that can contain one or more commands. It has a `.bat` extension. For more information, see batch files.
  3. Run the batch file to make sure your build commands work.

     For example, if your batch file is called `myBuild.bat`, at a DOS command prompt, enter:

     ```
     cmd.exe /C myBuild.bat
     ```
  4. Create a project from the batch file.

     If you ran the command in the previous step, at a DOS command prompt, enter:

     ```
     polyspace-configure cmd.exe /C myBuild.bat
     ```
- If your build command uses redirection with the `>` or `|` character, the redirection occurs after Polyspace traces the command. Therefore, Polyspace does not handle the redirection.

  For example, if your command occurs as

  ```
  command1 | command2
  ```
  And you enter

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

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

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

See Also

polyspaceConfigure

Related Examples

- “Create Project Automatically”
- “Create Project Automatically at Command Line”
- “Create Project Automatically from MATLAB Command Line”

Your Compiler Is Not Supported

For automatic project creation from your build system, your compiler configuration must be available to Polyspace. For information on supported compilers, see “Requirements for Project Creation from Build Systems”. If your compiler is not supported, you can write your own compiler configuration file to enable support.

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

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

   To edit the file, save it outside the installation folder. After you have finished editing, you must copy the file back to `matlabroot\polyspace\configure\compiler_configuration\`.
3. Edit the contents of the file to represent your compiler. Replace the entries between the XML elements with appropriate content.

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

XML Element	Content Description	Content Example for GNU C Compiler

`<compiler_names><name> ...`
`</name><compiler_names>`
	Name of the compiler executable. This executable transforms your `.c` files into object files. You can add several binary names, each in a separate `<name>...</name>` element. `polyspaceConfigure` checks	• `gcc`
		• `gpp`

XML Element	Content Description	Content Example for GNU C Compiler
	for each of the provided names and uses the compiler name for which it finds a match. You must not specify the linker binary inside the `<name>...</name>` elements.	

`<include_options><opt> ...`
`</opt></include_options>`
	The option that you use with your compiler to specify include folders.	`-I`

`<system_include_options>`
`<opt> ... </opt>`
`</system_include_options>`
	The option that you use with your compiler to specify system headers.	`-isystem`

`<preinclude_options><opt> ...`
`</opt></preinclude_options>`
	The option that you use with your compiler to force inclusion of a file in the compiled object.	`-include`

`<define_options><opt> ...`
`</opt></define_options>`
	The option that you use with your compiler to predefine a macro.	`-D`

`<undefine_options><opt> ...`
`</opt></undefine_options>`
	The option that you use with your compiler to undo any previous definition of a macro.	`-U`

`<semantic_options><opt> ...`
`</opt></semantic_options>`
	The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.	• `-ansi`
		• `-std =C90`
		• `-std =c++11`
		• `-fun signed -char`

XML Element	Content Description	Content Example for GNU C Compiler
`<dialect> ... </dialect>`	The options that specify the Polyspace dialect used by your compiler. For the complete list of dialects, on the **Configuration** pane, select **Target & Compiler**.	`gnu4.7`

`<preprocess_options_list>`
`<opt> ... </opt>`
`</preprocess_options_list>`
	The options that specify how your compiler generates a preprocessed file.	`-E`

`<src_extensions><ext> ...`
`</ext></src_extensions>`
	The file extensions for source files.	• `c`
		• `cpp`
		• `c++`

`<obj_extensions><ext> ...`
`</ext></obj_extensions>`
	The file extensions for object files.	

`<precompiled_header_extensions> ...`
`</precompiled_header_extensions>`
	The file extensions for precompiled headers (if available).	

`<polyspace_c_extra_options_list>`
`<opt> ... </opt>`
`</polyspace_c_extra_options_list>`
	Additional options that will be added to your project configuration	To avoid compilation errors due to non-ANSI® extension keywords, enter `-D` *`keyword`*. For more information, see “Preprocessor definitions (C/C++)”.

XML Element	Content Description	Content Example for GNU C Compiler

`<polyspace_cpp_extra_options_list>`
`<opt> ... </opt>`
`</polyspace_cpp_extra_options_list>`
	Additional options that will be added to your C++ project configuration	To avoid compilation errors due to non-ANSI extension keywords, enter `-D` *`keyword`*. For more information, see “Preprocessor definitions (C/C++)”.

4 After saving the edited XML file to *`matlabroot`*`\polyspace\configure\compiler_configuration\`, create a project automatically using your build command. For more information, see:

- “Create Project Automatically”
- “Create Project Automatically at Command Line”
- “Create Project Automatically from MATLAB Command Line”

Create Project Manually

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

Otherwise, see “Create Project Automatically”. If automatic project creation is not supported for your compiler, see the suggestions in “Requirements for Project Creation from Build Systems”. If the suggestions do not work, create a project manually.

To create a project manually, you must know:

- Location of your source files
- Location of your include files

In this section...

Create Project

This example shows how to create a new project.

1 Select **File** > **New Project**.

2 In the Project – Properties dialog box, enter the following information:

   - **Project name**
   - **Location**: Folder where you will store the project file with extension `.psprj`. You can use this file to open an existing project.

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

3 Add source files and include folders to your project.

   - Navigate to the location where you stored your source files. Select the source files for your project. Click **Add Source Files**.

- The software automatically adds the standard include files to your project. To use custom include files, navigate to the *folder* containing your include files. Click **Add Include Folders**.

4. Click **Finish**.

   The new project opens in the **Project Browser** pane. Your source files are automatically copied to the first module in the project.

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

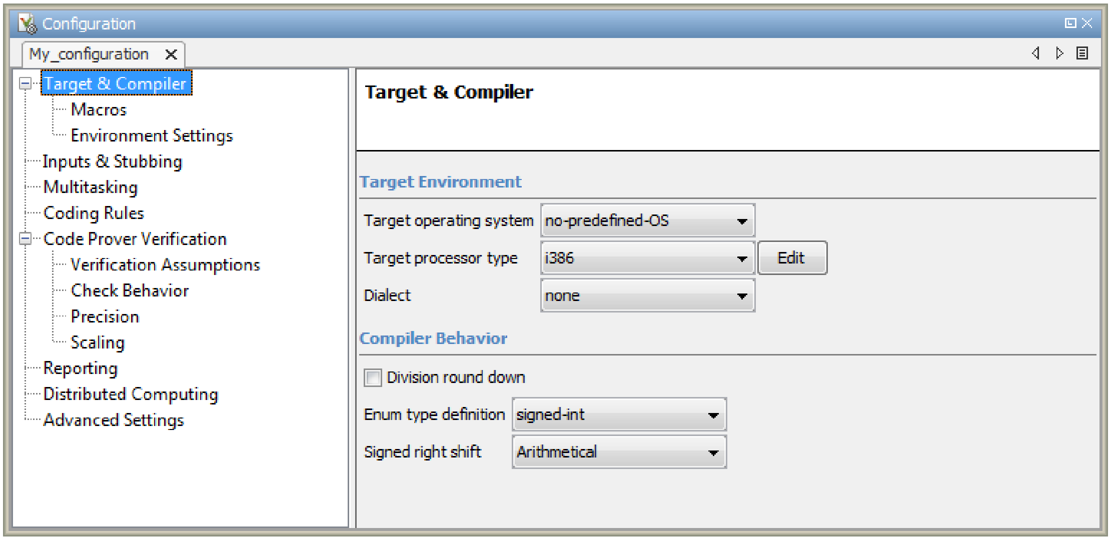
   To close the project at any time, in the **Project Browser**, right-click the project node and select **Close**.

Specify Analysis Options

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

Each project consists of one or more modules. Before running verification on a module, you can change the analysis options. Each module has a **Configuration** that consists of the default analysis options. To change the analysis options:

1. On the **Project Browser**, below the **Configuration** node of the module, select the configuration.
2. Change the options on the **Configuration** pane.



Target & Compiler

Target Environment

Target operating system: no-predefined-OS

Target processor type: i386

Dialect: none

Compiler Behavior

□ Division round down

Enum type definition: signed-int

Signed right shift: Arithmetical

For instance:

- To specify the target processor, select the **Target & Compiler** node. Select your processor from the **Target processor type** drop-down list.
- To specify verification precision, select the **Precision** node. Select a number for **Precision level**.

You can also create another configuration in your module. For more information, see “Create Configurations in Module”.

For more information on the options, see:

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

Related Examples

- “Create Project Automatically”
- “Create Project Using Template”
- “Update Project”

Create Project Using Template

A **Project Template** is a predefined set of analysis options for a specific compilation environment. When creating a new project, you can do one of the following:

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

  Polyspace software provides predefined templates for common compilers such as `IAR`, `Kiel`, and `VxWorks` `Aonix`, `Rational`, and `Greenhills`. For additional templates, see Polyspace Compiler Templates .
- Set analysis options manually. You can then save your options as a template and reuse them later.

In this section...

Use Predefined Template

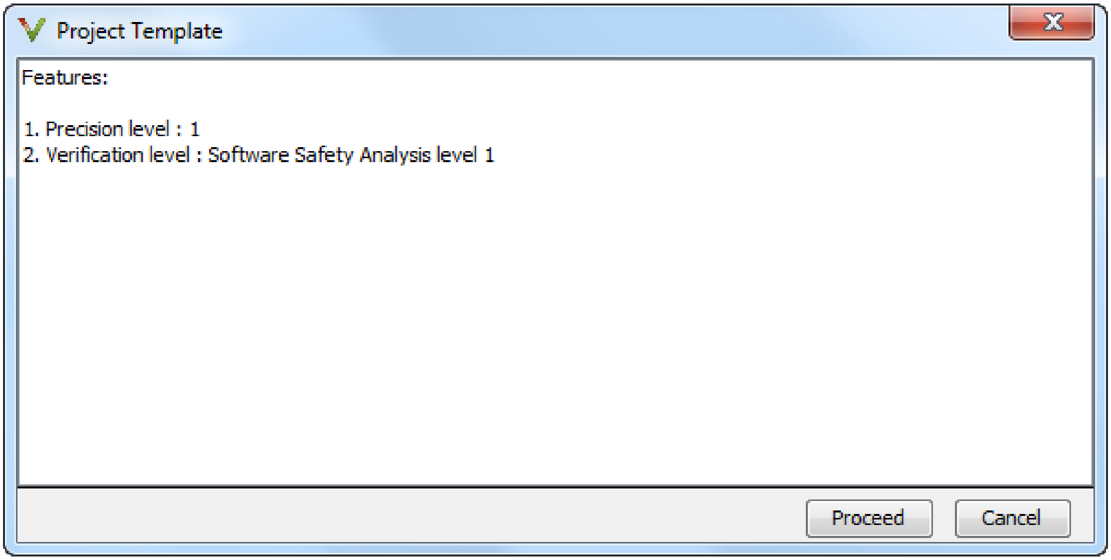
1. Select **File** > **New Project**.
2. On the Project – Properties dialog box, after specifying the project name, location and language, under **Project configuration**, select **Use template**.
3. On the next screen, select the template appropriate for your compiler. For further details on a template,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

   If your compiler does not appear in the list of predefined templates, select **Baseline**.
4. On the next screen, add your source files and include folders. For more information, see "Create Project Manually".

Create Your Own Template

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

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

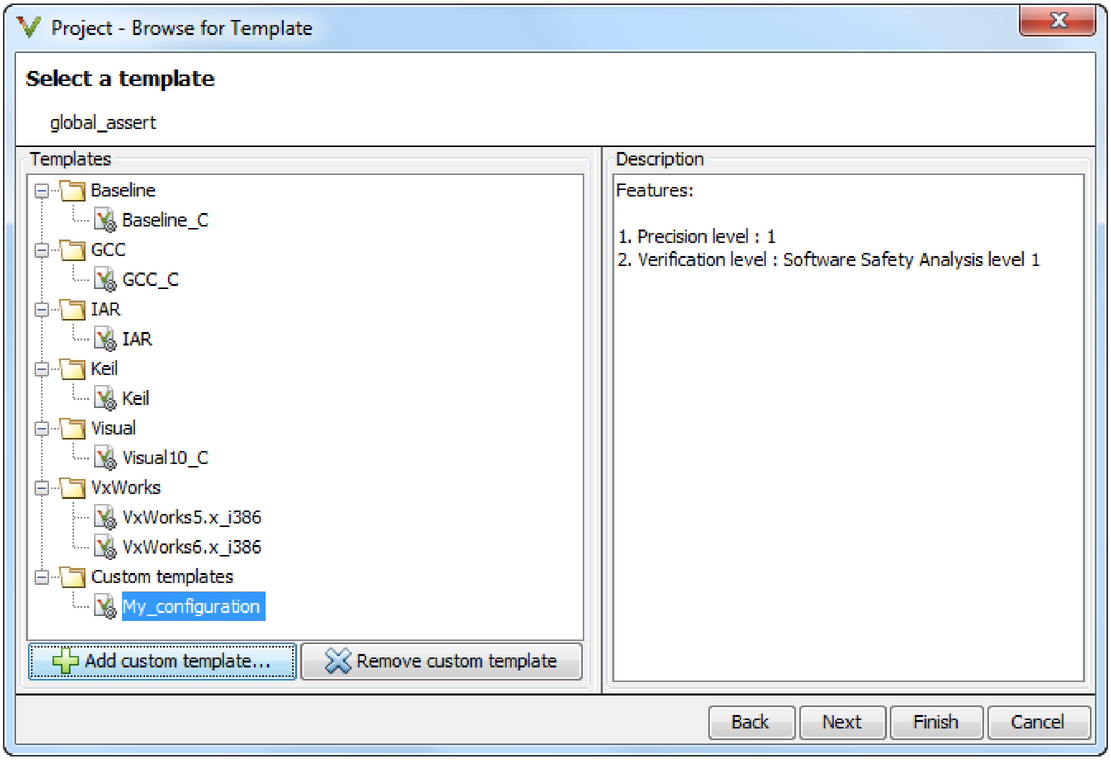


Features:

1. Precision level : 1
2. Verification level : Software Safety Analysis level 1

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

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



Project - Browse for Template

Select a template

global_assert

Templates

- Baseline
  - Baseline_C
- GCC
  - GCC_C
- IAR
  - IAR
- Keil
  - Keil
- Visual
  - Visual10_C
- VxWorks
  - VxWorks5.x_i386
  - VxWorks6.x_i386
- Custom templates
  - My_configuration

Description

Features:

1. Precision level : 1
2. Verification level : Software Safety Analysis level 1

Related Examples

- “Create Project Automatically”
- “Create Project Manually”
- “Update Project”

Update Project

If you created your project automatically from your build system, to update the project later:

1. On the **Project Browser** pane, right-click the project and select **Update Project**.
2. Enter the same information you did when creating the original project. For more information, see “Create Project Automatically”.

You can also manually add source files and include folders to an existing project, or change the analysis options.

In this section...

Add Sources and Includes

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

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

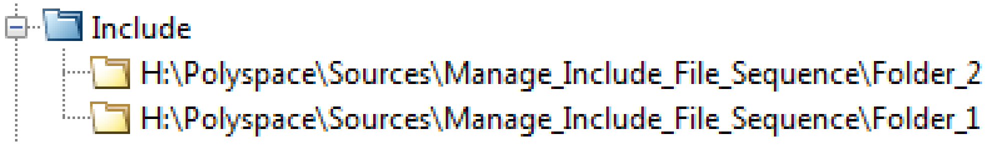
   a. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files.
   b. Right-click your selection.
   c. Select **Copy to** > **Module_*n***. *n* is the module number.

Manage Include File Sequence

You can change the order of include folders to manage the sequence in which include files are compiled.

When multiple include files by the same name exist in different folders, you might want to change the order of include folders instead of reorganizing the contents of your folders. For a particular include file name, the software includes the file in the first include folder under ***`Project_Name`*** > **Include**.

In the following figure, `Folder_1` and `Folder_2` contain the same include file `include.h`. If your source code includes this header file, during compilation, `Folder_2/include.h` is included in preference to `Folder_1/include.h`.



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

To change the order of include folders:

1. In the **Project Browser**, expand the **Include** folder.
2. Select the include folder that you want to move.
3. To move the folder, click either ⇧ or ⇩ on the **Project Browser** toolbar.

Change Analysis Options

For later verifications, you might have to change your analysis options. For instance:

- To avoid compilation errors in Polyspace for constructs that are allowed by your compiler, specify your target and compiler options.

  For more information, see:

  - C code: “Target & Compiler”
  - C++ code: “Target & Compiler”
- If you provide partially developed code, you can specify external constraints to stand in for the remaining code. Towards the end of your development cycle, as you provide more complete code for verification, you can remove some of these constraints.

  For more information, see:

  - C code: “Inputs & Stubbing”
  - C++ code: “Inputs & Stubbing”
- If your code is intended for multitasking, you can specify your entry points and protection mechanisms.

  For more information, see “Multitasking”.
- To allow Polyspace to prove more operations and therefore produce fewer non-critical orange checks, you can specify appropriate options.

  For more information, see “Reduce Orange Checks”.

For more information, see “Specify Analysis Options”.

Related Examples

- “Create Project Automatically”
- “Create Project Manually”

Modularize Project Manually

You can create multiple modules in a Polyspace Code Prover project. In each module, you can copy all or some of your source files.

On the **Project Browser** pane, each module contains the following nodes.

Node	Content
Source	All or some of the source files in the project. When you run verification on the module, the software verifies these source files.
Configuration	One or more configurations. Each configuration consists of a set of analysis options.
Result	One or more results.

In your file system, each module corresponds to a subfolder of your project folder.

Note: If you add your source files when creating a new project, they are automatically copied to the first module, **Module_1**. If you add them later, you must copy them manually to a module.

In this section...

Create New Module

Suppose you have one module, **Module_1**, in your project.

1 Do one of the following on the **Project Browser** pane:

- Select your project. Click the button on the **Project Browser** toolbar.
- Right-click your project or the existing module. Select **Create New Module**.

You see a new module, **Module_2**, in your project.

2 In your project, below the **Source** node, right-click the files that you want to add to the module. From the context menu, select **Copy to** > **Module_2**.

The software displays these files below the **Source** node of `Module_2`.

Create Configurations in Module

By default, when you create a new module, it contains a configuration with the default analysis options. To run verification on the module with different options, do one of the following:

- Change the analysis options in this configuration.
- Create a new configuration and change the options in the new configuration. You can retain the default analysis options in the original configuration.

Tip To copy a configuration to another module, right-click the configuration. Select **Copy Configuration to** > ***`Module_name`***.

To create a new configuration in your module:

1 Right-click the **Configuration** folder in the module. From the context menu, select **Create New Configuration**.

- On the **Project Browser** pane, the software displays a new configuration *`project_name`*`_1`. To rename the configuration, double-click it.
- On the **Configuration** pane, the new configuration appears as an additional tab.

2 On the **Configuration** pane, specify the analysis options for the new configuration.

3 To use this new configuration, double-click it.

When you run a new verification on the module, it uses the analysis options in this configuration.

4 To see the configuration you used for a certain result, right-click the result on the **Project Browser**. Select **Open Configuration**.

You can see a read-only form of the configuration.

Note: If you are viewing the results and do not have the corresponding project open on your **Project Browser**, to see the configuration you used, select **Window > Show/Hide View > Configuration**.

Related Examples

- "Modularize Project Automatically"

Modularize Project Automatically

If the source code in your project represents a single application, you might want to analyze all of the code together. However, if the application is extremely large, the verification can take a long time, for example, days.

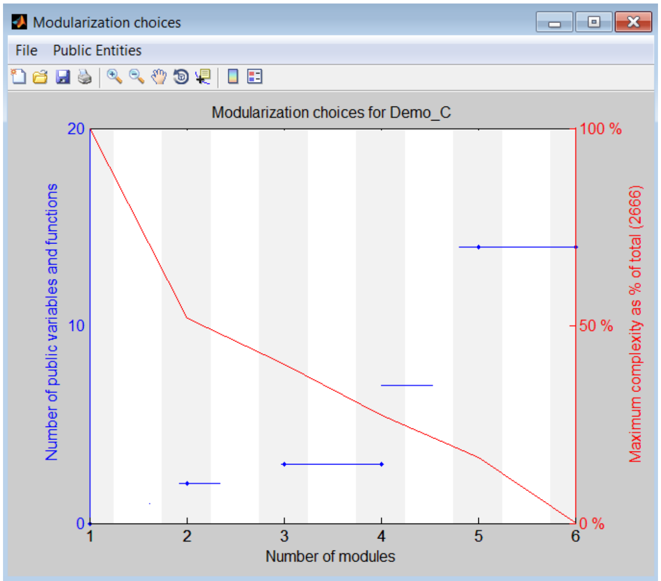
For a large application, Polyspace allows you to:

- Partition the application into modules that individually require less time to verify.
- Specify the number of modules in a trade-off between verification speed and precision.

You can carry out faster analysis with a larger number of small modules. During partitioning, the software automatically minimizes cross-module references. However, with more modules, greater cross-module referencing is required during verification, which results in a loss of precision.

To partition your application into modules:

1. Run an initial verification, which performs a limited analysis but processes all the files of your application. For example, run a verification with the following **Precision** pane settings:
   - **Precision level** — `0`
   - **Verification level** — `Software Safety Analysis level 0`
2. In the **Project Browser** view, select the results folder.
3. Select **Tools** > **Run Modularize**. The software analyzes your application code and displays two plots in a new Modularization choices window.


Modularization choices
File
Public Entities
Modularization choices for Demo_C
Number of public variables and functions
20
10
0
100 %
50 %
0 %
Maximum complexity as % of total (2666)
1
2
3
4
5
6
Number of modules

The plots show the following information:

- Red — Maximum complexity of a module versus number of modules, which is expressed as a percentage of the total complexity of the application.
- Blue — Number of public variables and functions when modules are limited by a given complexity.

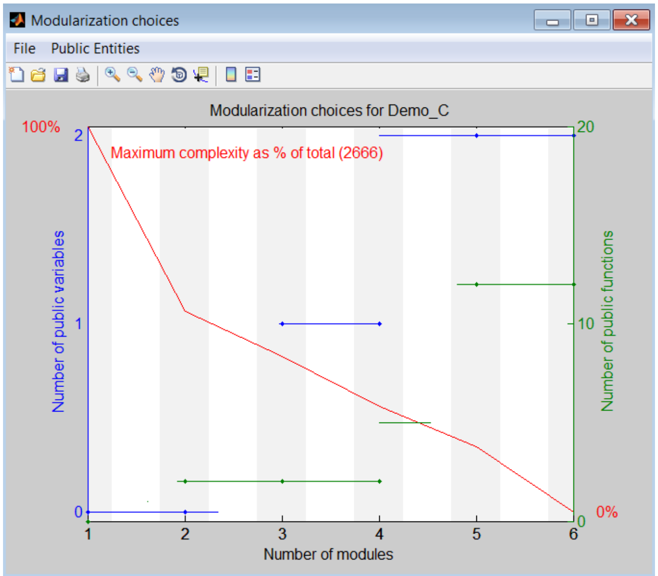
4 From the plots, identify the number of modules into which your application must be partitioned. In this example, a suitable number is 2 or 4.

The number of partitioned modules that you choose involves a trade-off between the following:

- Time — The smaller the maximum complexity, the shorter the time required for verification. This time saving is even greater if the different modules are verified in parallel.
- Precision — The smaller the number of public variables and functions, the greater the precision of the verification.

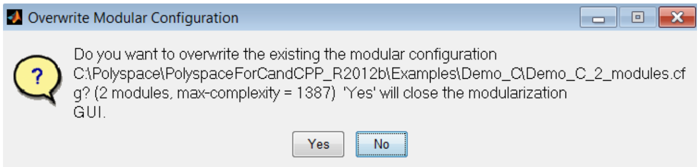
Select a number just after a big drop in maximum complexity and before a big increase in the number of public functions and variables. The precision of a modular verification can be very sensitive to the number of public variables. If the series of horizontal blue lines ascends so gradually that there is no clear number choice, then:

a On the toolbar, select **Public Entities > Separate functions and variables**. The software displays the number of public variables and functions separately.


Modularization choices
File
Public Entities
Modularization choices for Demo_C
100%
2
20
Maximum complexity as % of total (2666)
Number of public variables
1
10
Number of public functions
0
0%
0
1
2
3
4
5
6
Number of modules

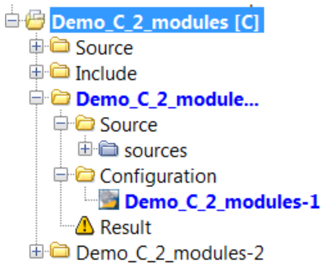
b Select a point just before a big jump in the number of public variables. In this example, you must click the gray region associated with 2.

5 Click the vertical gray region associated with the number of modules that you choose, for example, 2. A dialog box opens.



Do you want to overwrite the existing the modular configuration C:\Polyspace\PolyspaceForCandCPP_R2012b\Examples\Demo_C\Demo_C_2_modules.cfg? (2 modules, max-complexity = 1387) 'Yes' will close the modularization GUI.

6 Click **Yes**. The software generates a new project with two modules containing the partitioned code.



You can now verify each module separately.

Related Examples

- “Modularize Project Manually”

Setting Up Polyspace User Interface

Specify External Text Editor

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

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

2 On the Polyspace Preferences dialog box, select the **Editors** tab.

3 From the **Text editor** drop-down list, select **External**.

4 In the **Text editor** field, specify the path to your text editor. For example:

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

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

   Polyspace has already specified the command-line arguments for the following editors:

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

   If you are using one of these editors, select it from the **Arguments** drop-down list. If you are using another text editor, select `Custom` from the drop-down list, and enter the command-line options in the field provided.

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

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

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

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

Change Default Font Size

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

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

2 On the **Miscellaneous** tab:

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

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

3 Click **OK**.

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

Customize Results Folder Location and Name

By default, the software saves results in `Module_#` subfolders within the project folder. However, through the Polyspace Preferences dialog box, you can define a parent folder for your results:

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

2 On the **Project and Results Folder** tab, select the **Create new result folder** check box.

3 In the **Parent results folder location** field, specify the location that you want.

Note: If you do not specify a parent results folder, the software uses the active module folder as the parent folder.

4 If you want your results to be stored in a subfolder instead of directly in the parent folder, select **Add a subfolder using the project name**. This subfolder takes the name of the project.

5 If required, specify additional formatting options for the folder name . The options allow you to incorporate the following information into the name of the results folder:

- **Result folder prefix** — A string that you define. Default is `Result`.
- **Project variable** — Project, module, and configuration.
- **Date format** — Date of analysis
- **Time format** — Time of analysis
- **Counter** — Count value that automatically increments by one for each new results folder

The software now creates a new results folders with the file name *`ResultFolderPrefix_ProjectVariable_DateFormat_TimeFormat_Counter`*.

Storage of Polyspace Preferences

The software stores the settings that you specify through the Polyspace Preferences dialog box in the following file:

- Windows: `$Drive\Users\$User\AppData\Roaming\MathWorks \MATLAB \$Release\Polyspace\polyspace.prf`
- Linux: `/home/$User/.matlab/$Release/Polyspace/polyspace.prf`

Here, *`$Drive`* is the drive where the operating system files are located such as `C:`, *`$User`* is the username and *`$Release`* is the release number.

The following file stores the location of all installed Polyspace products across various releases:

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

Emulating Your Runtime Environment

Set Up a Target

In this section...

Target & Compiler Overview

Many applications are designed to run on specific target CPUs and operating systems. The type of CPU determines many data characteristics, such as data sizes and addressing. These factors can determine whether errors occur, for example, overflows.

Since some run-time errors are dependent on the target CPU and operating system, you must specify the type of CPU and operating system used in the target environment before running a verification.

Specify Target and Compiler

Before verification, you can specify the target environment and compiler behavior for your application.

For example, to specify the target environment for your application:

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

2 For **Target operating system**, select the operating system on which your application is designed to run.

3 For **Target processor type**, select the processor on which your application is designed to run.

   For detailed specification of each predefined target processor, see “Target processor type (C)” or “Target processor type (C++)”.

Modify Predefined Target Processor Attributes

If your processor is not listed under **Target processor type**, you can select a similar processor and modify its characteristics to match your processor. For the settings that you can modify for each target, see the values listed in [ ] on “Target processor type (C)” or “Target processor type (C++)”.

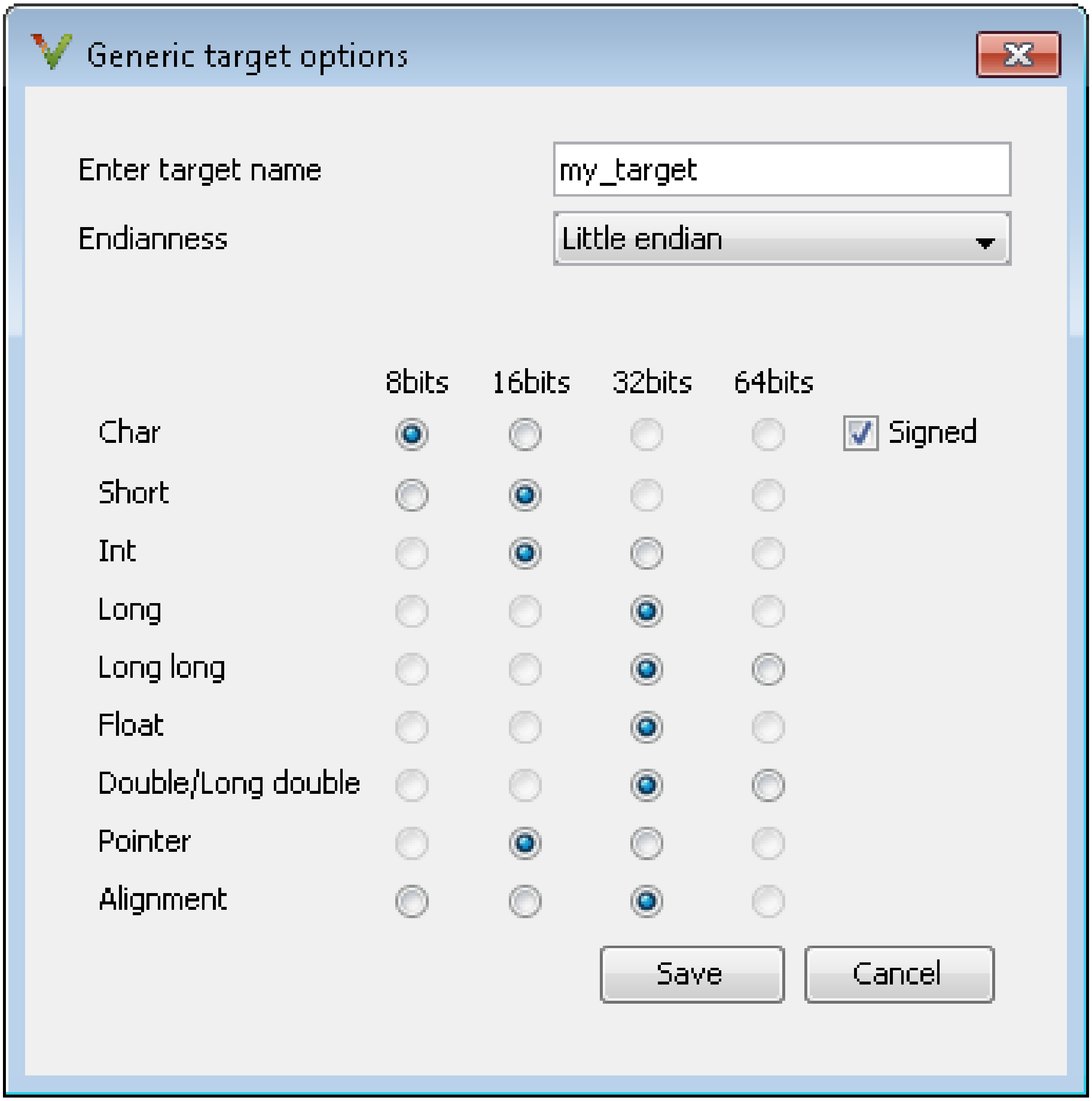
1. On the **Configuration** pane, select **Target & Compiler**.
2. For **Target processor type**, select the target processor that you want to use.
3. To the right of the **Target processor type** field, click **Edit**.
4. Modify the attributes as required.

Define Generic Target Processors

If your application is designed for a custom target processor, you can specify its attributes.

1. On the **Configuration** pane, select **Target & Compiler**.
2. For **Target processor type**, select `mcpu... (Advanced)`.

   The Generic target options dialog box opens.



Enter target name: my_target

Endianness: Little endian

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

3 Enter a target name, for example, `my_target`.

4 Specify the parameters for your target, such as the size of basic types, and alignment with arrays and structures.

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

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

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

ST9 (GNU C compiler : gcc9 for ST9)

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

Hitachi H8/300, H8/300L

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

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

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

Hitachi H8/300H, H8S, H8C, H8/Tiny	char	short	int	long	long long	float	double	long double	ptr	char is	endian
alignmen	8	16	32/ 16	32/16	32/16	32/16	32/16	32/16	32/16	N/A	N/A

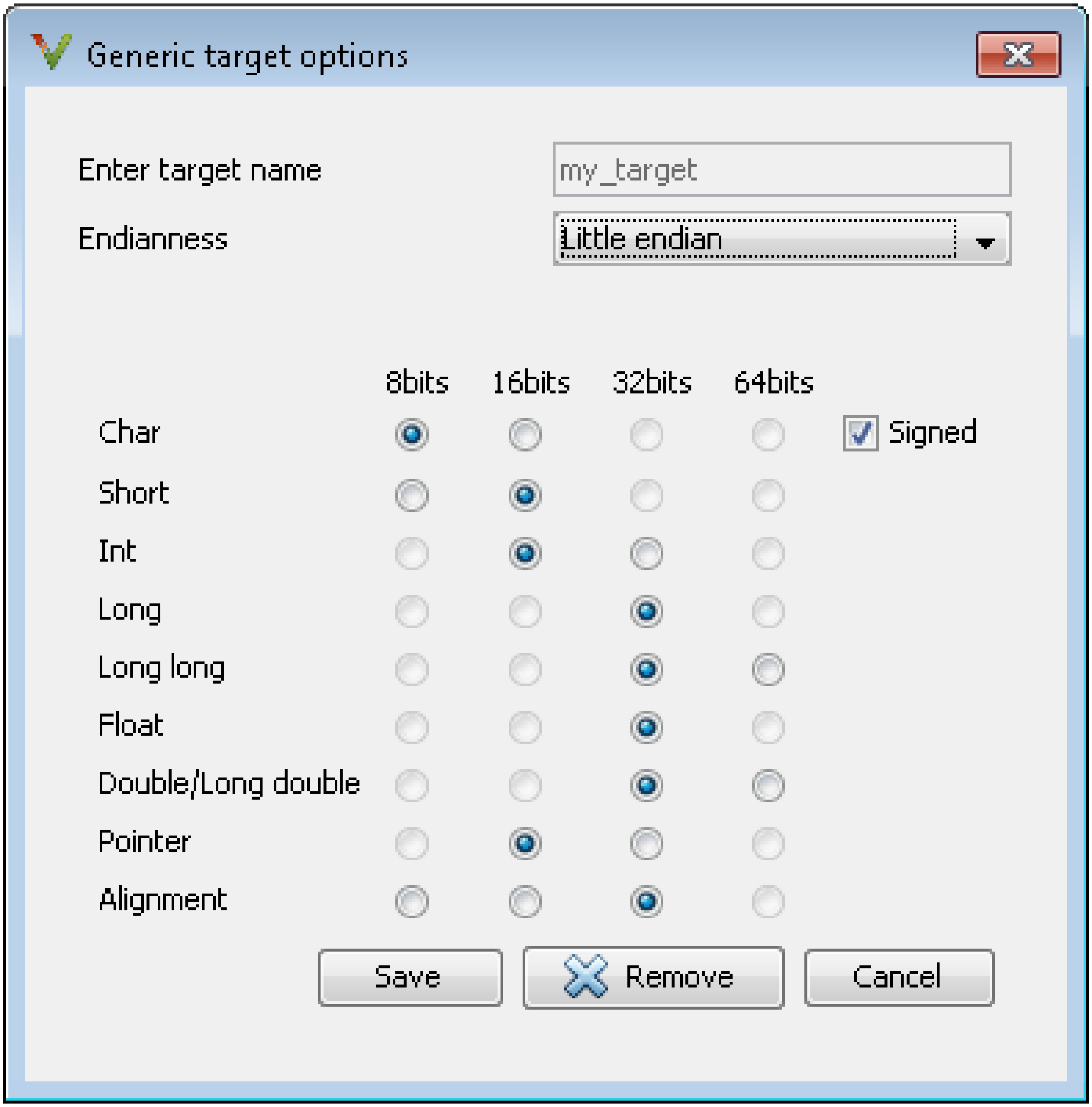
View or Modify Existing Generic Targets

To view or modify generic targets that you previously created:

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

2 For **Target processor type**, select your target, for example, `MyTarget`.

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



Generic target options

Enter target name: my_target

Endianness: Little endian

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

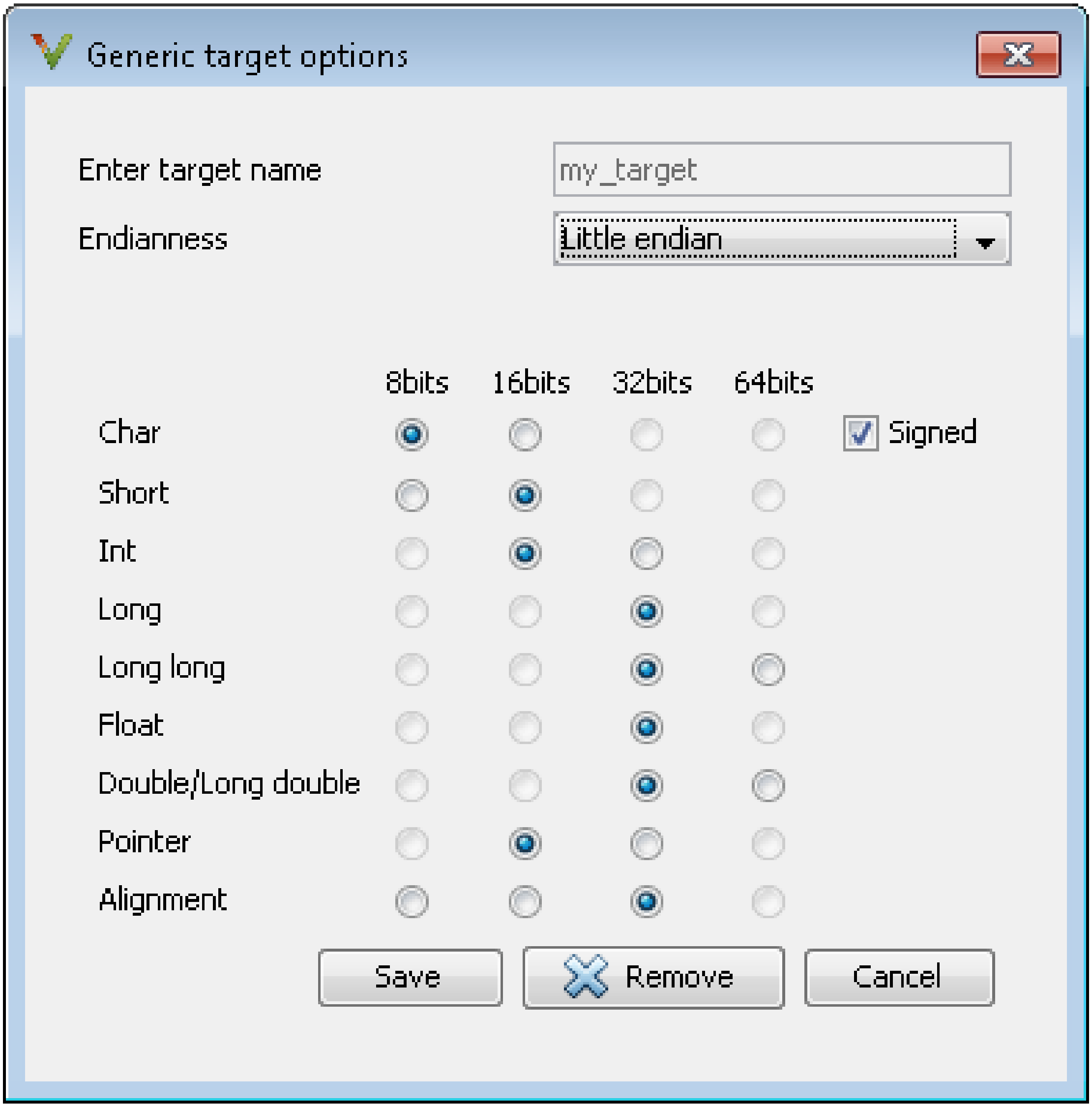
4 If required, specify new attributes for your target. Then click **Save**.

5 Otherwise, click **Cancel**.

Delete Generic Target

To delete a generic target:

1. On the **Configuration** pane, select **Target & Compiler**.
2. For **Target processor type**, select the target that you want to remove, for example, `my_target`.



Generic target options

Enter target name: my_target

Endianness: Little endian

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

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

Compile Operating System Dependent Code

- “Predefined Compilation Flags for C Code” on page 5-12
- “Predefined Compilation Flags for C++ Code” on page 5-14
- “My Target Application Runs on Linux” on page 5-18
- “My Target Application Runs on Solaris” on page 5-18
- “My Target Application Runs on Vxworks” on page 5-19
- “My Target Application Does Not Run on Linux, VxWorks, or Solaris” on page 5-19

Predefined Compilation Flags for C Code

If you choose a value for **Target operation system**, before verification, Polyspace automatically:

- Defines certain compilation flags.
- Includes certain header files.

Target operating system	Compilation flags	—include file and content
no-predefined-OS	-D__STDC__	
Visual	-D__STDC__	-include *<product_dir>*/cinclude/pst-visual.h
VxWorks	-D__STDC__ -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__GNUC__=2 -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4__	-include *<product_dir>*/cinclude/pst-vxworks.h

Target operating system	Compilation flags	—include file and content
	-D__SVR4	
Linux	-D__STDC__ -D__GNUC__=2 -D__GNUC_MINOR__=6 -D__GNUC__=2 -D__GNUC_MINOR__=6 -D__ELF__ -Dunix -D__unix -D__unix__ -Dlinux -D__linux -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686 -D__i686__ -Dpentiumpro -D__pentiumpro -D__pentiumpro__	*<product_dir>*/cinclude/pst-linux.h

Target operating system	Compilation flags	—include file and content
Solaris	-D__STDC__ -D__GNUC__=2 -D__GNUC_MINOR__=8 -D__GNUC__=2 -D__GNUC_MINOR__=8 -D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4__ -D__SVR4	No -include file mentioned

Note: The use of the `-OS-target` option is equivalent to the following approaches:

- Setting the same `-D` flags manually
- Using the `-include` option on a copied and modified pst-*`OS-target`*.h file

Predefined Compilation Flags for C++ Code

The following table shown for each `–OS-target`, the list of compilation flags defined by default, including pre-include header file (see also `–include`):

Target operating system	Compilation flags	-include file	Minimum set of options
Linux	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int	<product_dir>/cinclude/pst-linux.h	polyspace-[desktop-]cpp -OS-target Linux \

Target operating system	Compilation flags	-include file	Minimum set of options
	-D__inline__=inline -D__signed__=signed - D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_ SPECIALIZATION -D__GNU_SOURCE -D__STDC__ -D__ELF__ -Dunix -D__unix -D__unix__ -Dlinux -D__linux -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686 -D__i686__ -Dpentiumpro -D__pentiumpro -D__pentiumpro__		-I <polyspace_install>/ include/ include-linux \ -I <product_dir>/include/ include-linux/next Where the Polyspace product has been installed in the folder <polyspace_install>

Target operating system	Compilation flags	-include file	Minimum set of options
VxWorks	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__STDC -D__GNU_SOURCE -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4	<product_dir>/cinclude/pstvxworks. h	polyspace-[desktop-]cpp \ -OS-target vxworks \ -I /your_path_to/Vxworks_include_folders

Target operating system	Compilation flags	-include file	Minimum set of options
Visual	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION	<product_dir>/cinclude/pstvisual. h	
Solaris	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNU_SOURCE -D__STDC -D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4		If Polyspace runs on a Linux machine: polyspace-[desktop-]cpp \ -OS-target Solaris \ -I /your_path_to_solaris_include If Polyspace runs on a Solaris machine: polyspace-code-prover-nodesktop -lang cpp\ -OS-target Solaris \ -I /usr/include

Target operating system	Compilation flags	-include file	Minimum set of options
no-predefined-OS	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION		polyspace-[desktop-]cpp \ -OS-target no-predefined-OS \ -I /your_path_to/MyTarget_include_folders

Note: This list of compiler flags is written in every log file.

My Target Application Runs on Linux

The minimum set of options is as follows:

```
polyspace-code-prover-nodesktop \
 -OS-target Linux \
 -I MATLAB_Install/polyspace/verifier/cxx/include/include-libc \

 ...
```

If your target application runs on Linux but you are starting your verification from Windows, the minimum set of options is as follows:

```
polyspace-code-prover-nodesktop \
 -OS-target Linux \
 -I MATLAB_Install\polyspace\verifier\cxx\include\include-libc \

 ...
```

MATLAB_Install is your MATLAB® installation folder.

My Target Application Runs on Solaris

If Polyspace software runs on a Linux machine:

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

My Target Application Runs on Vxworks

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

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

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

If Polyspace software does not run on a Linux machine:

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

Address Alignment

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

- Global `sizeof` and `offsetof` fields are optimum, that is, as short as possible.
- The alignment of addressable units is respected.
- Global alignment is respected.

Consider the example:

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

- Each field must be aligned; that is, the starting offset of a field must be a multiple of its own size[2]
- So in the example, `char a` begins at offset 0 and its size is 8 bits. `int b` cannot begin at 8 (the end of the previous field) because the starting offset must be a multiple of its own size (32 bits). Consequently, `int b` begins at offset=32. The size of the `struct foo` before global alignment is therefore 64 bits.

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

- The global alignment of a structure is the maximum of the individual alignments of each of its fields;
- In the example, `global_alignment = max (alignment char a, alignment int b) = max (8, 32) = 32`
- The size of a struct must be a multiple of its global alignment. In our case, b begins at 32 and is 32 long, and the size of the struct (64) is a multiple of the `global_alignment` (32), so `sizeof` is not adjusted.

Ignore or Replace Keywords Before Compilation

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

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

Content of myTpl.pl

```
#!/usr/bin/perl

##############################################################
# Post Processing template script
#
##############################################################
# Usage from GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: MATLAB_Install\sys\perl\win32\bin\perl.exe <pathtoscript>\
PostProcessingTemplate.pl
#
##############################################################

$version = 0.1;

$INFILE = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{

 # Remove far keyword
 s/far//;
 # Remove "@ 0xFE1" address constructs
 s/\@\s0x[A-F0-9]*//g;

 # Remove "@0xFE1" address constructs
 # s/\@0x[A-F0-9]*//g;

 # Remove "@ ((unsigned)&LATD*8)+2" type constructs
 s/\@\s\(\(unsigned\)\&[A-Z0-9]+\*8\)\+\d//g;

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

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

For reference, see a summary of Perl regular expressions.

Perl Regular Expressions

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

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

For more information, see `-post-preprocessing-command`.

Language Extensions

The software allows a verification to accept a subset of common C language constructs and extended keywords, as defined by the C99 standard or supported by many compilers.

By default, the following constructs are accepted:

- Designated initializers (labeling initialized elements)

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

The software ignores the following extended keywords:

- `near`
- `far`
- `restrict`
- `_attribute_(X)`
- `rom`

Verify Keil or IAR Dialects

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

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

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

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

```
int status,P0;

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

You can verify this type of code using the **Dialect** (`-dialect`) option . This option allows the software to support the Keil or IAR C language extensions even if some structures, keywords, and syntax are not ANSI standard. The following tables summarize what is supported when verifying code that is associated with the Keil or IAR dialects.

The following table summarizes the supported Keil C language extensions:

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

Type/Language	Description	Example	Restrictions

Type `bit`	• An expression to type bit gives values in range [0,1].
	• Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c ++ bool type.
		`bit x = 0, y = 1,`
		` z = 2;`
		`assert(x == 0);`
		`assert(y == 1);`
		`assert(z == 1);`
		`assert(sizeof(bit)`
		` == sizeof(int));`
			pointers to bits and arrays of bits are not allowed

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

Type/Language	Description	Example	Restrictions
	will be considered as volatile.		

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

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

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

Keywords removed during preprocessing	The software removes certain Keil keywords during preprocessing. If you use any of these keywords as a	`alien`, `bdata`, `far`, `idata`, `ebdata`, `huge`, `sdata`	

Type/Language	Description	Example	Restrictions
	variable name, you see a compilation error. To avoid the compilation error, do one of the following: • In the user interface, enter `__PST_KEIL_NO_KEYW` for **Preprocessor definitions**. • On the command-line, use the flag `-D __PST_KEIL_NO_KEYW`		

The following table summarize the IAR dialect:

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

Type/Language	Description	Example	Restrictions

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

Type/Language	Description	Example	Restrictions

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

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

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

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

Type/Language	Description	Example	Restrictions
	and then restores the previous interrupt state at function exit.		
Keywords removed during preprocessing	The software removes certain IAR keywords during preprocessing. If you use any of these keywords as a variable name, you see a compilation error. To avoid the compilation error, do one of the following: • In the user interface, enter `__PST_IAR_NO_KEYWO` for **Preprocessor definitions**. • On the command-line, use the flag `-D __PST_IAR_NO_KEYWO`	`saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `data`, `xdata`, `xhuge`, `interrupt`, `__interrupt` and `__intrinsic`	

Unnamed struct/ union	• Fields of unions/ structs with no tag and no name can be accessed without naming their parent struct.
	• On a conflict between a field of an anonymous struct with other identifiers:
	  • with a variable name, field name is hidden
		`union { int x; };`
		`union { int y; struct { int z; }; } @ 0xF0;`	

Type/Language	Description	Example	Restrictions
	• with a field of another anonymous struct at different scope, closer scope is chosen • with a field of another anonymous struct at same scope: an error "anonymous struct field name <name> conflict“ is displayed in the log file.		

`no_init` attribute	• a global variable declared with this attribute is handled like an external variable.
	• It is handled like a type qualifier.
		`no_init int x;`
		`no_init union`
		`{ int y; } @ 0xFE;`
			#pragma no_init has no effect

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

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

For example:

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

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

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

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

Gather Compilation Options Efficiently

The code is often tuned for the target (see “Verify Keil or IAR Dialects” on page 5-23). Instead of applying minor changes to the code, create a single `polyspace.h` file that contains all target specific functions and options. The `-include` option can then be used to force the inclusion of the `polyspace.h` file in all source files under verification.

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

Direct benefits:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- The position of the error will be identified more precisely.
- There will be no need to modify original source files.

Indirect benefits:

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

Example

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

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

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

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

```
// A prototype may be positioned here to aid in the solution of
// a link phase conflict between
// declaration and definition. This will allow detection of the
// same error at compilation time instead of at link time.
// Leads to:
// - earlier detection
// - precise localisation of conflict at compilation time
void f(int);

// The same also applies to variables.
extern int x;

// Standard library stubs can be avoided,
// and OS standard prototypes redefined.

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

Supported C++ 2011 Standards

Standard	Description	Supported
C++2011-N2118	Rvalue references	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N2756	Non-static data member initializers	Yes
C++2011-N2242	Variadic templates	Yes
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2672	Initializer lists	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N3276	decltype and call expressions	Yes

Standard	Description	Supported
C++2011-N1757	Right angle brackets	Yes
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N2431	Null pointer constant	Yes
C++2011-N2347	Strongly-typed enums	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2341	Alignment support	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2249	New character types	Yes
C++2011-N2442	Unicode string literals	Yes

Standard	Description	Supported
C++2011-N2442	Raw string literals	Yes
C++2011-N2170	Universal character name literals	No
C++2011-N2765	User-defined literals	Yes
C++2011-N2342	Standard Layout Types	Not applicable[1]
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N1791	Extended friend declarations	No
C++2011-N2253	Extending sizeof	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes
C++2011-N2239	Concurrency - Sequence points	Not applicable[1]
C++2011-N2427	Concurrency - Atomic operations	No

Standard	Description	Supported
C++2011-N2748	Concurrency - Strong Compare and Exchange	No
C++2011-N2752	Concurrency - Bidirectional Fences	No
C++2011-N2429	Concurrency - Memory model	Not applicable[1]
C++2011-N2664	Concurrency - Data-dependency ordering: atomics and memory model	No
C++2011-N2179	Concurrency - Propagating exceptions	No
C++2011-N2440	Concurrency - Abandoning a process and at_quick_exit	Yes
C++2011-N2547	Concurrency - Allow atomics use in signal handlers	No
C++2011-N2659	Concurrency - Thread-local storage	No
C++2011-N2660	Concurrency - Dynamic initialization and destruction with concurrency	No
C++2011-N2340	__func__ predefined identifier	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1811	long long	Yes
C++2011-N1988	Extended integral types	Not applicable[1]

[1] This C++11 requirement is not a factor in a Polyspace verification.

See Also

"C++11 Extensions (C++)"

Verify C Application Without `main` Function

Polyspace verification requires that your code must have a `main` function. You can do one of the following:

- Provide a `main` function in your code.
- Specify that Polyspace must generate a `main`.

In this section...
"Generate `main` Function" on page 5-36
"Manually Write `main` Function" on page 5-36

Generate `main` Function

Before verification, specify one of the following options:

Option	Description
"Verify whole application (C/C++)"	The verification stops if the software does not detect a `main`.
"Verify module (C)"	Before verification, Polyspace checks if your code contains a `main` function. If a `main` function exists, the software uses that `main`. Otherwise, the software generates a `main` using the options that you specify: • "Variables to initialize (C)" • "Initialization functions (C)" • "Functions to call (C)"

Manually Write `main` Function

During automatic `main` generation, the software makes certain assumptions about the function call sequence or behavior of global variables. For instance, it assumes that:

- The functions that you specify using the option “Functions to call (C)” can be called in arbitrary order.
- In the beginning of each function body, global variables can have the full range of values allowed by their type.

To provide a more accurate model of the call sequence, you can manually write a `main` function for the purposes of verification. In some cases, providing the accurate call sequence can reduce the number of orange checks. For example, in the following code, Polyspace assumes that `f` and `g` can be called in any order. Therefore, it produces an orange overflow for the case when `f` is called before `g`. If you know that `f` is called after `g`, you can write a `main` function to model this sequence.

```
static char x;
static int y;

void f(void)
{
    y = 300;
}

void g(void)
{
    x = y;
}
```

To manually define a `main`:

1. Identify the functions that you want to call in the `main`. Copy their declarations.
2. Write a `main` containing declarations of a `volatile` variable for each function parameter type.
3. Write a loop with a `volatile` termination condition.
4. Inside this loop, write a `switch` block with a `volatile` condition.
5. For each function, write a `case` branch that calls the function using the `volatile` variable parameters that you created.

Consider the following example. Suppose the functions are:

```
int func1(void *ptr, int x);
void func2(int x, int y);
```

You can write the following `main`:

```
void main()
{
volatile int random;
volatile void * volatile ptr;
while (random) {
 switch (random) {
 case 1:
  random = func1(ptr, random); break;
 default:
  func2(random, random);
 }
}
```

Verify C++ Classes

In this section...

Verification of Classes

Object-oriented languages such as C++ are designed for reusability. When developing code in such a language, you do not necessarily know all contexts in which the class is deployed. A class or a class family is safe for reuse if it free of defects for all possible contexts. To make your classes safe against all possible contexts, perform a robustness verification and remove as many run-time errors as possible.

Polyspace Code Prover performs a robustness verification by default. If you provide the software the class definition together with the definition of the class methods, the software simulates alluses of the class. If some of the method definitions are missing, the software automatically stubs them.

1 The software verifies each constructor by creating an object using the constructor. If a constructor does not exist, the software uses the default constructor.

2 The software verifies the public, static and protected class methods of those objects assuming that:

   - The methods can be called in arbitrary order.
   - The method parameters can have any value in the range allowed by their data type.

   To perform this verification, by default, it generates a `main` function that calls the methods that are not called elsewhere in the code. If you want all your methods to be verified for all contexts, modify this behavior so that the generated `main` calls all public and protected methods instead of just the uncalled ones. For more information, see “Functions to call within the specified classes (C++)”.

3 The software calls the destructor of those objects (if they exist) and verifies them.

When verifying classes, Polyspace makes certain assumptions.

<table>
<tr><th>Code Construct</th><th>Assumption</th></tr>
<tr><td>Global variable</td><td>Unless explicitly initialized, in each method, global variables can have any value allowed by their type.
For instance, in the following code, Polyspace assumes that <code>globvar1</code> can have any value allowed by its type. Therefore, an orange <b>Division by zero</b> appears on the division by <code>globvar1</code>. However, because <code>globvar2</code> is explicitly initialized, the <b>Division by zero</b> check on division by <code>globvar2</code> is green.
<pre><code>extern int fround(float fx);

// global variables
int globvar1;
int globvar2 = 100;

class Location
{
private:
    int x;

public:
    Location(int intx = 0) {
        x = intx;
    };

    void setx(int intx) {
        x = intx;
    };

    void fsetx(float fx) {
        int tx = fround(fx);</code></pre></td></tr>
</table>

Code Construct	Assumption

	`        if (tx / globvar1 != 0)`
	`        {`
	`            tx = tx / globvar2;`
	`            setx(tx);`
	`        }`
	`    };`
	`};`

<table>
<tr><th>Code Construct</th><th>Assumption</th></tr>
<tr><td>Classes with undefined constructors</td><td>The members of the classes can be non-initialized.

In the following example, Polyspace assumes that m_loc.x can be non-initialized. Therefore, an orange <b>Non-initialized variable</b> error appears on x in the getMember method. Following the check, Polyspace assumes that the variable can have any value allowed by its type. Therefore, an orange <b>Overflow</b> appears on the addition operation in the show method.

<pre>class OtherClass
{
protected:
    int x;
public:
    OtherClass (int intx);
    int getMember(void) {
        return x;
    };
};

class MyClass
{
    OtherClass m_loc;
public:
    MyClass(int intx) : m_loc(0) {};
    void show(void) {
        int wx, wl;
        wx = m_loc.getMember();
        wl = wx + 2;
    };
};</pre></td></tr>
</table>

Methods and Class Specifics

Template Classes

A template class allows you to create a class without explicit knowledge of the data type that the class operations handle. Polyspace cannot verify a template class directly. The software can only verify a specific instance of the template class. To verify a template class:

1 Create an explicit instance of the class.

2 Define a `typedef` of the instance and provide that `typedef` for verification.

In the following example, `calc` is a template class that can handle any data type through the identifier `myType`.

```
template <class myType> class calc
{
public:
    myType multiply(myType x, myType y);
    myType add(myType x, myType y);
};
template <class myType> myType calc<myType>::multiply(myType x,myType y)
{
    return x*y;
}
template <class myType> myType calc<myType>::add(myType x, myType y)
{
    return x+y;
}
```

To verify this class:

1 Add the following code to your Polyspace project.

```
template class calc<int>;
typedef calc<int> my_template;
```

2 Provide `my_template` as argument of the option **Class**. See “Class (C++)”.

Abstract Classes

In the real world, an instance of an abstract class cannot be created, so it cannot be analyzed. However, it is easy to establish a verification by removing the pure declarations. For example, this can be accomplished via an abstract class definition change:

```
void abstract_func () = 0; by void abstract_func ();
```

If an abstract class is provided for verification, the software will make the change automatically and the virtual pure function (`abstract_func` in the example above) will then be ignored during the verification of the abstract class.

This means that no call will be made from the generated main, so the function is completely ignored. Moreover, if the function is called by another one, the pure virtual function will be stubbed and an orange check will be placed on the call with the message "call of virtual function [f] may be pure."

Static Classes

If a class defines a static methods, it is called in the generated main as a classical one.

Inherited Classes

When a function is not defined in a derived class, even if it is visible because it is inherited from a father's class, it is not called in the generated main. In the example below, the class `Point` is derived from the class `Location`:

```
class Location
{
protected:
 int x;
 int y;
 Location (int intx, int inty);
public:
 int getx(void) {return x;};
 int gety(void) {return y;};
};
class Point : public Location
{
protected:
 bool visible;
public :
 Point(int intx, int inty) : Location (intx, inty)
 {
 visible = false;
 };
 void show(void) { visible = true;};
 void hide(void) { visible = false;};
 bool isvisible(void) {return visible;};
};
```

Although the two methods `Location::getx` and `Location::gety` are visible for derived classes, the generated main does not include these methods when analyzing the class `Point`.

Inherited members are considered to be volatile if they are not explicitly initialized in the father's constructors. In the example above, the two members `Location::x` and `Location::y` will be considered volatile. If we analyze the above example in its current state, the method `Location:: Location(constructor)` will be stubbed.

Simple Class

Consider the following class:

`Stack.h`

```
#define MAXARRAY 100

class stack
{
  int array[MAXARRAY];
  long toparray;

public:
  int top (void);
  bool isempty (void);
  bool push (int newval);
  void pop (void);
  stack ();
};
```

`stack.cpp`

```
#include "stack.h"

stack::stack ()
{
 toparray = -1;
 for (int i = 0 ; i < MAXARRAY; i++)
7  array[i] = 0;
8 }
9
10 int stack::top (void)
11 {
12  int i = toparray;
13  return (array[i]);
14 }
15
16 bool stack::isempty (void)
17 {
18  if (toparray >= 0)
19   return false;
20  else
21   return true;
22 }
23
24 bool stack::push (int newvalue)
25 {
26  if (toparray < MAXARRAY)
27  {
28   array[++toparray] = newvalue;
29   return true;
30  }
31
32  return false;
33 }
34
35 void stack::pop (void)
36 {
37  if (toparray >= 0)
38   toparray--;
39 }
```

The class analyzer calls the constructor and then all methods in any order many times.

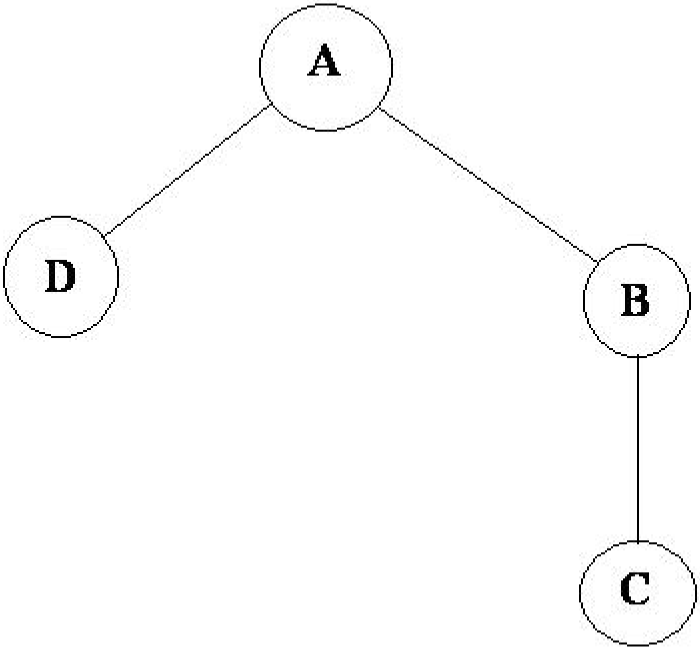
The verification of this class highlights two problems:

- The `stack::push` method may write after the last element of the array, resulting in the OBAI orange check at line 28.
- If called before `push`, the `stack::top` method will access element -1, resulting in the OBAI and NIV checks at line 13.

Fixing these problems will eliminate run-time errors in this class.

Simple Inheritance

Consider the following classes:



A is the base class of B and D.

B is the base class of C.

In a case such a this, Polyspace software allows you to run the following verifications:

1. You can analyze class A just by providing its code to the software. This corresponds to the previous "Simple Class" section in this chapter.
2. You can analyze class B class by providing its code and the class A declaration. In this case, A code will be stubbed automatically by the software.
3. You can analyze class B class by providing B and A codes (declaration and definition). This is a "first level of integration" verification. The class analyzer will not call A methods. In this case, the objective is to find bugs only in the class B code.

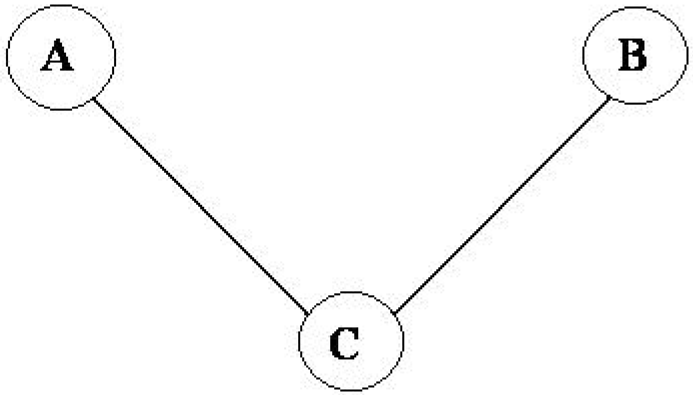
4 You can analyze class C by providing the C code, the B class declaration and the A class declaration. In this case, A and B codes will be stubbed automatically.

5 You can analyze class C by providing the A, B and C code for an integration verification. The class analyzer will call all the C methods but not inherited methods from B and A. The objective is to find only defects in class C.

In these cases, there is no need to provide D class code for analyzing A, B and C classes as long as they do not use the class (e.g., member type) or need it (e.g., inherit).

Multiple Inheritance

Consider the following classes:



A and B are base classes of C.

In this case, Polyspace software allows you to run the following verifications:

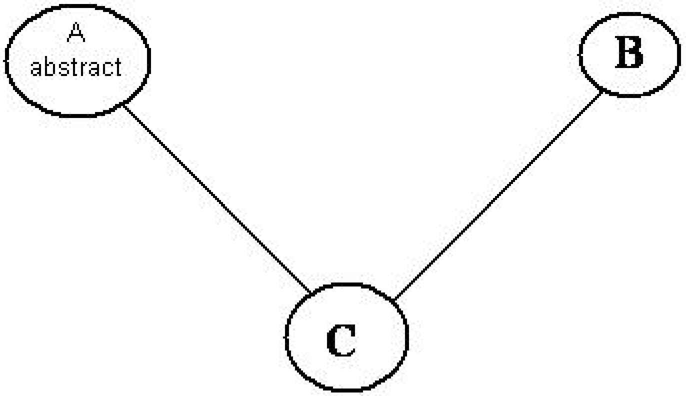
1 You can analyze classes A and B separately just by providing their codes to the software. This corresponds to the previous "Simple Class" section in this chapter.

2 You can analyze class C by providing its code with A and B declarations. A and B methods will be stubbed automatically.

3 You can analyze class C by providing A, B and C codes for an integration verification. The class analyzer will call all the C methods but not inherited methods from A and B. The objective is to find bugs only in class C.

Abstract Classes

Consider the following classes:



A is an abstract class

B is a simple class.

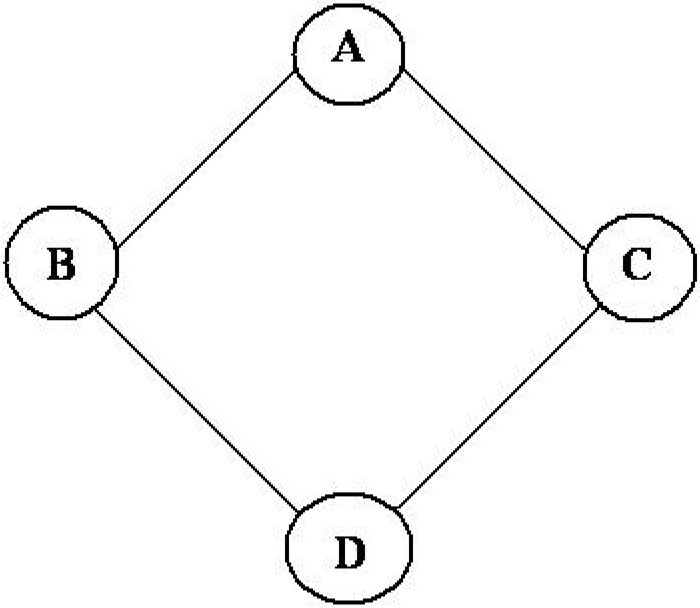
A and B are base classes of C.

C is not an abstract class.

As it is not possible to create an object of class A, this class cannot be analyzed separately from other classes. Therefore, you are not allowed to specify class A to the Polyspace class analyzer. Of course, class C can be analyzed in the same way as in the previous section “Multiple Inheritance.”

Virtual Inheritance

Consider the following classes:



B and C classes virtually inherit the A class

B and C are base classes of D.

A, B, C and D can be analyzed in the same way as described in the previous section “Abstract Classes.”

Virtual inheritance has no impact on the way of using the class analyzer.

Other Types of Classes

Template Class

A template class can not be analyzed directly. But a class instantiating a template can be analyzed by Polyspace software.

Note: If only the template declaration is provided, missing functions' definitions will automatically be stubbed.

Example

```
template<class T > class A {
public:
 T i;
 T geti() {return i;}
 A() : i(1) {}
};
```

You have to define a `typedef` to create a specialization of the template:

```
template class A<int>;      // Explicit specialization
typedef class A<int> my_template;  // complete instance of the template
```

and use option `-class-analyzer my_template`.

The software will analyze a single instance of the template.

Class Integration

Consider a `C` class that inherits from `A` and `B` classes and has object members of `AA` and `BB` classes.

A class integration verification consists of verifying class `C` and providing the codes for `A`, `B`, `AA` and `BB`. If some definitions are missing, the software will automatically stub them.

Specify Constraints

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

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

Because of these broad assumptions, Polyspace can 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.

To reduce the number of such false positives, you can specify additional constraints on variables using a **Data Range Specification** or DRS template. After you create a template, you can save the template as an XML file and use it for subsequent verifications. If your source code changes, you can update the previous template. You do not have to create another template.

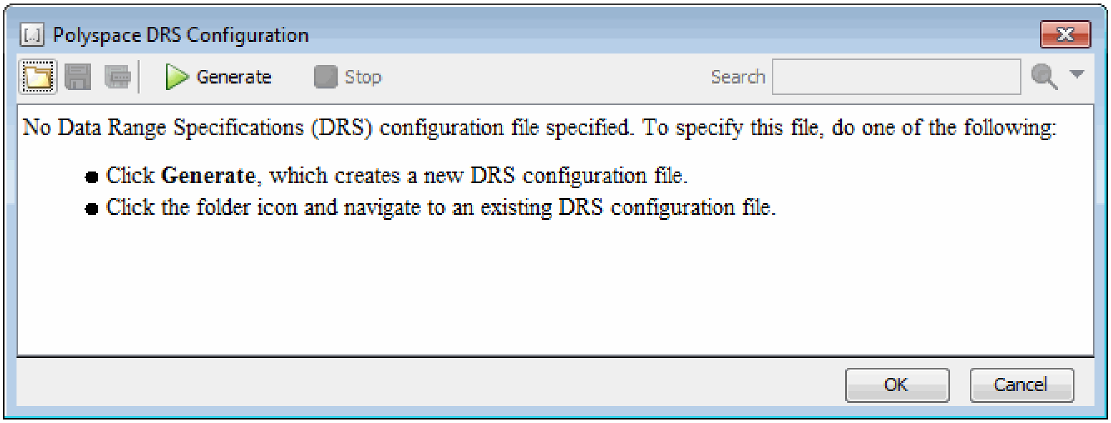
In this section...

Create Constraint Template

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

2 To the right of **Variable/function range setup**, click the **Edit** button.

The Polyspace DRS Configuration dialog box opens.



No Data Range Specifications (DRS) configuration file specified. To specify this file, do one of the following:

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

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

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

4 Specify your constraints and save the template as an XML file. For more information, see "Constraints".

5 Click **OK**.

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

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

To specify constraints in your code, you can use:

- Appropriate error handling tests in your code.
  Polyspace checks if the errors can actually occur. If they do not occur, the test blocks appear as **Unreachable code**.
- The assert macro. For example, to constrain a variable `var` in the range [0,10], you can use `assert(var >= 0 && var <=10);`.

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

Create Constraint Template After Verification

When you create a template starting from verification results, you know which variables you must constrain to avoid false positives.

1. Open your results. Browse orange checks.
2. If the software can trace an orange check to a root cause, a ? icon appears active on the **Check Details** pane. Click this icon.

   The root cause is highlighted on the **Orange Sources** pane.
3. If you can address the root cause using constraints, on the **Orange Sources** pane, in the **Suggestion** column, you see the Add DRS button. Click this button.
4. Specify your constraints on the **Data Range Configuration** pane.

   For more information, see "Constraints".
5. Save the constraint template as an XML file.

   You can use this file for subsequent verifications.
6. To use the template file for verification:

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

   b. In the **Variable/function range setup** field, enter the full path to the file.

Update Existing Template

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

   - Enter the path to the template XML file in the **Variable/function range setup** field. Click **Edit**.

- Click **Edit**. In the Polyspace DRS Configuration dialog box, click the 📁 icon, to navigate to your template file.

3 Click **Update**.

   a Variables that are no longer present in your source code appear under the **Non Applicable** node. To remove an entry under the **Non Applicable** node or the node itself, right-click and select **Remove This Node**.

   b Specify your new constraints for any of the other variables.

See Also

"Variable/function range setup (C/C++)"

Related Examples

- "Constrain Global Variables"

More About

- "XML File Format for Constraints"

Constrain Global Variables

This example shows how to use the **Global Assert** mode to constrain the range of a global variable. In this mode, Polyspace performs a **Correctness condition** check on each write access to the global variable. After the write access, this check determines whether the variable is within the range that you specified.

For the more general workflow on specifying constraints, see “Specify Constraints”.

1. Run verification on your code and open the results.
2. On the **Source** pane, select the **Data Range Configuration** tab.

   Under the **Global Variables** node, you see a list of global variables.
3. For the global variable that you want to constrain, from the drop-down list on the **Global Assert** column, select `YES`.
4. In the **Global Assert Range** column, enter the range in the format *`min..max`*. *`min`* is the minimum value and *`max`* the maximum value for the global variable.
5. To save your specifications, click the 💾 button.

   A **Save Data Range Specifications (DRS) as** window opens. Save your entries as an `xml` file.
6. Open your project configuration. On the **Configuration** pane, under **Inputs & Stubbing**, in the **Variable/function range setup** field, enter the full path to the `xml` file.

   Instead of typing the location, you can use the Edit button to navigate to the location of the `.xml` file.
7. Rerun the verification and open the results.

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

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

In a multitasking application, when two or more tasks access the same global variable, if a **Correctness condition** check on a write access in one task turns orange, the **Correctness condition** check on write accesses in all other tasks appear orange. The other orange checks appear even if the other write accesses do not take the variable outside the **Global Assert** range.

See Also

"Variable/function range setup (C/C++)" | Correctness condition

More About

- "Constraints"

Constrain Stubbed Functions

By default, Polyspace stubs undefined functions. For more information on Polyspace assumptions about stubbed functions, see “Polyspace Software Assumptions”.

To work around the Polyspace assumptions, you can do one of the following:

- Specify analysis options so that Polyspace does not stub functions.

  However, if you do not have the definition for a function, the verification stops. You must provide the function definition or define your own stub.
- Specify constraints on arguments and return values of stubbed functions.

In this section...

Define Stubs for Functions

To prevent Polyspace from automatically stubbing functions:

1. On the **Configuration** pane, select **Inputs & Stubbing**.
2. Select **No automatic stubbing**.

   If your source files contain undefined functions, the verification stops.
3. Update your project by adding source files containing stubs for undefined functions. For more information, see “Add Sources and Includes”.

   For example, if a function `func` is declared as:

   ```
   int func(int *x);
   ```
   and not defined, Polyspace considers that `func` potentially writes to its argument. To work around this assumption, provide an empty stub:

   ```
   int func(int *x) {}
   ```

Constrain Function Arguments and Return Values

Polyspace makes certain assumptions about the arguments and return values of undefined functions. You can specify constraints to narrow down these assumptions.

For example, Polyspace assumes that variables returned from undefined functions take full range of values allowed by their type. You can specify that the variable returned by a certain undefined function lies in a specific range.

To specify a constraint, do one of the following:

- Before verification, create a constraint template. Specify this template for verification.

  If you want to specify constraints for all undefined functions, use this approach. For more information, see “Create Constraint Template”.
- Create a constrain template from your verification results. Specify this template for the next verification.

  If you want to constrain only those undefined functions that cause noncritical orange checks, use this approach. For more information, see “Create Constraint Template After Verification”.

Constraints

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

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

For more information, see “Specify Constraints”.

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

Column	Settings
Name	Displays the list of variables and functions in your Project for which you can specify data ranges. This Column displays three expandable menu items: • **Globals** – Displays global variables in the project. • **User defined functions** – Displays user-defined functions in the project. Expand a function name to see its inputs. • **Stubbed functions** – Displays a list of stub functions in the project. Expand a function name to see the inputs and return values.
File	Displays the name of the source file containing the variable or function.
Attributes	Displays information about the variable or function. For example, static variables display `static`.
Data Type	Displays the variable type.
Main Generator Called	Applicable only for user-defined functions. Specifies whether the main generator calls the function: • `MAIN GENERATOR` – Main generator may call this function, depending on the value of the `-functions-called-in-loop` (C) or `-main-generator-calls` (C++) parameter. • `NO` – Main generator will not call this function.

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

Column	Settings

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

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

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

Pointer Examples

For pointer p, **# Allocated objects** = `1`, and **Init Allocated** = `Single`:

```
void f(int *p) {
  int x;
  x = p[0]; // green IDP, green NIV
  x = p[1]; // red IDP: out of bounds
}
```

Note: Pointer p may point to any element inside the array.

For pointer p (a pointer to int), **# Allocated objects** = 3, and **Init Allocated** = MULTI:

```
void f(int *p) {
  int x;
  x = p[0]; // green IDP, green NIV
  x = p[1]; // orange IDP, green NIV
  x = p[2]; // orange IDP, green NIV
  x = p[3]; // red IDP: out of bounds
}
```

XML File Format for Constraints

Syntax Description — XML Elements

The DRS file contains the following XML elements:

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

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

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field line contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the GUI to compute the min and max values. The field comment is used to add information about any node.
- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a `struct` field.

- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:

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

  For example, the value "**10**" means that modes "`INIT`" and "`MAIN_GENERATOR`" are allowed. To see how this value is computed, refer to "Valid Modes and Default Values" on page 5-69.
- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to SINGLE or MULTI.

<`file`> Element

Field	Syntax
`name`	*`filepath_or_filename`*
`comment`	*`string`*

<`scalar`> Element

Field	Syntax
`name` (**)	*`name`*
`line` (*)	*`line`*

`base_type` (*)	`intx`
	`uintx`
	`floatx`

`Attributes` (***)	`volatile`
	`extern`
	`static`
	`const`

Field	Syntax
`complete_type` (*)	*type*

`init_mode`	`MAIN_GENERATOR`
	`IGNORE`
	`INIT`
	`PERMANENT`
	`disabled`
	`unsupported`

`init_modes_allowed` (*)	*single value* (****)

`init_range`	*range*
	`disabled`
	`unsupported`

`global_ assert`	`YES`
	`NO`
	`disabled`
	`unsupported`

`assert_range`	*range*
	`disabled`
	`unsupported`

`comment`(*)	*string*

`<pointer>` Element

Field	Syntax
`Name` (**)	*name*
`line` (*)	*line*

`Attributes` (***)	`volatile`
	`extern`
	`static`
	`const`

`complete_type` (*)	*type*

`init_mode`	`MAIN_GENERATOR`
	`IGNORE`
	`INIT`
	`PERMANENT`

Field	Syntax
	disabled unsupported
init_modes_allowed (*)	*single value* (****)
initialize_ pointer	May be: NULL Not NULL NULL
number_ allocated	*single value* disabled unsupported
init_pointed	MAIN_GENERATOR NONE SINGLE MULTI disabled
comment	*string*

<array> and <struct> Elements

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

<function> Element

Field	Syntax
Name (**)	*name*
line (*)	*line*

Field	Syntax
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	*string*

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependant
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT PERMANENT	YES NO			INIT min..max
	Struct	Struct field	Refer to field type				
	Array	Array element	Refer to element type				
Global variables	Pointer	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependant
		Volatile pointer	un- supported		un- supported	un- supported	
		Extern pointer	IGNORE INIT		May be NULL Not NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
					NULL		
		Pointed volatile scalar	unsupported	unsupported			
		Pointed extern scalar	INIT	unsupported			INIT min..max
		Pointed other scalars	MAIN_GENERATOR INIT	unsupported			MAIN_GENERATOR dependant
		Pointed pointer	MAIN_GENERATOR INIT/	unsupported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_GENERATOR dependant
		Pointed function	unsupported	unsupported			
Function parameters	Userdef function	Scalar parameters	MAIN_GENERATOR INIT	unsupported			INIT min..max
		Pointer parameters	MAIN_GENERATOR INIT	unsupported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Other parameters	Refer to parameter type				
	Stubbed function	Scalar parameter	disabled	unsupported			
		Pointer parameters	disabled		disabled	NONE SINGLE MULTI	MULTI
		Pointed parameters	PERMANENT	unsupported			PERMANENT min..max

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointed const parameters	disabled	un-supportec			
Function return	Userdef function	Return	disabled	un-supportec	disabled	disabled	
	Stubbed function	Scalar return	PERMANENT	un-supportec			PERMANENT min..max
		Pointer return	PERMANENT	un-supportec	May be NULL Not NULL NULL	NONE SINGLE MULTI	PERMANENT May be NULL max MULTI

Provide Context for C Code Verification

This example shows how to provide context for your C code verification. If you use default options and do not provide a `main` function, Polyspace Code Prover checks your code for robustness against all verification conditions. For instance, the software:

- Considers that global variables and inputs of uncalled functions are full range.
- Generates a `main` that calls uncalled functions in arbitrary order.

In addition, if you do not define a function but declare and call it in your code, Polyspace stubs the function. For a detailed list of assumptions, see “Polyspace Software Assumptions”.

You can use analysis options on the **Configuration** pane to change the default behavior and provide more context about your code. Performing contextual verification can result in more proven code and therefore fewer orange checks.

Control Variable Range

Use the following options. The options appear under the **Code Prover Verification** node.

Option	Purpose
“Variables to initialize (C)”	Specify the global variables that Polyspace must consider as initialized despite no explicit initialization in the code.
“Variable/function range setup (C/C++)”	Specify range for global variables.

Control Function Call Sequence

Use the following options. The options appear under the **Code Prover Verification** node.

Option	Purpose
“Initialization functions (C)”	Specify the functions that the generated `main` must call first.
“Functions to call (C)”	Specify the functions that the generated `main` must call later.

Control Stubbing Behavior

Use the following options. The options appear under the **Inputs & Stubbing** node.

Option	Purpose
"No automatic stubbing (C/C++)"	Specify that verification must stop if a function is not defined in the source files.
"Functions to stub (C)"	Specify the functions that Polyspace must stub.

Provide Context for C++ Code Verification

This example shows how to provide context to your C++ code verification. If you use default options and do not provide a `main` function, Polyspace Code Prover checks your code for robustness against all verification conditions. For instance, the software:

- Considers that global variables and inputs of uncalled functions and methods are full range.
- Generates a `main` that calls uncalled functions in arbitrary order.

In addition, if you do not define a function but declare and call it in your code, Polyspace stubs the function. For a detailed list of assumptions, see "Polyspace Software Assumptions".

You can use analysis options on the **Configuration** pane to change the default behavior and provide more context about your code. Performing contextual verification can result in more proven code and therefore fewer orange checks.

Control Variable Range

Use the following options. The options appear under the **Code Prover Verification** node.

Option	Purpose
"Variables to initialize (C++)"	Specify the global variables that Polyspace must consider as initialized despite no explicit initialization in the code.
"Variable/function range setup (C/C++)"	Specify range for global variables.

Control Function Call Sequence

1 Use the following options to call class methods. The options appear under the **Code Prover Verification** node.

Option	Purpose
"Class (C++)"	Specify classes whose methods the generated `main` must call.
"Functions to call within the specified classes (C++)"	Specify methods that the generated `main` must call.

Option	Purpose
"Analyze class contents only (C++)"	Specify that the generated `main` must call class methods only.
"Skip member initialization check (C++)"	Specify that the generated `main` must not check whether each class constructor initializes all class members.

2 Use the following options to call functions that are not class methods. The options appear under the **Code Prover Verification** node.

Option	Purpose
"Initialization functions (C++)"	Specify the functions that the generated `main` must call first.
"Functions to call (C++)"	Specify the functions that the generated `main` must call later.

Control Stubbing Behavior

Use the following options. The options appear under the **Inputs & Stubbing** node.

Option	Purpose
"No automatic stubbing (C/C++)"	Specify that verification must stop if a function is not defined in the source files.
"No STL stubs (C++)"	Specify that the verification must not use Polyspace implementations of the standard template library.
"Functions to stub (C)"	Specify the functions that Polyspace must stub.

Preparing Source Code for Verification

Stubbing Overview

A function stub is a piece of code that models a function whose body is not provided during verification.

Stubs need not model the details of functions. Depending on your requirements, you can:

- Provide the function argument types and return types.
- Provide a bound on the function arguments and return values.
- Provide other details about how the function relates to the rest of the code.

Stubbing allows you to verify code before functions are developed. The more closely your stub models the actual function, the more precise the verification results will be.

Unless you specify the option **Inputs & Stubbing** > **No automatic stubbing**, Polyspace automatically stubs undefined functions.

When to Provide Function Stubs

By default, Polyspace software automatically stubs undefined functions. Stub functions manually when:

- You note that the automatic stubs do not represent your function arguments and return values. For instance, the automatic stubs can return a broader range of values than you want.
- You want your source code to be complete. If you specify the option **No automatic stubbing**, verification stops if a function is not defined. This behavior allows you to detect undefined functions.
- You want to reduce unproven code. Sometimes, automatic stubs do not provide sufficient information to allow Polyspace to prove presence or absence of run-time errors.
- Your function modifies global variables. Automatic stubs cannot model this behavior.

Manual stubs

If a function `func` represents:

- A timing constraint such as a timer set/reset, a task activation, a delay, or a counter of ticks between two precise locations in the code, stub `func` with an empty action

  ```
  void func(void) {
  }
  ```
  Polyspace takes into account scheduling and interleaving of concurrent execution. Therefore, do not stub functions that set or reset a timer. Declare the variable representing time as volatile.

- An I/O access, such as to a hardware port, a sensor, a read/write of a file, a read of an EEPROM, or a write to a volatile variable, then,

  - You do not need to stub a *write* access. If you want to do so, stub a write access to an empty action (`void func(void)`).
  - Stub *read* accesses to "read all possible values (volatile)".

- A write to a global variable, you may need to consider which procedures or functions write to `func` and why. Do not stub the concerned `func` if:

  - The variable is volatile.
  - The variable is a task list. Such lists are accounted for by default because tasks declared with the `-task` option are automatically modelled as though they have been started. Write `func` manually if:

    - The variable is a regular variable read by other procedures or functions.
    - The variable is a read from a global variable. If you want Polyspace software to detect that the variable is a shared variable, stub a read access. Copy the value into a local variable.

Provide Stubs for Functions

The following example shows a header for a missing function (which might occur, for example, if the code is a subset of a project). The missing function copies the value of the `src` parameter to `dest` so there would be a division by zero, a run-time error..

```
void main(void)
{
 a = 1;
 b = 0;
 a_missing_function(&a, b);
 b = 1 / a;
}
```

Due to the reliance on the software's default stub, the division is shown with an orange warning because a is assumed to be anywhere in the full permissible integer range (including 0). If the function is commented out, then the division would be a green "/ ". You could only achieve a red "/ " with a manual stub.

Default Stubbing	Manual Stubbing	Function Ignored

void main(void)
{
 a = 1;
 b = 0;
 a_missing_function(&a,
b);
 b = 1 / a;
// orange division
}
	void a_missing_function
	(int *x, int y;)
	{ *x = y; }
	
	void main(void)
	{
	 a = 1;
	 b = 0;
	 a_missing_function(&a,
	b);
	 b = 1 / a;
	// red division
		void a_missing_function
		(int *x, int y;)
		{ }
		
		void main(void)
		{
		 a = 1;
		 b = 0;
		 a_missing_function(&a,
		b);
		 b = 1 / a;
		// green division

Due to the reliance on the software's default stub, the software ignores the assembly code and the division " /" is green. You could only achieve the red division "/" with a manual stub.

Stubbing Examples

The following examples consider the pros and cons of manual and automatic stubbing.

Example: Specification

```
typedef struct _c {
int cnx_id;
int port;
int data;
} T_connection ;

int Lib_connection_create(T_connection *in_cnx) ;
int Lib_connection_open (T_connection *in_cnx) ;
```

File: connection_lib		**Function: Lib_connection_create**
param in	None	
param in/out	in_cnx	all fields might be changed in case of a success
returns	int	0 : failure of connection establishment 1 : success

Note: Default stubbing is suitable here.

Here are the reasons why:

- The content of the *in_cnx* structure might be changed by this function.
- The possible return values of 0 or 1 compared to the full range of an integer wont have much impact on the Run-Time Error aspect. It is unlikely that the results of this operation will be used to compute some mathematical algorithm. It is probably a Boolean status flag and if so is likely to be stored and compared to 0 or 1. Therefore, the default stub does not have a detrimental effect.

File: connection_lib		**Function: Lib_connection_open**
param in	T_connection *in_cnx	in_cnx->cnx_id is the only parameter used to open the connection, and is a read-only parameter

File: connection_lib		**Function: Lib_connection_open**
		cnx_id, port and data remain unchanged
param in/out	None	
returns	int	0 : failure of connection establishment 1 : success

Note: Default stubbing works here but manual stubbing would give more benefit.

Here are the reasons why:

- For the return value, default stubbing would be applicable as explained in the previous example.
- Since the structure is a read-only parameter, it will be worth creating manually a stub that reflects the behavior of the missing code. Benefits: Polyspace verification will find more red and gray code

Note: Even in the examples above, it concerns some C code like; stubs of functions members in classes follow same behavior.

Example: Colored Source Code

```
typedef struct _c {
int a;
int b;
} T;

void send_message(T *);
void main(void)
{
int i;
T x = {10, 20};
send_message(&x);
i = x.b /x.a;  // orange with the default stubbing
}
```

Suppose that it is known that send_message does not write into its argument. The division by x.a will be orange if default stubbing is used, warning of a potential division by zero. A manual stub that accurately reflects the behavior of the missing code will result in a green division instead, thus increasing the selectivity.

Manual stubbing examples for send_message:

```
void send_message(T *) {}
```

In this case, an empty function would be a sound manual stub.

Automatic Stubbing Behavior for C++ Pointer/Reference

For parameters of a pointer/reference type, the behavior of automatically stubbed C++ functions differs from the behavior of automatically stubbed C functions. As a result, automatic stubs for C++ do not always write to their arguments.

For C++, the software stubs functions by randomizing the contents of the object passed as actual of the stubbed function, but does not modify the object pointed to by the actual (or by one component of the actual if the latter is a `struct/class` object or an array).

Consider the following example:

```
extern void stub_def_pointer(struct S *p);
extern void stub_def_array(struct S *p);

int fx = 0, fw = 0;
struct S def = {"-dummy", &fx};
struct S def_array[] = {{ "-foo", &fw } };

assert(*(def.pvar) == 0); // GREEN
stub_def_pointer(&def);
assert(fx == 0);          // GREEN because stubbed stub_def_pointer
          // does not write *(def.pvar)

assert(*(def_array[0].pvar) == 0);  // GREEN
stub_def_array(def_array);
assert(fw == 0);       // GREEN because stubbed stub_def_array
                        // does not write *(def_array[0].pvar)
```

In this situation, you should manually stub the missing routine. For example, you could stub `stub_def_pointer` and `stub_def_array` as follows:

```
volatile int rd;

void stub_def_pointer(struct S *p)
 {
  *(p->pvar) = rd;   // write the object pointed to by p->pvar
 }

void stub_def_array(struct S *p)
 {
  int i = rd;
  for (i; i < rd; i++)
  {
   *(p[i].pvar) = rd; // write the object pointed to
          // by p[i]->pvar
   i++;
  }
 }
```

Using these manual stubs, the verification result become:

```
assert(*(def.pvar) == 0);           // GREEN
stub_def_pointer(&def);
assert(fx == 0);                    // ORANGE

assert(*(def_array[0].pvar) == 0); // GREEN
stub_def_array(def_array);
assert(fw == 0);                    // ORANGE
```

Specify Functions to Stub Automatically

You can specify a list of functions that you want the software to stub automatically.

To specify functions to stub:

1. On the **Configuration** pane, select **Inputs & Stubbing**.
2. To the right of the **Functions to stub** view, click ✚. The software creates a new row.
3. In the new row, enter the name of a function that you want to stub. Enter one function name per row.

Special Characters in Function Names

The following special characters are allowed for C functions:
`( ) < > ; _`

The following special characters are allowed for C++:
`( ) < > ; _ * & [ ]`

Space characters are allowed for C++, but are not allowed for C functions.

Function Syntax for C++

When entering function names, two syntaxes are supported for C++:

- Basic syntax, with extensions for classes and templates:

Function Type	Syntax
Simple function	`test`
Class method	`A::test`
Template method	`A<T>::test`

- Syntax with function arguments, to differentiate overloaded functions. Function arguments are separated with semicolons:

Function Type	Syntax
Simple function	`test()`

Function Type	Syntax
Class method	`A::test(int;int)`
Template method	`A<T>::test(T;T)`

Note: Overloaded versions of the function will be discarded.

Constrain Data with Stubbing

In this section...

Add Precision Constraints Using Stubs

You can improve the selectivity of your verification by using stubs to indicate that some variables vary within functional ranges instead of the full range of the considered type.

You can apply this approach to:

- Parameters passed to functions.
- Variables that change from one execution to another (mostly globals), for example, calibration data or mission specific data. These variables might be read directly within the code, or read through an API of functions.

If a function returns an integer, default automatic stubbing assumes the function can take any value from the full range of the integer type. This can lead to unproven code (orange checks) in your results. You can achieve more precise results by providing a manual stub that provides external data that is representative of the data expected when the code is implemented.

There are a number of ways to model such data ranges within the code. The following table shows some approaches.

with volatile and assert	with assert and without volatile	without assert, without volatile, without "if"

`#include <assert.h>`
 	
`int stub(void)`
`{`
` volatile int random;`
	`#include <assert.h>`
	
	`extern int other_func(void);`
	`int stub(void)`
	`{`
		
		`extern int other_func(void);`
		`int stub(void)`
		`{`

<table>
<tr><td>int tmp;
tmp = random;
assert(tmp>=1 && tmp<=10);
return</td><td>int tmp;
tmp= other_func();
assert(tmp>=1 && tmp<=10);
return
}</td><td>int tmp;
do {tmp= other_func();}
while (tmp<1 || tmp>10);
return tmp;
}</td></tr>
</table>

There is no particular advantage to any one of these approaches, except that the assertions in the first two approaches can produce orange checks in your results.

Default Behavior of Global Data

Initially, consider how Polyspace verification handles the verification of global variables.

There is a maximum range of values which may be assigned to each variable as defined by its type. By default, Polyspace verification assigns that full range for each global variable, ensuring that a meaningful verification of such a variable can take place even when the functions that write to it are not included. If a range of values was not considered in these circumstances, such a variable would be assumed to have a value of zero throughout.

Sometimes, to reflect practical use, it is helpful to limit the range of values assigned to some variables . These ranges will be propagated to the whole call tree, and hence will limit the number of "impossible values" that are considered throughout the verification.

This thinking does not just apply to global variables; it is equally appropriate where such a variable is passed as a parameter to a function, or where return values from stubbed functions are under consideration.

To some extent, the effectiveness of this technique is limited by compromises made by Polyspace verification to deal with issues of code complexity. For instance, you cannot assume that all of these ranges will be propagated throughout all function calls. Sometimes, perhaps as a result of complex function interactions or constructions where Polyspace verification is known to be imprecise, the potential value of a variable will assume its full "type" range despite this technique having been applied.

Constraining the Data

Restricting data, such as global variables, to a functional range can be a useful technique if the process can be automated. The technique may not be advantageous if the process requires significant manual effort.

The technique requires:

- A knowledge of the variables and the maximum ranges they may take in practice.
- A data dictionary in electronic format from which the variable names and their minimum and maximum values can be extracted.

Apply the Technique

1 Create the range setting stubs:

  a create 6 functions for each type (8,16 or 32 bits, signed and unsigned)

  b declare 6 global volatile variables for each type

  c write the functions which returns sub-ranges (an example follows)

2 Gather the initialization of relevant variables into a single procedure

3 Call this procedure at the beginning of the main. This should replace existing initialization code.

Integer Example

```
volatile int tmp;

int polyspace_return_range(int min_value, int max_value)
{
int ret_value;

ret_value = tmp;
assert (ret_value>=min_value && ret_value<=max_value);

return ret_value;
}
void init_all(void)
{
x1 = polyspace_return_range(1,10);
x2 = polyspace_return_range(0,100);
x3 = polyspace_return_range(-10,10);
}

void main(void)
{
init_all();

while(1)
```

```
 {
 if (tmp) function1();
 if (tmp) function2();
 // ...
 }
}
```

Recode Specific Functions

Once data ranges have been specified (above), it may be beneficial to recode some functions in support of them.

Sometimes, perhaps as a result of complex function interactions or constructions where Polyspace verification is known to be imprecise, the potential value of a variable will assume its full “type” range data ranges having been restricted. Recoding those complex functions will address this issue.

Identify in the modules:

- API which read global variables through pointers

  Replace this API:

```
 typedef struct _points {
 int x,y,nb;
 char *p;
 }T;

 #define MAX_Calibration_Constant_1 7
char Calibration_Constant_1[MAX_Calibration_Constant_1] =     \
 { 1, 50, 75, 87, 95, 97, 100} ;
 T Constant_1 = { 0, 0,
   MAX_Calibration_Constant_1,
   &Calibration_Constant_1[0] } ;

 int read_calibration(T * in, int index)
 {
 if ((index <= in->nb) && (index >=0)) return in->p[index];
 }

 void interpolation(int i)
 {
 int a,b;

 a= read_calibration(&Constant_1,i);
 }
```

  With this one:

```
char Constant_1 ;

#define read_calibration(in,index) *in

void main(void)
{
Constant_1 = polyspace_return_range(1, 100);
}

void interpolation(int i)
{
int a,b;

a= read_calibration(&Constant_1,i);
}
```

- Points in the source code which expand the data range perceived by Polyspace verification
- Functions responsible for full range data, as shown by the VOA (Value on assignment) check.

  if direct access to data is responsible, define the functions as macros.

  ```
  #define read_from_data(param) read_from_data##param

  int read_from_data_my_global1(void)
  { return [a functional range for my_global1]; }

  Char read_from_data_my_global2(void)
  { }
  ```

- stub complicated algorithms, calibration read accesses and API functions reading global data - as usual. For instance, if an algorithm is iterative - stub it.
- variables

  - where the data range held by each element of an array is the same, replace that array with a single variable.
  - where the data range held by each element of an array differs, separate it into discrete variables.

Default and Alternative Behavior for Stubbing

External functions are assumed to have no effect (read, write) on global variables. Any external function for which this assumption is not valid must be explicitly stubbed.

Consider the example `int f(char *);`.

When verifying this function, there are three options for automatic stubbing, as shown in the following table.

Approach	Worst Case Scenario in Stub

Default automatic stubbing	`int f(char *x)`
	`{`
	` *x = rand();`
	` return 0;`
	`}`

pragma POLYSPACE_WORST	`int f(char *x)`
	`{`
	` strcpy(x, "the quick`
	` brown fox, etc.");`
	` return &(x[2]);`
	`}`

pragma POLYSPACE_PURE	`int f(char *x)`
	`{`
	` return strlen(x);`
	`}`

If the automatic stub does not accurately model the function using any of these approaches, you can use manual stubbing to achieve more precise results.

PURE and WORST Stubbing Examples

The following table provides examples of stubbing approaches.

Initial Prototype	With pragma POLYSPACE_PURE	With pragma POLYSPACE_WORST	Default Automatic Stubbing
`void f1(void);`	Do nothing		

`int f2`
` (int u);`
	Returns [-2^31, 2^31-1]		Returns [-2^31, 2^31-1]

<table>
<tr><th>Initial Prototype</th><th>With pragma POLYSPACE_PURE</th><th>With pragma POLYSPACE_WORST</th><th>Default Automatic Stubbing</th></tr>
<tr><td><code>int f3</code>
<code> (int *u);</code></td><td></td><td>Returns [-2^31, 2^31-1] <b>and</b> assumes the ability to write into (int *) u</td><td>Assumes the ability to write into *u to any depth <b>and</b> returns [-2^31, 2^31-1]</td></tr>
<tr><td><code>int* f4</code>
<code> (int u);</code></td><td>Returns an absolute address (AA)</td><td>Returns AA or (int *) u <b>and</b> assumes the ability to write into (int *) u</td><td>Returns an absolute address</td></tr>
<tr><td><code>int* f5</code>
<code> (int *u);</code></td><td>Returns an absolute address</td><td>Returns [-2^31, 2^31-1] <b>and</b> assumes the ability to write into *u, to any depth</td><td>Assumes the ability to write into *u, to any depth <b>and</b> returns an absolute address</td></tr>
<tr><td><code>void f6</code>
<code> (void (*ptr)(int),</code>
<code>param2)</code></td><td rowspan="2">Does nothing</td><td colspan="2">The function pointed to by ptr is called with a full-range random value for the integer. Rules for param2 are the same as the preceding rules.</td></tr>
<tr><td><code>void f7</code>
<code> (void (*ptr)(</code>
<code>param2)</code></td><td colspan="2">This function is not stubbed. The parameter (<code>int *</code>) associated with the function pointer is too complicated for the software to stub it, and verification stops. You must stub this function manually.
<b>Note:</b> If (*ptr) contains a pointer as a parameter, it is not stubbed automatically and with –<code>permissive-stubber</code>, the function pointer ptr is called with random as a parameter.</td></tr>
</table>

Function Pointer Cases

Function Prototype	Comments

`void _reg(int);`
`int _seq(void *);`
 	
`unsigned char bar(void){`
` return 0;`
`}`
 	
`void main(void){`
` unsigned char x=0;`
` _reg(_seq(bar));`
`}`
	Both functions, “_reg” and “_seq”, are automatically stubbed, but the Polyspace software does not exercise the call to the `bar` function.
	
	The function that is a parameter is only called in stubbed functions if the stubbed function prototype contains a function pointer as parameter.
	
	Because in this example, the stubbed function is a “void *”, it is not a function pointer.

Stub Functions with Variable Argument Number

Polyspace software can stub most `vararg` functions. However:

- This stubbing can generate imprecision in pointer verification.
- The stubbing causes a significant increase in complexity and in verification time.

There are three ways that you can deal with this stubbing issue:

- Stub manually
- On every `varargs` function that you know to be pure, add a `#pragma POLYSPACE_PURE "function_1"`. This action reduces greatly the complexity of pointer verification tenfold.

  For example:

  ```
  #pragma POLYSPACE_PURE f

  void main(void) {
    int x = 0;
    f(&x);
    assert ( x == 0 ); // Green assertion,
          //orange without use of #pragma POLYSPACE_PURE
  }
  ```
- Use `#define` to eliminate calls to functions. For example, functions like `printf` generate complexity but are not useful for verification because they only display a message.

  For example:

  ```
  #ifdef POLYSPACE
   #define example_of_function(format, args...)
  #else
   void example_of_function(char * format, ...)
  #endif
  void main(void)
  {
   int i = 3;
   example_of_function("test1 %d", i);
  }
  ```

  ```
  polyspace-code-prover-nodesktop -D POLYSPACE
  ```

You can place this kind of line in any `.c` or `.h` file of the verification.

Note: Use `#define` only with functions that are pure.

Prepare Code for Built-In Functions

In this section...

Overview

Polyspace software stubs functions that are not defined within the verification. Polyspace software provides an accurate stub for the functions defined in the `stl` and in the standard `libc`, taking into account functional aspects of the function.

Stubs of `stl` Functions

Functions of the `stl` are stubbed by Polyspace software. Using the `-no-stl-stubs` option allows deactivating standard stl stubs (not recommended for further possible scaling trouble).

Note: Allocation functions found in the code to analyze like new, new[], delete and delete[] are replaced by internal and optimized stubs of new and delete. A warning is given in the log file when such replace occurs.

Stubs of `libc` Functions

Functions are declared in the standard list of headers. You can redefine these functions by invalidating the associated set of functions and providing new definitions in your code.

To invalidate standard functions, use:

- `-D POLYSPACE_NO_STANDARD_STUBS` for functions declared in Standard ANSI headers: `assert.h`, `ctype.h`, `errno.h`, `locale.h`, `math.h`, `setjmp.h` (`setjmp` and `longjmp` functions are partially implemented — see *`Polyspace_Install`*`/polyspace/verifier/cxx/cinclude/__polyspace__stdstubs.c`), `signal.h` (`signal` and `raise` functions are partially implemented — see *`Polyspace_Install`*`/polyspace/verifier/cxx/cinclude/`

__polyspace__stdstubs.c), stdio.h, stdarg.h, stdlib.h, string.h, and time.h.

- -D POLYSPACE_STRICT_ANSI_STANDARD_STUBS for functions declared only in strings.h, unistd.h, and fcntl.h.

Note: You cannot redefine the following functions that deal with memory allocation: malloc(), calloc(), realloc(), valloc(), alloca(), __built_in_malloc(), and __built_in_alloca().

To invalidate a specific function, use -D __polyspace_no_*function_name*.

For example, if you want to redefine the fabs() function:

- For the verification, specify the option -D __polyspace_no_fabs.
- In the code, provide your fabs() function.

If your **Include** folders contain the standard header files stdio.h and string.h, Polyspace may recognize your function declarations even if they do not exactly match the standard declarations. For example, you might declare memset as:

```
void memset ( void * ptr, unsigned int value, size_t num );
```

instead of:

```
void * memset ( void * ptr, int value, size_t num );
```

In this case, a verification does not generate a compilation error. If your **Include** folders do not contain stdio.h and string.h, you can activate this Polyspace feature by specifying the option -D__polyspace_adapt_types_for_stubs. If your **Include** folders contain stdio.h and string.h but you want to deactivate the feature, specify the option -D__polyspace_static_types_for_stubs.

Note: If your function version differs from the standard function, the internal conversion of parameters and return type during verification may cause a loss of precision.

Verify Multitasking Applications

Your source code can contain functions that are intended to execute concurrently in separate threads (tasks). To verify if your variables are protected against concurrent access, you must specify your tasks and protection mechanisms.

1. Before you specify tasks to Polyspace, you must code them in a specific format. If your code is already written and does not follow the format, you can make minor adjustments to make your code suitable for multitasking verification.

   - The tasks must have the following prototype:

     ```
     void func(void)
     ```

   - The `main` function must not contain an infinite loop or a run-time error. Polyspace requires that before tasks begin, your `main` function has completed execution.
   - If your task executes indefinitely in cycles, it must contain an infinite loop.
   - If your task acts as an interrupt that can execute any number of times, it must contain a loop with unspecified number of runs. Use the following code in the task definition:

     ```
     volatile int randomValue = 0;
     while(randomValue) {
      /* Your task body goes here */
     }
     ```

   - If your interrupt occurs only after another task has executed a certain number of times, you can a create wrapper task to model this sequence.

2. On the **Configuration** pane, select **Multitasking**. Specify your tasks.

   For more information, see “Entry points (C/C++)”.

3. To protect variables from concurrent access, you must provide specific protection mechanisms in your code.

   - If you want two sections of code to execute without interruption from each other, you can enclose them in the same critical section. Place the two sections of code between calls to the same two functions.
   - If you want two tasks to execute without interruption from each other, you can specify them to be temporally exclusive.

4. On the **Configuration** pane, specify your protection mechanisms. For more information, see:

   - “Critical section details (C/C++)”
   - “Temporally exclusive tasks (C/C++)”

5. Run verification.

   After verification, the **Results Summary** pane lists potentially unprotected global variables.

   Determine which operations on the variable can occur concurrently. See “Review Global Variable Usage”.

See Also

Shared protected global variable | Shared unprotected global variable

Related Examples

- <title>Review Results</title>
  “Model Tasks” “Model Tasks if main Contains Infinite Loop” “Model Execution Sequence in Tasks”“Prevent Concurrent Access Using Critical Sections”“Prevent Concurrent Access Using Temporally Exclusive Tasks”

Model Tasks

This tutorial shows how to prepare for verification of multitasking code. If your code has functions that are intended for concurrent execution, you must specify them before verification.

For the high-level workflow on verifying multitasking applications, see “Verify Multitasking Applications”.

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

```
int a;

void performTaskCycle(void);

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

void interrupt(int val) {
 a=val;
}

void main() {
}
```

The code has two functions intended for concurrent execution.

- The function `task` must execute indefinitely.
- The function `interrupt` can execute any number of times.

In this example, you learn what happens when you:

1. Run verification without specifying entry points.
2. Specify entry points but do not modify your code.
3. Modify code appropriately so that Polyspace can accept your entry points.

In this section...

In this section...
"Run Multitasking Verification After Modifying Tasks" on page 6-31

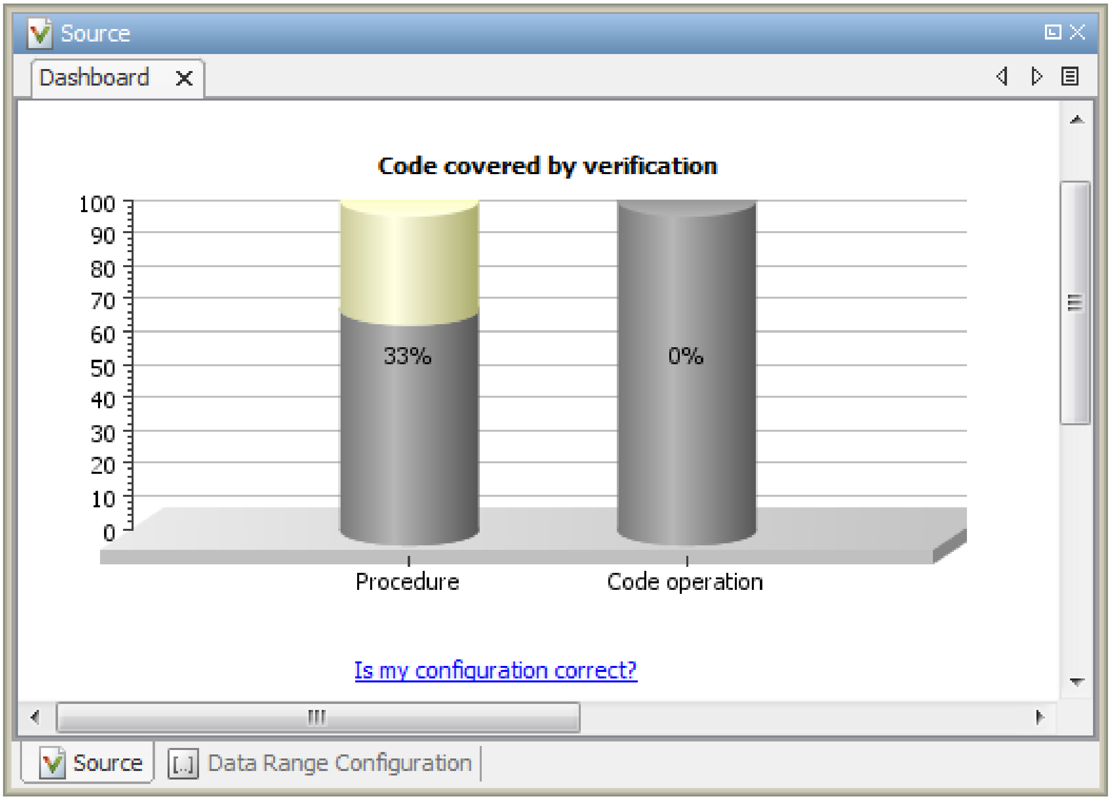
Run Non-Multitasking Verification

1 Create a Polyspace project. Add `multitasking_code.c` to the project.

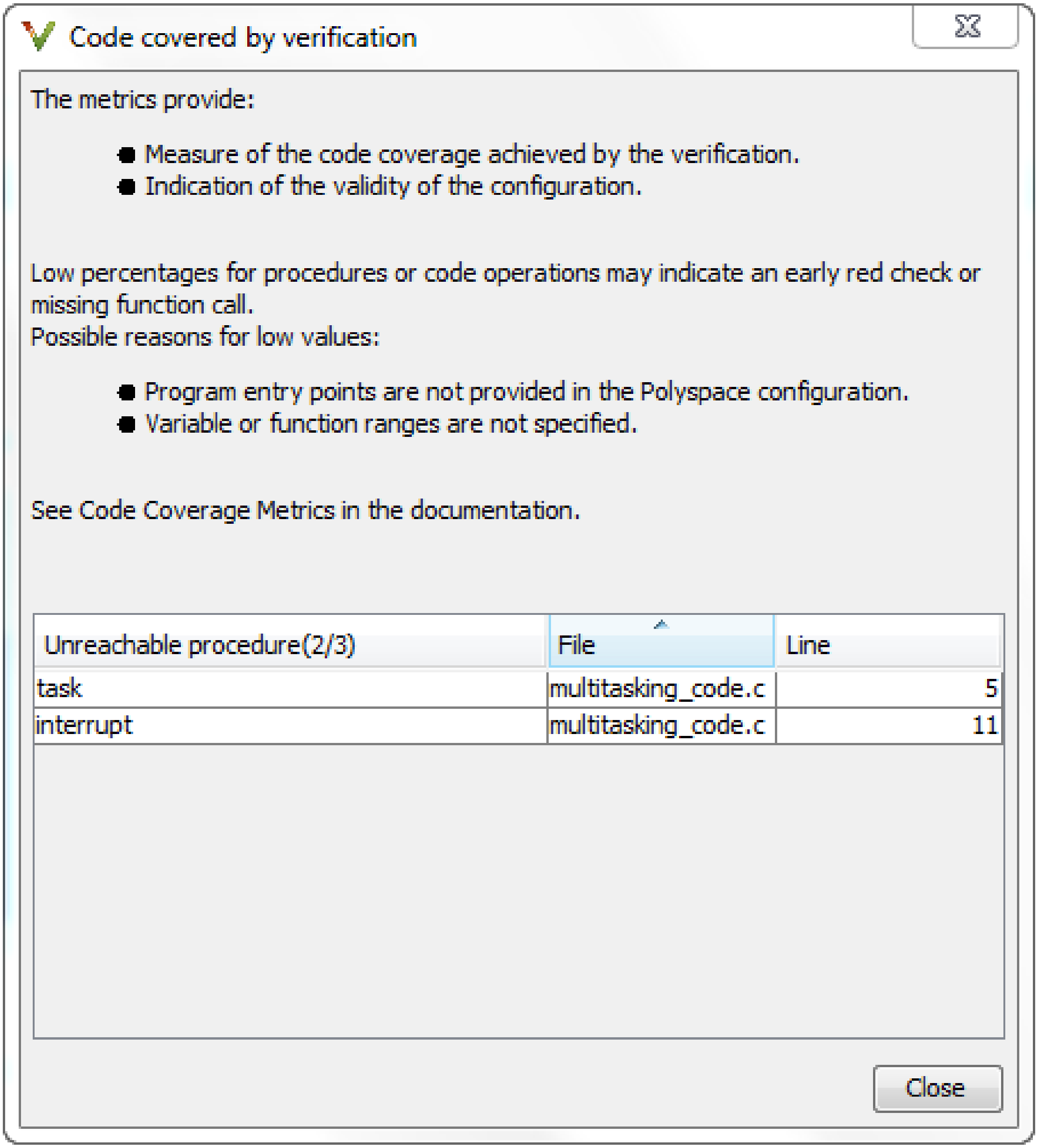
2 On the **Configuration** pane, select **Code Prover Verification**. Then select **Verify whole application**.

3 Run verification on your project. Open the results.

- On the **Source** pane **Dashboard**, you find that the verification covered only a third of the procedures. This information indicates that `task` and `interrupt` were not covered.



- If you click the **Code covered by verification** graph, you see `task` and `interrupt` listed as **Unreachable procedure**.



The metrics provide:

- Measure of the code coverage achieved by the verification.
- Indication of the validity of the configuration.

Low percentages for procedures or code operations may indicate an early red check or missing function call.
Possible reasons for low values:

- Program entry points are not provided in the Polyspace configuration.
- Variable or function ranges are not specified.

See Code Coverage Metrics in the documentation.

Unreachable procedure(2/3)	File	Line
task	multitasking_code.c	5
interrupt	multitasking_code.c	11

The verification did not cover `task` and `interrupt` because if you do not run a multitasking verification, `main` is the only entry point. In this case, the `main` did not call `task` and `interrupt`, so they are unreachable.

Run Multitasking Verification Without Modifying Tasks

1. On the **Configuration** pane, select **Mulitasking**. Select the **Multitasking** check box.
2. Specify `task` and `interrupt` as **Entry points**.

   a. Click the ✚ button to create a text field.

   b. In each field, enter one function name.

   The next verification recognizes that these functions are intended for concurrent execution.
3. Run verification again. You get the following compilation error:

   ```
   task 'interrupt' has non-void prototype
   ```
   This error appears because functions specified as entry points must have the prototype

   ```
   void func(void)
   ```
   If your entry point functions do not have this form, you must write a wrapper function to encapsulate them. In this example, `interrupt` takes an `int` argument. You must encapsulate it in a wrapper function.

Run Multitasking Verification After Modifying Tasks

1. Create a wrapper function `interrupt_handler` both argument and return type `void`. Call `interrupt` inside `interrupt_handler` with a `volatile int` argument. To do this:

   a. In a new file, enter the following code.

   ```
   void interrupt_handler(void) {
    volatile int input = 0;
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt(input);
   ```

```
  }
}
```

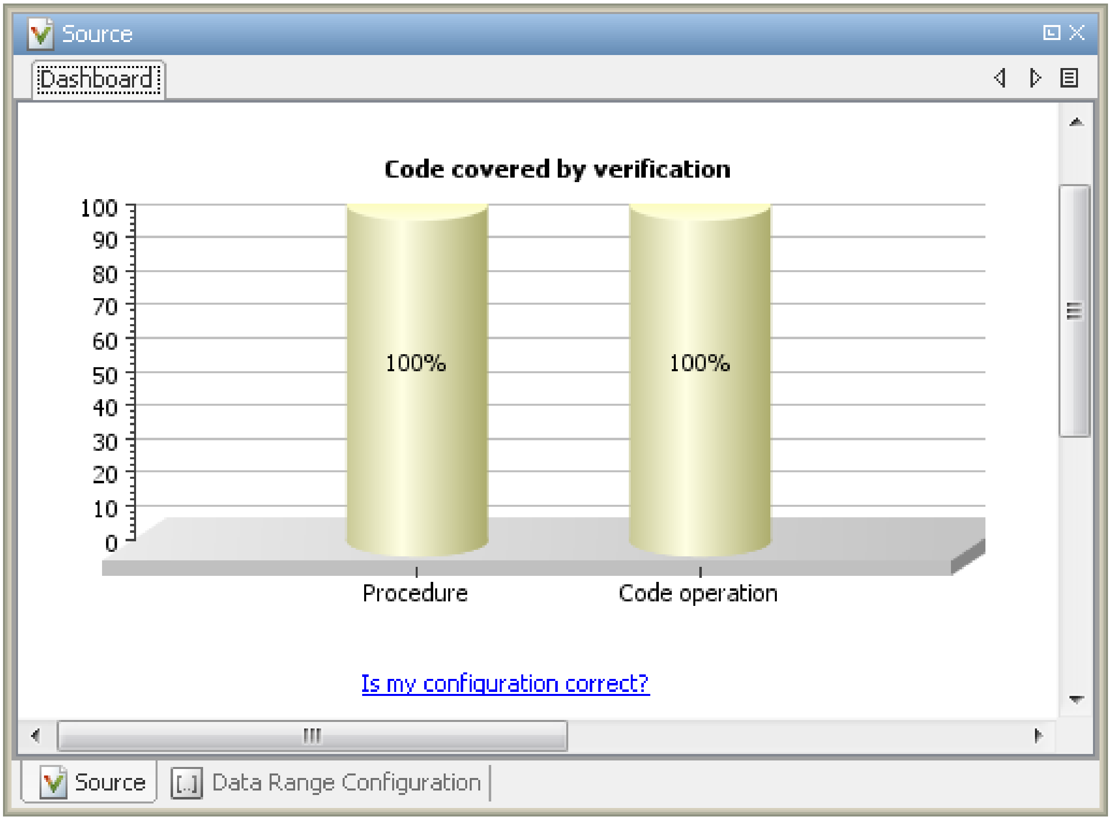
Polyspace considers that a `volatile int` variable can have any value allowed by its type at any time during execution. By defining `randomValue` as a `volatile int` variable, you specify that `interrupt` can run any number of times. Also, you initialize `randomValue` to zero to prevent an orange **Non-initialized local variable** check.

b. Add the new file to your Polyspace project. Copy it to the module on which you are running verification.

2. On the **Configuration** pane, select **Multitasking**. Replace the entry point `interrupt` by `interrupt_handler`.

3. Run verification again. Open the results.

   From the **Procedure** column on the **Source** pane **Dashboard**, you find that the verification covered all procedures.



Polyspace recognizes that:

- Your code is intended for multitasking.
- `task` and `interrupt_handler` are the entry points to your code.

See Also

Shared protected global variable | Shared unprotected global variable

Related Examples

- "Model Tasks if main Contains Infinite Loop"
- "Model Execution Sequence in Tasks"

- "Prevent Concurrent Access Using Temporally Exclusive Tasks"
- "Prevent Concurrent Access Using Critical Sections"
- "Review Global Variable Usage"

Model Tasks if main Contains Infinite Loop

This tutorial shows how to model tasks if your main function contains an infinite loop. Polyspace requires that before tasks begin, the main function has completed execution. If you want your main to run concurrently with the tasks instead of completing before them, your main function might already contain an infinite loop. If so, for precise multitasking verification using Polyspace, you must modify your code.

For the high-level workflow on verifying multitasking applications, see “Verify Multitasking Applications”.

For this example, use the following code:

```
void performTask1Cycle(void);
void performTask2Cycle(void);

void main() {
 while(1) {
    performTask1Cycle();
  }
}

void task2() {
 while(1) {
    performTask2Cycle();
  }
}
```

In this example, you learn what happens when you:

1 Specify entry points but retain an infinite loop in main.

2 Modify the main appropriately so that Polyspace can verify entry point functions.

In this section...

Run Multitasking Verification Without Modifying Code

1 Save the code in a file multi.c.

2 Create a Polyspace project and add `multi.c` to it.

3 On the **Configuration** pane, specify the following analysis options:

   a Select **Code Prover Verification** > **Verify whole application**.

   b Select **Multitasking** > **Multitasking**.

   c For **Multitasking** > **Entry Points**, specify `task2`. You do not have to specify `main` because Polyspace considers `main` as an entry point by default.

4 Run verification and open the results. On the **Results Summary** pane, you find a gray **Function not reachable** check on `task2`.

Polyspace treats `task2` as not reachable, even though you specified it as an entry point, because the `main` function contains an infinite loop.

Run Multitasking Verification After Modifying Code

1 Replace the following portion of the code

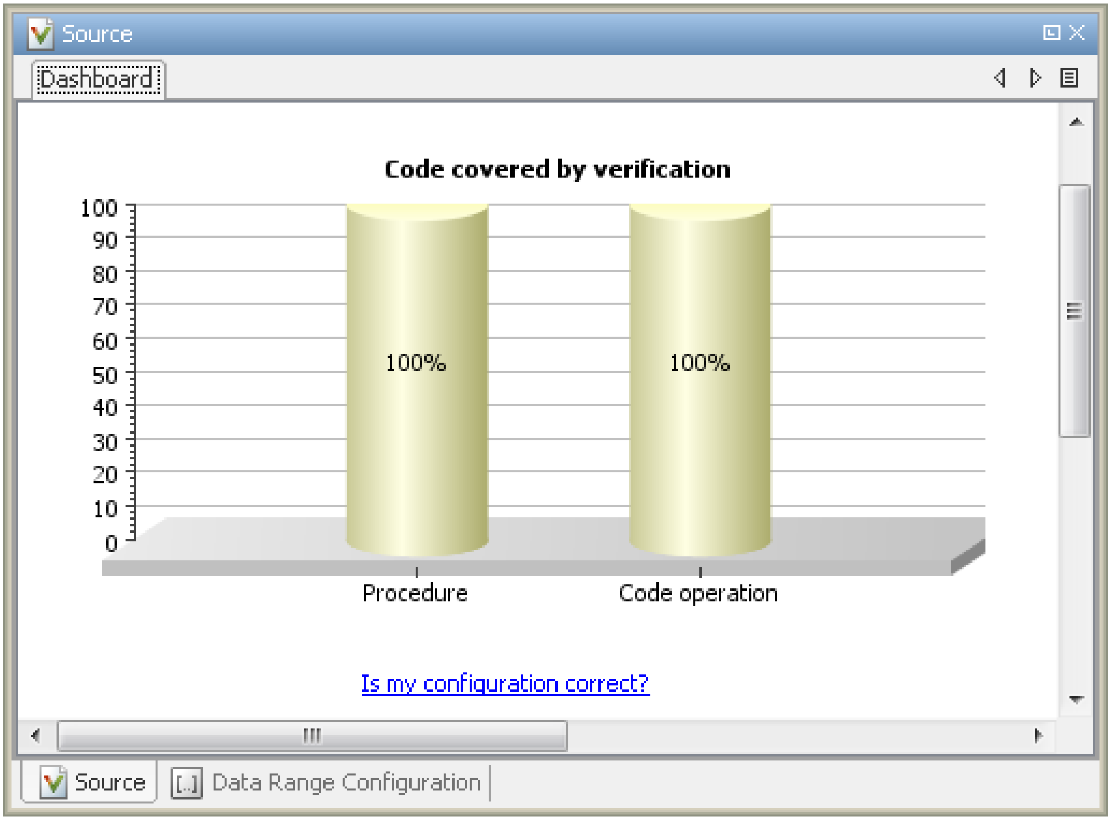
```
void main() {
 while(1) {
    performTask1Cycle();
  }
}
```

with

```
void main() {
}

void task1() {
 while(1) {
    performTask1Cycle();
  }
}
```

2 Run verification again. Open the results.

   From the **Procedure** column on the **Source** pane **Dashboard**, you find that the verification covered all procedures.



Polyspace verifies both `task1` and `task2` because the `main` function executes to completion.

See Also

Shared protected global variable | Shared unprotected global variable

Related Examples

- "Model Tasks"
- "Model Execution Sequence in Tasks"
- "Prevent Concurrent Access Using Temporally Exclusive Tasks"

- “Prevent Concurrent Access Using Critical Sections”
- “Review Global Variable Usage”

More About

- “Verify Multitasking Applications”

Model Execution Sequence in Tasks

This tutorial shows how to create a wrapper task for your functions so that they execute in a specific sequence in the task.

For the high-level workflow on verifying multitasking applications, see “Verify Multitasking Applications”.

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

```
int var;

void reset(void) {
 var=0;
}

void inc(void) {
  var+=2;
}


void task1(void) {
 volatile int randomValue = 0;
 while(randomValue) {
   inc();
  }
}

void task2(void) {
 volatile int randomValue = 0;
 while(randomValue) {
   reset();
  }
}

void main() {
}
```

In this example, you will learn what happens when you:

1. Specify entry points without modifying your code. The tasks execute in an arbitrary sequence and can interrupt each other any time.
2. Create a new entry point so that the tasks execute in a definite sequence.

3 Modify the new entry point so that each task in the sequence might or might not execute.

In this section...
"Specify Entry Points" on page 6-40
"Specify Definite Execution Sequence" on page 6-40
"Specify Indefinite Execution Sequence" on page 6-41

Specify Entry Points

1 Create a Polyspace project and add `multi.c` to it.

2 On the **Configuration** pane, specify the following analysis options:

   a Select **Code Prover Verification** > **Verify whole application**.

   b Select the **Multitasking** > **Multitasking** box.

   c For **Multitasking** > **Entry Points**, specify `task1` and `task2`, each on its own line.

3 Run verification and open the results.

   An orange **Overflow** error appears on the addition operator in `inc`. The error is not red because it does not occur along all execution paths. The error occurs only if `task1` executes sufficient number of times in succession without interruption from `task2`.

Specify Definite Execution Sequence

Suppose that you want to model that `reset` executes after `inc` has executed five times. This task sequence resets `var` after every five additions and prevents an overflow. To do this:

1 In a separate file `multi_sequence.c`, define a new wrapper function `task` as follows:

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   inc();
   inc();
   inc();
   inc();
   inc();
   reset();
   }
 }
```

2 Add `multi_sequence.c` to the project that you are running verification on.

3 On the **Configuration** pane, under **Multitasking**, do the following to the **Entry points** list:

   a Remove `task1` and `task2`.

   b Add `task`.

4 Run verification and open results.

   The orange **Overflow** error does not appear in `inc`. The **Overflow** check is green.

Specify Indefinite Execution Sequence

Suppose, you want to model that `reset` can execute after `inc` has executed zero to five times. This task sequence resets `var` after zero to five additions and also prevents an overflow. To do this:

1 In the file `multi_sequence.c`, modify `task` as follows:

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   reset();
   }
 }
```

   Because `randomValue` is a `volatile` variable, Polyspace considers that the execution can enter or skip any of the five `if` branches.

2 Run verification and open the results.

Again, the **Overflow** check on the addition in `inc` is green.

See Also

Shared protected global variable | Shared unprotected global variable

Related Examples

- “Model Tasks”
- “Model Tasks if main Contains Infinite Loop”
- “Prevent Concurrent Access Using Temporally Exclusive Tasks”
- “Prevent Concurrent Access Using Critical Sections”
- “Review Global Variable Usage”

Prevent Concurrent Access Using Temporally Exclusive Tasks

This tutorial shows how to protect shared variables from concurrent access. A shared variable is written or read by more than one task. Therefore, when the tasks accessing this variable execute concurrently, the variable value at a given time can be undetermined. To protect variables from concurrent access by multiple tasks:

- Specify that the tasks are temporally exclusive.
- If you do not want to specify the tasks as temporally exclusive, place read or write access to those variables inside critical sections.

This example shows the first approach.

For the high-level workflow on verifying multitasking applications, see “Verify Multitasking Applications”.

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

```
#include <limits.h>
int shared_var;

void inc() {
 shared_var+=2;
}

void reset() {
 shared_var = 0;
}

void task() {
  volatile int randomValue = 0;
  while(randomValue) {
    reset();
    inc();
    inc();
  }
}

void interrupt() {
 shared_var = INT_MAX;
}

void interrupt_handler() {
  volatile int randomValue = 0;
  while(randomValue) {
   interrupt();
  }
}

 void main() {
}
```

In this example, you will learn what happens when you:

1 Specify entry points and run verification. Your tasks can interrupt each other any time.
2 Run verification after specifying temporally exclusive tasks.

In this section...
"View Unprotected Access in Polyspace Results" on page 6-44
"Specify Temporally Exclusive Tasks" on page 6-45

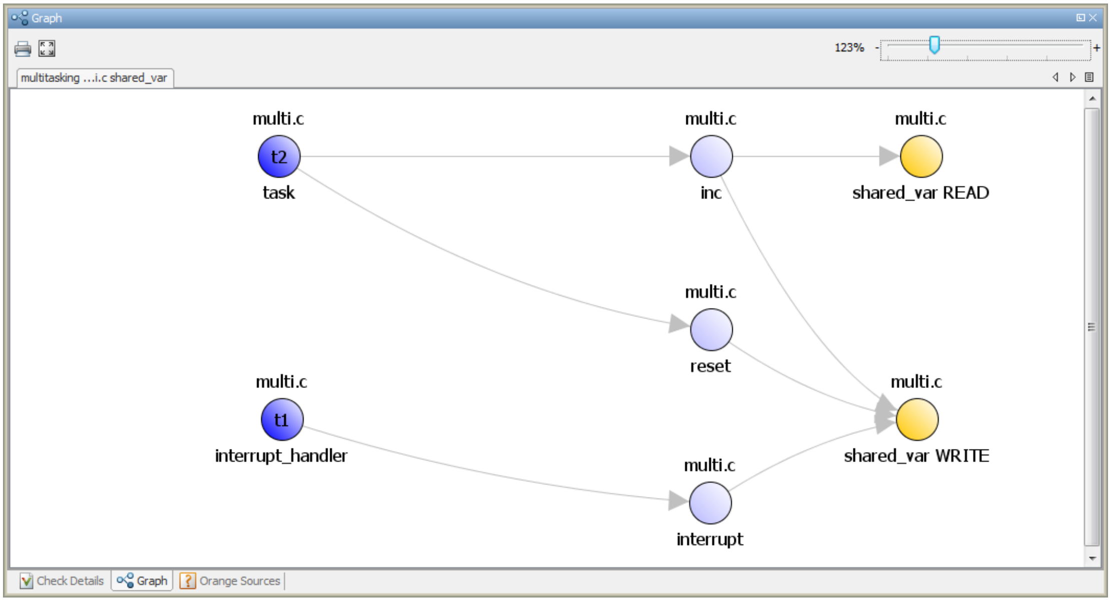
View Unprotected Access in Polyspace Results

1 Create a Polyspace project and add `multi.c` to it.
2 On the **Configuration** pane, specify the following analysis options:
   a Select **Code Prover Verification** > **Verify whole application**.
   b Select the **Multitasking** > **Multitasking** check box.
   c For **Multitasking** > **Entry Points**, specify `task` and `interrupt_handler`, each on its own line.
3 Run verification and view the results.

   On the **Results Summary** pane, select **Group by** > **Family**. Under the node **Shared** > **Potentially unprotected variable**, you see the variable `shared_var`.

   The global variable `shared_var` is not protected from concurrent access by tasks `task` and `interrupt_handler`.
4 Select the result.
   - On the **Check Details** pane, see all read and write operations on the variable `shared_var`.

- On the **Check Details** pane, if you click the button, you see a graphical representation of the operations on the variable.



Specify Temporally Exclusive Tasks

You can protect `shared_var` from concurrent access by making `task` and `interrupt_handler` temporally exclusive tasks.

1. On the **Configuration** pane:

   - Retain the analysis options from the previous verification.
   - For the option **Multitasking** > **Temporally exclusive tasks**, click the ✚ button. Enter `task interrupt_handler`.

2. Run verification and view the results.

   On the **Results Summary** pane, select **Group by** > **Family**. Under the node **Global Variable** > **Shared** > **Protected variable**, you see the variable `shared_var`.

Polyspace Code Prover has proved the protection of `shared_var` from concurrent access.

3 On the **Results Summary** pane, there is an orange **Overflow** error.

   a Select this error.

   b On the **Source** pane, place your cursor on the orange plus sign.

You see that the left operand can be 2^{31}`-1`.

Although Polyspace proves that `shared_var` is protected from concurrent access by `task` and `interrupt_handler`, it does not take this fact into account during verification. Therefore, it considers that an **Overflow** can occur if:

a Inside `task`, `reset` executes and assigns 0 to `shared_var`.

b `interrupt_handler` executes and assigns `INT_MAX` or 2^{31}`-1` to `shared_var`.

c Inside `task`, `inc` executes and adds 2 to `INT_MAX` causing the overflow.

Related Examples

- "Prevent Concurrent Access Using Critical Sections"
- "Model Tasks"
- "Model Tasks if main Contains Infinite Loop"
- "Model Execution Sequence in Tasks"
- "Review Global Variable Usage"

More About

- Shared protected global variable
- Shared unprotected global variable

Prevent Concurrent Access Using Critical Sections

This tutorial shows how to protect shared variables from concurrent access. A shared variable is written or read by more than one task. Therefore, when the tasks accessing this variable execute concurrently, the variable value at a given time can be undetermined. To protect variables from concurrent access by multiple tasks:

- Specify that the tasks are temporally exclusive.
- If you do not want to specify the tasks as temporally exclusive, place read or write access to those variables inside critical sections.

This example shows the second approach.

For the high-level workflow on verifying multitasking applications, see “Verify Multitasking Applications”.

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

```
#include <limits.h>
int shared_var;

void inc() {
 shared_var+=2;
}

void reset() {
 shared_var = 0;
}

void task() {
  volatile int randomValue = 0;
  while(randomValue) {
    reset();
    inc();
    inc();
  }
}

void interrupt() {
 shared_var = INT_MAX;
}

void interrupt_handler() {
  volatile int randomValue = 0;
  while(randomValue) {
   interrupt();
  }
}

 void main() {
}
```

In this example, you will learn what happens when you:

1. Specify entry points and run verification. Your tasks can interrupt each other any time.
2. Protect two sections of code from interruption by each other using a critical section. To implement the critical section, place the two sections of code between calls to the same two functions.

In this section...
"View Unprotected Access in Polyspace Results" on page 6-48
"Specify Critical Sections" on page 6-49

View Unprotected Access in Polyspace Results

1. Create a Polyspace project and add `multi.c` to it.
2. On the **Configuration** pane, specify the following analysis options:

   a. Select **Code Prover Verification** > **Verify whole application**.
   b. Select the **Multitasking** > **Multitasking** check box.
   c. For **Multitasking** > **Entry Points**, specify `task` and `interrupt_handler`, each on its own line.

3. Run verification and view the results.

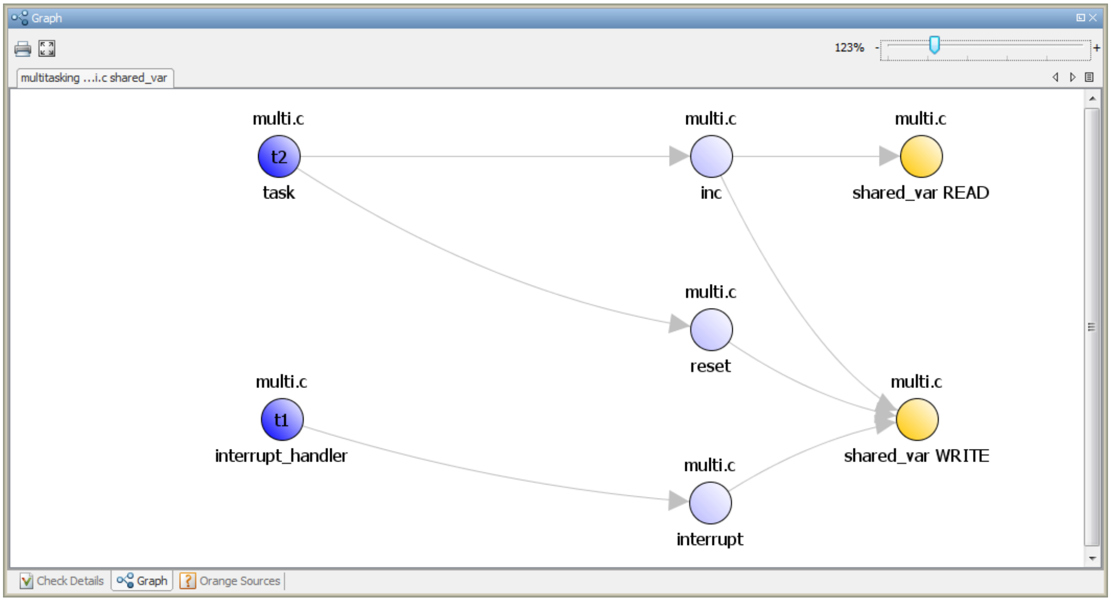
   On the **Results Summary** pane, select **Group by** > **Family**. Under the node **Shared** > **Potentially unprotected variable**, you see the variable `shared_var`.

   The global variable `shared_var` is not protected from concurrent access by tasks `task` and `interrupt_handler`.

4. Select the result.

   - On the **Check Details** pane, see all read and write operations on the variable `shared_var`.

- On the **Check Details** pane, if you click the button, you see a graphical representation of the operations on the variable.



Specify Critical Sections

You can protect `shared_var` from concurrent access by placing the accesses inside a critical section.

1 Save the following code in another file `multi_critical_section.c`.

```
#include <limits.h>
int shared_var;

void inc() {
 shared_var+=2;
}

void reset() {
 shared_var = 0;
}
```

```
void take_semaphore(void);
void give_semaphore(void);

void task() {
  volatile int randomValue = 0;
  while(randomValue) {
    take_semaphore();
    reset();
    inc();
    inc();
    give_semaphore();
  }
}

void interrupt() {
 shared_var = INT_MAX;
}

void interrupt_handler() {
  volatile int randomValue = 0;
  while(randomValue) {
   take_semaphore();
   interrupt();
   give_semaphore();
  }
}

void main() {
}
```

The differences between `multi.c` and `multi_critical_section.c` are:

- There are two new functions `take_semaphore()` and `give_semaphore()` in `multi_critical_section.c` with the prototype:

  ```
  void func_name(void);
  ```

- The cycle code in functions `task()` and `interrupt_handler()` is between calls to `take_semaphore()` and `give_semaphore()`.

2. Add `multi_critical_section.c` to your project. Create a new module in your project and copy the file to that module.
3. On the **Configuration** pane:

- Retain the analysis options from the previous verification.
- For the option **Multitasking** > **Critical section details**, specify `take_semaphore` as the **Starting procedure** and `give_semaphore` as the **Ending procedure**.

4 Run verification and view the results.

On the **Results Summary** pane, select **Group by** > **Family**. Under the node **Global Variable** > **Shared** > **Protected variable**, you see the variable `shared_var`.

Polyspace Code Prover has proved the protection of `shared_var` from concurrent access.

5 On the **Results Summary** pane, there is still an orange **Overflow** error.

a Select this error.

b On the **Source** pane, place your cursor on the orange + sign.

You see that the left operand can be $2^{31}-1$.

Although Polyspace proves that `shared_var` is protected from concurrent access by `task` and `interrupt_handler`, it does not take this fact into account during verification. Therefore, it considers that an **Overflow** can occur if:

a Inside `task`, `reset` executes and assigns 0 to `shared_var`.

b `interrupt_handler` executes and assigns `INT_MAX` or $2^{31}-1$ to `shared_var`.

c Inside `task`, `inc` executes and adds 2 to `INT_MAX` causing the overflow.

Related Examples

- "Prevent Concurrent Access Using Temporally Exclusive Tasks"
- "Model Tasks"
- "Model Tasks if main Contains Infinite Loop"
- "Model Execution Sequence in Tasks"
- "Review Global Variable Usage"

More About

- Shared protected global variable
- Shared unprotected global variable

Exclude Assembly Code if Compiler Generates Errors

Polyspace ignores most assembly code during verification.

If Polyspace cannot parse assembly code during the **Compile** phase, use the command line options `-asm-begin` and `-asm-end` to indicate assembly code sections.

Consider the following code.

```
int x=12;

void f(void)
{
#pragma will_be_ignored
 x =0;
 x= 1/x;           // no color is displayed
                        // not even C code
#pragma was_ignored
 x++;
 x=15;
}

void main (void)
{
 int y;
 f();
 y = 1/x + 1 / (x-15);  // Red ZDV, x is equal to 15

}
```

The verification ignores text or code placed between the two `#pragma` statements if you specify the following options:

```
-asm-begin will_be_ignored -asm-end was_ignored
```

This approach allows an unsupported assembly code section to be ignored without changing the meaning of the original code.

See Also

-asm-begin -asm-end

More About

- "Assembly Code"

Running a Verification

Specify Results Folder

This example shows how to specify a results folder. In the **Project Browser** pane, the folder appears as a node under the **Result** node of your project. By default, the software creates a new results folder for each analysis. Before starting an analysis, you can choose to overwrite an existing results folder. For example, if you stopped an analysis before completion and want to restart it, you can overwrite a results folder.

- To create a new folder, on the **Project Browser** pane, select **Create new result folder**.
  - By default, the new folder is created in *`Project_folder`* / *`Module_name`*. *`Project_folder`* is the project location you specified when creating a new project.
  - You can also create a parent folder for storing your results. Select **Tools** > **Preferences** and enter the parent folder location on the **Project and Results Folder** tab. If you enter a parent folder location, any new result folder will be created under this parent folder.
- To overwrite an existing folder that is open in the **Project Browser** pane, clear **Create new result folder**. Before running verification, select the result that you want to overwrite.
- To store results in a folder that is not open in the **Project Browser**, right-click the **Result** node. Select **Choose a Result folder**. Select the folder where you want your results stored.

When you start the verification, the software saves the results in the specified folder.

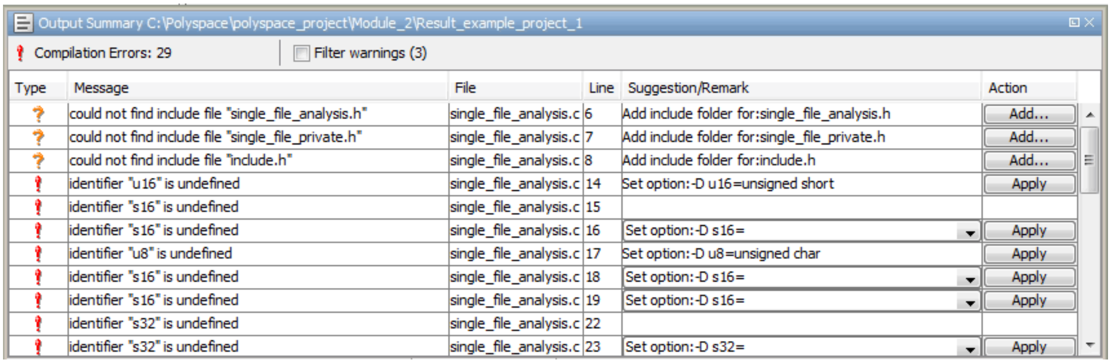
Check for Compilation Issues

During a verification, if the Compilation Assistant detects compilation errors, the verification stops and the software displays errors and possible solutions on the **Output Summary**.

To check your project for compilation problems:

1. Select **Tools** > **Preferences**.
2. In the Polyspace Preferences dialog box, click the **Project and Results Folder** tab.
3. Select **Use Compilation Assistant**.
4. On the toolbar, click the ▷ Run button.

   The software compiles your code and checks for errors, and reports the results on the **Output Summary** tab.
5. Select a **Suggestion/Remark** cell to see a list of possible solutions for the problem.



Output Summary C:\Polyspace\polyspace_project\Module_2\Result_example_project_1

! Compilation Errors: 29 □ Filter warnings (3)

Type	Message	File	Line	Suggestion/Remark	Action
?	could not find include file "single_file_analysis.h"	single_file_analysis.c	6	Add include folder for:single_file_analysis.h	Add...
?	could not find include file "single_file_private.h"	single_file_analysis.c	7	Add include folder for:single_file_private.h	Add...
?	could not find include file "include.h"	single_file_analysis.c	8	Add include folder for:include.h	Add...
!	identifier "u16" is undefined	single_file_analysis.c	14	Set option:-D u16=unsigned short	Apply
!	identifier "s16" is undefined	single_file_analysis.c	15		
!	identifier "s16" is undefined	single_file_analysis.c	16	Set option:-D s16= ▾	Apply
!	identifier "u8" is undefined	single_file_analysis.c	17	Set option:-D u8=unsigned char	Apply
!	identifier "s16" is undefined	single_file_analysis.c	18	Set option:-D s16= ▾	Apply
!	identifier "s16" is undefined	single_file_analysis.c	19	Set option:-D s16= ▾	Apply
!	identifier "s32" is undefined	single_file_analysis.c	22		
!	identifier "s32" is undefined	single_file_analysis.c	23	Set option:-D s32= ▾	Apply

In this example, you can either add the missing include files, or set options to compile the code without the missing include files:

- Select **Apply** to set the selected option for your project. The software automatically sets the option.
- Select **Add** to add suggested include folders to your project. The Add Source Files and Include Folders dialog box opens, allowing you to add additional include folders.

When you have addressed compilation problems, run the verification again.

The Compilation Assistant is automatically disabled if you specify one of the following options:

- `-unit-by-unit`
- `-post-preprocessing-command`

Run Local Verification

Before running verification on your source files, you must add them to a Polyspace project. For more information, see “Create Project”.

Start Verification

To start a verification on your local desktop:

1 On the **Project Browser** pane, select the project module that you want to verify.

2 On the toolbar, click the ▷ Run button.

Tip To run verification on all modules in the project, expand the drop-down list beside the ▷ Run. Select **Run All Modules**.

Monitor Progress

To monitor the progress of a local verification, use the following panes. If you have closed a pane, to open it again, select **Window > Show/Hide View**.

- **Output Summary** — Displays progress of verification, compile phase messages and errors.
- **Run Log** — This tab displays messages, errors, and statistics for all phases of the verification.

Tip To search for a term in the **Output Summary** or **Run Log**, enter the term on the **Search** pane. Select **Output Summary** or **Run Log** from the drop-down list beside the search box.

If the **Search** pane is not open by default, select **Windows** > **Show/Hide View** > **Search**.

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

Stop Verification

To stop a local verification:

1 On the toolbar, click the **Stop** button.

  A warning dialog box opens asking whether you want to stop the execution.

2 Click **Yes**. The verification stops, and results are incomplete. If you start another verification, the verification starts from the beginning.

Open Results

After verification, the results open automatically on the **Results Summary** pane. If you are looking at previous results when a verification is over, you can load the new results or retain the previous results on the **Results Summary** pane.

To open the new results later:

1 On the **Project Browser** pane, navigate to the results set that you want to review.

2 Double-click the results set, for example, **Result_1**.

  The software loads the verification results in the **Results Summary** pane.

To open results of verification when the corresponding project is not open in the **Project Browser** pane:

1 Select **File** > **Open**.

2 In the Open File dialog box, navigate to the results folder. For example:

```
My_project\Module_1\Result_1
```

3 Select the results file, for example, `My_project.pscp`.

4 Click **Open**.

Related Examples

- “Run File-by-File Local Verification”
- “Run Remote Verification”

More About

- “Phases of Verification”
- “Results Folder Contents”

Run Remote Verification

Run remote verification when:

- You want to shut down your local machine but not interrupt the verification.
- You want to free execution time on your local machine.
- You want to transfer verification to a more powerful computer.

Before you run remote verification, you must do the following:

- Set up a server for this purpose. For more information, see “Set Up Server for Remote Verification and Analysis”.
- Add your source files to a Polyspace project. For more information, see “Create Project”.

In this section...

Start Verification

To start a remote verification:

1. On the **Project Browser** pane, select the module you want to verify.
2. On the **Configuration** pane, select **Distributed Computing**. Select **Batch**.
3. Optionally, select **Add to results repository**.

   After verification, your results are uploaded to the Polyspace Metrics web dashboard.
4. On the toolbar, click the ▷ Run button.

   On the local host computer, the Polyspace Code Prover software performs code compilation and coding rule checking . Then the Parallel Computing Toolbox™ software submits the verification to the MATLAB job scheduler (MJS) on the head node of the MATLAB Distributed Computing Server™ cluster. For more information, see “Phases of Verification” on page 7-11.

Note: If you see the message `Verification process failed`, click **OK**. For more information on errors related to remote verification, see “Polyspace Cannot Find the Server”.

Monitor Progress

You can manage your verification through the Polyspace Job Monitor.

1. Select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, right-click your verification.
3. From the context menu, select your management task:
   - **View Log File** — Open the verification log.
   - **Download Results** — Download verification results from remote computer if the verification is complete.

Stop Verification

1. Select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, right-click your verification. From the context menu, select **Remove From Queue**.

Open Results

Your results are downloaded automatically after verification. To open them:

1. On the **Project Browser** pane, navigate to the results set.
2. Double-click the results set, for example, **Result_1**.

   The software loads the verification results in the **Results Summary** pane.

Note: If you select the option **Add to results repository**, your results are not downloaded automatically after verification. Use the Polyspace Metrics web dashboard to view the results and download them to your desktop. For more information, see “View Code Quality Metrics”.

Related Examples

- “Run File-by-File Remote Verification”
- “Run Local Verification”

More About

- “Phases of Verification”
- “Results Folder Contents”

Phases of Verification

A verification has three main phases:

1. Checking syntax and semantics (the compile phase). Because Polyspace software is compiler-independent, it helps you to produce code that is portable, maintainable, and compliant with ANSI standards.
2. Generating a main if the Polyspace software does not find a main and you have selected the **Verify module** option. For more information about generating a main, see:
   - "Verify module (C)" — C verification
   - "Verify module (C++)" — C++ verification
3. Analyzing the code for run-time errors and generating color-coded results.

Run File-by-File Local Verification

This example shows how to run a local verification on each file independently of other files in the module.

Before running verification on your source files, you must add them to a Polyspace project. For more information, see “Create Project”.

In this section...

Run Verification

1 On the **Configuration** pane, specify that each file must be verified independently of other files.

   a Select the **Code Prover Verification** node.

   b Select **Verify files independently**.

   c For **Common source files**, enter files that you want to include with verification of each file. Enter the full path to a file. Enter one file path per row.

   For example, if multiple files use a function, you must include the file containing the function definition as a common source file. Otherwise, Polyspace stubs the undefined functions leading to more orange checks.

2 On the toolbar, click the ▷ Run button.

   On the **Output Summary** pane, you can see that after the **Compile** phase, each file is verified independently. After the verification is complete for a file, you can view the results while other files are still being verified.

Open Results

After verification, your results appear in the **Project Browser** under the **Result** node of your module.

1 To open result for each source file, double-click the corresponding result file under the **Result** node. The result file has the same name as the source file.

2 To see an overview of the verification:

- **a** Under the **Result** node, right-click the 📁 icon.
- **b** Select **Open Folder with File Manager**.

  Your result folder opens in your file explorer.
- **c** Open the `html` file `Synthesis` in the result folder.

See Also

"Verify files independently (C/C++)" | "Common source files (C/C++)"

Related Examples

- "Run File-by-File Remote Verification"

Run File-by-File Remote Verification

This example shows how to run a remote verification on each file independently of other files in the module.

Before you run remote verification, you must do the following:

- Set up a server for this purpose. For more information, see “Set Up Server for Remote Verification and Analysis”.
- Add your source files to a Polyspace project. For more information, see “Create Project”.

In this section...

Run Verification

1 On the **Configuration** pane, specify remote verification.

a Select the **Distributed Computing** node.

b Select **Batch**.

c Optionally, select **Add to results repository**.

After verification, your results are uploaded to the Polyspace Metrics web dashboard.

2 On the **Configuration** pane, specify that each file must be verified independently of other files.

a Select the **Code Prover Verification** node.

b Select **Verify files independently**.

c For **Common source files**, enter files that you want to include with verification of each file. Enter the full path to a file. Enter one file path per row.

For example, if multiple files use a function, you must include the file containing the function definition as a common source file. Otherwise, Polyspace stubs the undefined functions leading to more orange checks.

3. On the toolbar, click the ▷ Run button.

   The Parallel Computing Toolbox software submits the verification units as separate jobs to your scheduler. The scheduler is on the head node of the MATLAB Distributed Computing Server cluster.

   After the **Compile** phase, you can view the jobs in the Polyspace Job Monitor.
4. Select **Tools** > **Open Job Monitor**.

   Your files appear as child nodes under the main verification node. After the verification is complete for a file, you can download and view the results while other files are still being verified. Right-click the row corresponding to the file and select **Download Results**.

Open Results

Your results are automatically downloaded after verification.

To open result for each source file, double-click the corresponding result file under the **Result** node. The result file has the same name as the source file.

Note: If you select the option **Add to results repository**, your results are not downloaded automatically after verification. Use the Polyspace Metrics web dashboard to view the results and download them to your desktop. For more information, see “View Code Quality Metrics”.

See Also

“Batch (C/C++)” | “Verify files independently (C/C++)” | “Common source files (C/C++)”

Related Examples

- “Run File-by-File Local Verification”

Create Project Automatically at Command Line

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

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

Use the `polyspace-configure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

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

  For the list of options allowed with the GNU `make`, see make options.
- Create an options file. You can then use the options file to run verification on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspace-configure -no-project -output-options-file myOptions ...
            make targetName buildOptions
  ```
  Use the options file to run verification:

  ```
  polyspace-code-prover-nodesktop -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspace-configure`. For more information, see the `-options value` argument for `polyspaceConfigure`.

Run Local Verification at Command Line

In this section...
"Specify Sources and Analysis Options Directly" on page 7-17
"Specify Sources and Analysis Options in Text File" on page 7-17

Specify Sources and Analysis Options Directly

At the Windows, Linux or Mac OS X command-line, append sources and analysis options to the `polyspace-code-prover-nodesktop` command.

For instance:

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, use the command:

  ```
  polyspace-code-prover-nodesktop -sources "file.c" -lang c -target m68k
  ```
- To specify verification precision, use the `-O` option. For instance, to set precision level to 2 for your source file `file.c`, use the command:

  ```
  polyspace-code-prover-nodesktop -sources "file.c" -lang c -O2
  ```

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

You can also enter the following at the command line:

```
polyspace-code-prover-nodesktop -help
```

Specify Sources and Analysis Options in Text File

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

   ```
   #These are the options for MyCodeProverProject
   -lang c
   -prog MyCodeProverProject
   -author jsmith
   -sources "mymain.c,funAlgebra.c,funGeometry.c"
   -OS-target no-predefined-OS
   -target x86_64
   -dialect none
   ```

```
-dos
-misra2 required-rules
-includes-to-ignore all-headers
-main-generator
-results-dir C:\Polyspace\MyCodeProverProject
```

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

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

More About

- “Scripts for Command-Line Verification”

Run Remote Analysis at Command Line

Before you run a remote analysis, you must set up a server for this purpose. For more information, see “Set Up Server for Remote Verification and Analysis”.

In this section...
“Run Remote Analysis” on page 7-19
“Manage Remote Analysis” on page 7-20
“Download Results” on page 7-22

Run Remote Analysis

Use the following command to run a remote verification:

```
MATLAB_Install\polyspace\bin\polyspace-code-prover-nodesktop
-batch -scheduler NodeHost | MJSName@NodeHost [options]
```

where:

- `MATLAB_Install` is your MATLAB installation folder.
- `NodeHost` is the name of the computer that hosts the head node of your MDCS cluster.
- `MJSName` is the name of the MATLAB Job Scheduler (MJS) on the head node host.
- `options` are the analysis options. These options are the same as that of a local analysis. For more information, see “Run Local Verification at Command Line”.

After compilation, the software submits the verification job to the cluster and provides you a job ID. Use the `polyspace-jobs-manager` command with the job ID to monitor your verification and download results after verification is complete. For more information, see:

- “Manage Remote Analysis” on page 7-20
- “Download Results” on page 7-22

Tip In Windows, to avoid typing the commands each time, you can save the commands in a batch file.

1. Save your analysis options in a file `listofoptions.txt`. See "Specify Sources and Analysis Options in Text File".
   To specify your sources, in the options file, instead of `-sources`, use -sources-list-file. This option is available only for remote analysis and allows you to specify your sources in a separate text file.
2. Create a file `launcher.bat` in a text editor like Notepad.
3. Enter the following commands in the file.
   ```
   echo off
   set POLYSPACE_PATH=C:\Program Files\MATLAB\R2015a\polyspace\bin
   set RESULTS_PATH=C:\Results
   set OPTIONS_FILE=C:\Options\listofoptions.txt
   "%POLYSPACE_PATH%\polyspace-code-prover-nodesktop.exe" -batch -scheduler localhost
                             -results-dir %RESULTS_PATH% -options-file %OPTIONS_FILE%
   pause
   ```
4. Replace the definitions of the following variables in the file:
   - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
   - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
   - `OPTIONS_FILE`: Enter the path to the file `listofoptions.txt`.

   Replace `localhost` with the name of the computer that hosts the head node of your MDCS cluster.
5. Double-click `launcher.bat` to run the verification.

If you run a Polyspace verification, a `.bat` file is automatically generated for you. You can relaunch verification using this file.

Manage Remote Analysis

To manage remote analyses, use this command:

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

where:

- *`MATLAB_Install`* is your MATLAB installation folder
- `schedulerOption` is one of the following:

  - Name of the computer that hosts the head node of your MDCS cluster (*NodeHost*).
  - Name of the MJS on the head node host (*MJSName@NodeHost*).
  - Name of a MATLAB cluster profile (*ClusterProfile*).

    For more information about clusters, see “Clusters and Cluster Profiles”

  If you do not specify a job scheduler, `polyspace-job-manager` uses the scheduler specified in the **Polyspace Preferences** > **Server Configuration** > **Job scheduler host name**.

- *action [options]* refer to the possible action commands to manage jobs on the scheduler:

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

Action	Options	Task
	`-job` *`ID`* `-results-folder` *`FolderPath`*	Download results of analysis with specified ID to folder specified by *`FolderPath`*.
`getlog`	`-job` *`ID`*	Open log for job with specified ID.
`remove`	`-job` *`ID`*	Remove job with specified ID.

Download Results

To download verification results from the command line, use the `polyspace-jobs-manager` command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager -download
-job Verification_ID -results-folder FolderPath
```

After downloading results, use the Polyspace user interface to view the results. See “Open Results”.

Modularize Application at Command Line

You can partition large applications into modules. For more information, see “Modularize Project Automatically”.

In this section...

Basic Options

You can partition an application into modules using the following batch command:

```
polyspace-modularize [target_folder] {options}
```

This table describes the basic options that you can use.

Option	Description
target_folder	Folder that contains the results of the initial run that processes source files. Default is the folder from which you run `polyspace-modularize`.
-o *output_folder*	Output folder for partitioned application. Default is the folder from which you run `polyspace-modularize`.
-gui *max_n*	The Polyspace verification environment displays the Modularizing choices window with a predefined limit for the maximum number of modules that you can select. Use this option to specify a new limit *max_n*.
-matlab *max_n*	If data cache for Modularizing choices window does not exist, create cache *project_name_max_n*.m. Cache enables faster display of Modularizing choices window. *project_name* is the value used by `-prog` option. *max_n* is the limit for the maximum number of modules that you can select.

Option	Description
	No action if cache already exists.
`-modules` *`n`*	Partition application into *`n`* modules. Identical to clicking the gray region associated with *`n`* in the Modularizing choices window.
`-max-complexity` *`max_c`*	Partitions application into modules with reference to specified maximum complexity *`max_c`*. The complexity of a function is a number that is related to the size of the function. The complexity of a module is the sum of the complexities of the functions in the module. When partitioning your application, the software minimizes the use of cross-module references to functions and variables, subject to the constraint that the complexity of a module does not exceed *`max_c`*. If you make *`max_c`* sufficiently large, the software generates only one module, which is identical to the original, unpartitioned application.

Constrain Module Complexity During Partitioning

To force all functions to have a complexity of 1, run the following command:

```
polyspace-modularize -uniform-complexities
```

Result Folder Names

By default, modularization results folders are named *`projectName_kk_module`*:

- *`kk`* is either the max complexity argument you give to `-max-complexity`, or the number of modules..

You can control the naming of result folders in the *`i`*th module using the `-stem` option:

```
polyspace-modularize -stem stem_format
polyspace-modularize -stem MyName
```

`stem_format` is a string. The `#` and `@` characters in the string are processed as follows:

- `#` — Replaced by the number of modules in the partitioning.

- @ — Replaced by the argument of -max-complexity.

For example, if you want a specific name, MyName, which overrides the project name and does not incorporate the module number, then run:

Forbid Cycles in Module Dependence Graph

By default, the software allows the module dependence graph to have cycles. However, in some cases, you might get better results with acyclic graphs. Use the following command:

```
polyspace-modularize -forbid-cycles
```

Create Project Automatically from MATLAB Command Line

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

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

Use the `polyspaceConfigure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspaceConfigure  -prog myProject ...
               make targetName buildOptions
  ```
- Create an options file. You can then use the options file to run verification on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspaceConfigure  -no-project -output-options-file myOptions ...
             make targetName buildOptions
  ```
  Use the options file to run verification:

  ```
  polyspaceCodeProver -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspaceConfigure`. For more information, see `polyspaceConfigure`.

Run Local Verification from MATLAB Command Line

At the MATLAB command-line, enter analysis options and their values as string arguments to the `polyspaceCodeProver` function.

For instance:

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, enter:

  ```
  polyspaceCodeProver('-sources','file.c','-lang','c','-target','m68k')
  ```
- To specify verification precision, use the `-O` option. For instance, to set precision level to 2 for your source file `file.c`, enter:

  ```
  polyspaceCodeProver('-sources','file.c','-lang','c','-O2')
  ```

To see the full list of analysis options, enter:

```
polyspaceCodeProver('-help')
```

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

See Also

polyspaceCodeProver

Troubleshooting Verification Problems

View Error Information When Verification Stops

If verification stops, you can view error information in the user interface or in the log file.

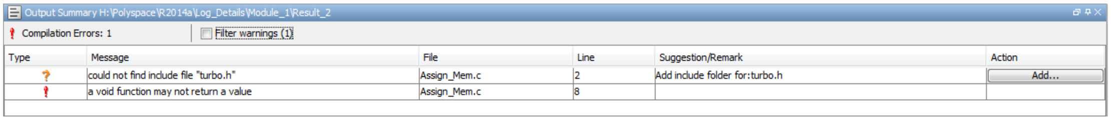
In this section...

View Error Information in User Interface

1 View the errors on the **Output Summary** tab.

2 If you have the **Compilation Assistant** on, to fix the error, you can perform certain actions on the **Output Summary** tab.

   The following figure shows an error due to a missing include file `turbo.h`. You can add the missing file through the **Output Summary** tab.



Output Summary H:\Polyspace\R2014a\Log_Details\Module_1\Result_2

Compilation Errors: 1 □ Filter warnings (1)

Type	Message	File	Line	Suggestion/Remark	Action
?	could not find include file "turbo.h"	Assign_Mem.c	2	Add include folder for:turbo.h	Add...
!	a void function may not return a value	Assign_Mem.c	8		

3 To open the source code at the line containing the error, double-click the message.

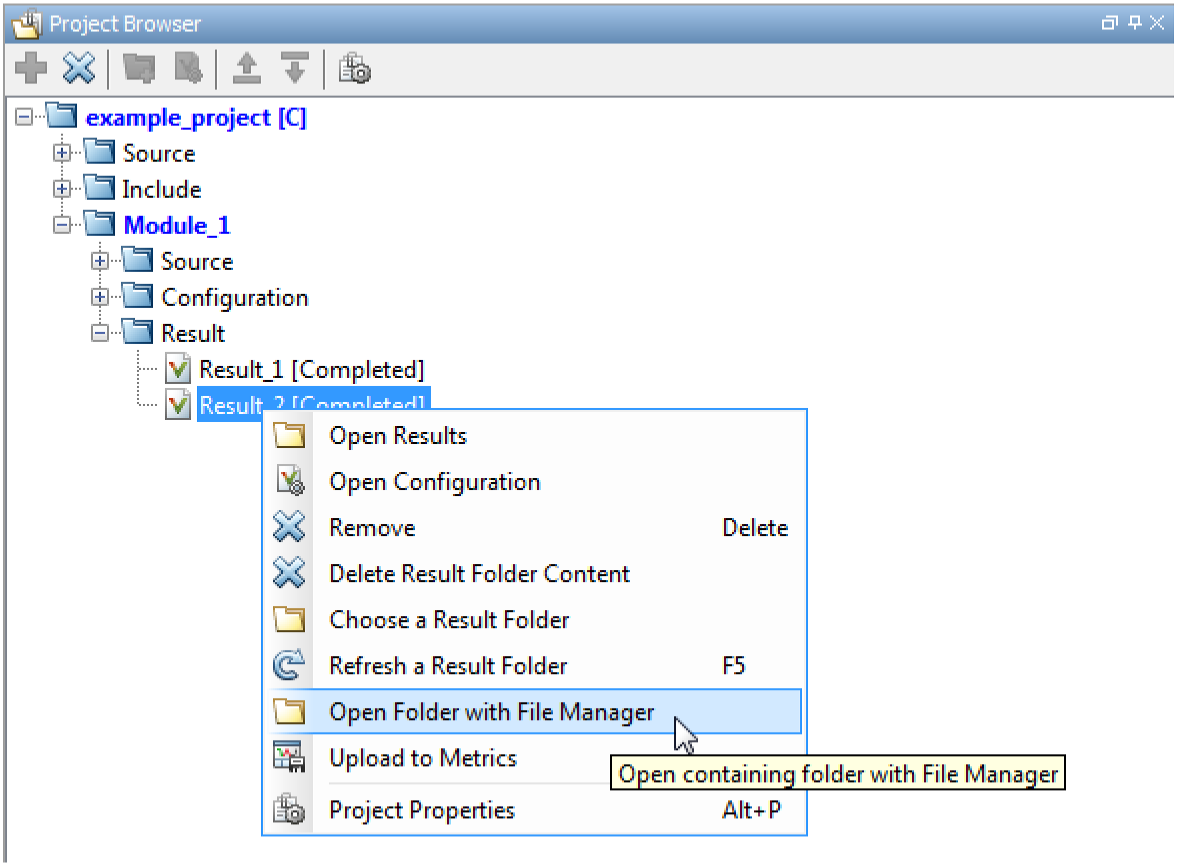
4 To search the log, on the **Search** pane, enter your search term. From the drop down list on this pane, select **Output Summary** or **Run Log**.

   If the **Search** pane is not open by default, select **Windows** > **Show/Hide View** > **Search**.

View Error Information in Log File

You can view errors directly in the log file. The log file is in your results folder. To open the log file:

1 Right-click the result folder name on the **Project Browser** pane. From the context menu, select **Open Folder with File Manager**.



2 Open the log file, `Polyspace_R20##`*`n_ProjectName_date-time`*`.log`

3 To view the errors, scroll through the verification log file, starting at the end and working backward.

The following example shows sample log file information. The error has occurred because the C++ option `-class-analyzer` *`arg`* was used, but the verification cannot find *`arg`* in the source code.

```
----------------------------------------------------------------------
User Program Error: Argument of option -class-analyzer not found.
|                       Class or typedef MyClass does not exist.
|Please correct the program and restart the verifier.
```

```
----------------------------------------------------------------------
-------------------------------------------------------------------
---                                                             ---
---  Verifier has encountered an internal error.                ---
---  Please contact your technical support.                     ---
---                                                             ---
-------------------------------------------------------------------
Failure at: Sep 24, 2009 17:16:26
User time for polyspace-code-prover-nodesktop: 25.6real, 25.6u + 0s
                                                           (0gc)
Error: Exiting because of previous error
***
*** End of Polyspace Verifier analysis
***
```

Troubleshoot Compiler and Linking Errors

When you obtain an error message related to compilation or linking, try:

- Checking whether the error message is related to the dialect that you specified. To specify a different dialect:

    - In the user interface, choose a dialect on the **Configuration** pane. In the **Configuration** tree view, select **Target & Compiler**. From the **Dialect** drop-down list, select an option.
    - At the command line, use the `-dialect` option.

    For more information, see “Dialect (C)”.

- Checking whether the error message is related to stubbing of standard library functions. To avoid this stubbing by Polyspace implementations:

    - For `C`: Use option `-D POLYSPACE_NO_STANDARD_STUBS` or `-D POLYSPACE_STRICT_ANSI_STANDARD_STUBS`.
    - For `C++`: Use option `-no-stl-stubs`.

    For more information, see “Prepare Code for Built-In Functions”.

Contact Technical Support

In this section...
"Provide System Information" on page 8-7
"Provide Information About the Issue" on page 8-7

Provide System Information

When you enter a support request, provide the following system information:

- Hardware configuration
- Operating system
- Polyspace and MATLAB license numbers
- Specific version numbers for Polyspace products
- Installed Bug Report patches

To obtain your configuration information, do one of the following:

- In the Polyspace user interface, select **Help** > **About**.
- At the command line, run the following command:
  - UNIX — `matlabroot/polyspace/bin/polyspace-code-prover-nodesktop -ver`
  - DOS — `MATLAB_Install\polyspace\bin\polyspace-code-prover-nodesktop -ver`

Provide Information About the Issue

If you face compilation issues with your project, see "Troubleshooting in Polyspace Code Prover". If you are still having issues, contact technical support with the following information:

- The verification log.

  The verification log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.txt`. It contains the error message, the options used for the verification and other relevant information.

- The source files related to the compilation error, if possible.

  If you cannot provide the source files:

  - Try to provide a screenshot of the source code section that causes the compilation issue.
  - Try to reproduce the issue with a different code. Provide that code to technical support.

If you are having trouble understanding a result, see the results review guidelines in “Run-Time Checks”. If you are still having trouble understanding the result, contact technical support with the following information:

- The verification log.

  The verification log is a text file generated in your results folder and titled `Polyspace_`*`version`*`_`*`project`*`_`*`date`*`_`*`time`*`.txt`. It contains the the options used for the verification and other relevant information.
- The source files related to the result if possible.

  If you cannot provide the source files:

  - Try provide a screenshot of the relevant source code from the **Source** pane on Polyspace user interface.
  - Try to reproduce the problem with a different code. Provide that code to technical support.

Header File Location Not Specified

Message

```
include.h: No such file or folder
```

Possible Cause

- You did not specify include folders.
- You specified include folders, but a header file is missing from the specified folders.

Solution

Do one of the following:

- Add the missing header file to the specified include folder.
- Specify another include folder that contains the missing file.

For more information, see “Add Sources and Includes”.

Polyspace Cannot Find the Server

Message

```
Error: Cannot instantiate Polyspace cluster
| Check the -scheduler option validity or your default cluster profile
| Could not contact an MJS lookup service using the host computer_name.
    The hostname, computer_name, could not be resolved.
```

Possible Cause

Polyspace uses information provided in **Preferences** to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Provide correct server information.

1. Select **Tools** > **Preferences**.
2. Select the **Server Configuration** tab. Provide your server information.

For more information, see “Set Up Server for Remote Verification and Analysis”.

Errors From Disk Defragmentation and Antivirus Software

Message

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:       0
  Number of invisibles:        949
Stats about alias writes:
  biggest sets of alias writes: foo1:a (733), foo2:x (728), foo1:b (728)
  procedures that write the biggest sets of aliases: foo1 (2679), foo2 (2266),
                                                             foo3 (1288)
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u + 0s (1.4gc)
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty)) raised.
unhandled exception: SysErr: No such file or directory [noent]


--------------------------------------------------------------------------
---                                                                    ---
---  Verifier has encountered an internal error.      ---
---  Please contact your technical support.          ---
---                                                                    ---
---------------------------------------------------------------------------
```

Possible Cause

A disk defragmentation tool or antivirus software is running on your machine.

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the antivirus software. Or, configuring exception rules for the antivirus software to allow Polyspace to run without a failure.

Note: Even if the verification does not fail, the antivirus software can reduce the speed of your verification. This reduction occurs because the software checks the temporary verification files. Configure the antivirus software to exclude your temporary folder, for example, `C:\Temp`, from the checking process.

Insufficient Memory During Report Generation

Message

```
....
Exporting views...
Initializing...
Polyspace Report Generator
Generating Report
 .....
    Converting report
Opening log file:  C:\Users\auser\AppData\Local\Temp\java.log.7512
Document conversion failed
.....
Java exception occurred:
java.lang.OutOfMemoryError: Java heap space
```

Possible Cause

During generation of very large reports, the software can sometimes indicate that there is insufficient memory.

Solution 1

If this error occurs, one workaround is to use the `rte` report format which uses less memory to generate.

1. Generate a report using one of the `rte` options (e.g., `Developer-RTE`).
2. Open the `rte` report in Microsoft® Word, or a similar application.
3. From Word, save or generate your report in the desired format. For example, save as a PDF.

Solution 2

If this error occurs, try increasing the Java® heap size. The default heap size in a 64-bit architecture is 1024 MB.

To increase the size:

1. Navigate to *`matlabroot`*`\polyspace\bin\`*`architecture`*. Where:
   - *`matlab`* is the installation folder.

- `*architecture*` is your computer architecture, for instance, `win32`, `win64`, etc.

2 Change the default heap size that is specified in the file, `java.opts`. For example, to increase the heap size to 2 GB, replace `1024m` with `2048m`.

3 If you do not have write permission for the file, copy the file to another location. After you have made your changes, copy the file back to `*matlabroot*\polyspace\bin\*architecture*\`.

Compilation Error Overview

You can use Polyspace software instead of your compiler to make syntactical, semantic, and other static checks. The Polyspace compiler follows the ANSI C90 standard.

Polyspace detects compilation errors during the standard compliance checking stage, which takes place before the verification stage. The compliance checking stage takes about the same amount of time to run as a compiler. Using Polyspace software early in development yields a number of benefits:

- Detection of link errors
- Detection of errors that only appear with two or more files
- Detection of compiler directives that you need to explicitly declare
- Objective, automatic, and early control of development work (possibly to check code into a configuration management system)

Running Multiple Polyspace Processes

Polyspace Code Prover can be opened simultaneously with Polyspace Bug Finder. However, only one code analysis can be run at a time.

If you try to run multiple Polyspace processes, you will get a `License Error –4,0`. To avoid this error, close any additional Polyspace windows before running an analysis.

Check Compilation Before Verification

Before running a verification, you can enable the Compilation Assistant. If the Compilation Assistant detects compilation errors, the software stops the verification. In the user interface, on the **Output Summary** pane, the software displays errors and suggests possible solutions.

For more information, see “Check for Compilation Issues”.

Syntax Error

Message

```
Verifying compilation.c compilation.c:3: syntax error; found `x'
expecting `;'
```

Code Used

```
void main(void)
{
    int far x;
    x = 0;
    x++;
}
```

Solution

The `far` keyword is unknown in ANSI C. This causes confusion at compilation time. Should `far` be a variable or a qualifier? The `int far x;` construction is illegal.

Possible corrections include:

- Remove `far` from the source code.
- Define `far` as a qualifier, such as `const` or `volatile`.
- Remove `far` artificially by specifying a compilation flag such as `-D far= ` (with a space after the equal sign).

Note: To specify `-D` compilation flags that are generic to the project, for efficiency, use the `-include` option. Refer to "Gather Compilation Options Efficiently".

Undeclared Identifier

Message

```
Verifying compilation.c compilation.c:3: undeclared identifier `x'
```

Code Used

```
void main(void) { x = 0; x++; }
```

Solution

The type is unknown, and therefore the compilation stops. Should `x` be a `float`, an `int`, or a `char`?

Some cross compilers define variables implicitly. Your code must declare variables verification. Polyspace software has no knowledge about implicit variables.

Similarly, some compilers interpret `__SP` as a reference to the stack pointer. Use the `-D` compilation flag.

Note: To specify `-D` compilation flags that are generic to the project, for efficiency, use the `-include` option. Refer to "Gather Compilation Options Efficiently".

Missing Identifiers with Keil or IAR Dialect

Message

```
expected an identifier
```

Possible Cause

If you select Keil or IAR as your dialect, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs. For a list of keywords that are removed, see “Verify Keil or IAR Dialects”.

Solution

To avoid removal of keywords and the compilation error, do one of the following:

- In the user interface, enter `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__` for **Preprocessor definitions**.
- At the command line, use the flag `-D __PST_KEIL_NO_KEYWORDS__` or `-D __PST_IAR_NO_KEYWORDS__`.

Unknown Prototype

Message

```
Error: function 'myfunc' has unknown prototype
```

Code Used

```
var = myfunc(s32var1, ptr->s32var2, 24);
```

`var`, `s32var` are `signed long` data types.

Solution

1. In an include file, for example, `myinclude.h`, specify the complete prototype for the function:

   ```
   #ifndef _INC_H
   #define _INC_H

   extern signed long myfunc(signed long, signed long, signed long);

   #endif
   ```
2. Rerun your verification with the option `-include myinclude.h`.

No Such File or Folder

Messages

Here are examples of messages that include `No such file or folder` and `catastrophic error: could not open source file`:

`compilation.c:1: one_file.h: No such file or folder`

`compilation.c:1: catastrophic error: could not open source file "one_file.h"` (where `one_file.h` is an include file)

Code Used

```
#include "one_file.h"
```

Solution

The `one_file.h` file is missing.

These files are essential for Polyspace software to complete the compilation, for

- Data coherency
- Automatic stubbing

The Polyspace software must be able to find the include folder that contains this file. Specify the include folder in the user interface, or use the `-I` option at the command line, as described in the-I reference page.

#error Directive

The Polyspace software can terminate during compilation with an unsupported platform #error. This error means that the software does not recognize the header data types due to missing compilation flags.

Message

```
#error directive: !Unsupported platform; stopping!
```

Code Used

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#  define MYINT int // then use the int type
#elif defined(__GNUC__) // GCC doesn't support myint
#  define MYINT long // but uses 'long' instead
#else
#  error !Unsupported platform; stopping!
#endif
```

Solution

In the Polyspace software, compilation directives must be explicit. In this example, the compilation stops because you did not specify the __BORLANDC__, or the __VISUALC32__, or the __GNUC__ compilation flags. To fix this error, on the **Configuration** pane, select **Macros**. For **Preprocessor definitions**, specify one of those three compilation flags and restart the verification.

Object is Too Large

A verification can terminate during compilation with a message saying that an object is too large. This error means that the software has detected an object such as an array, union, struct, or class, that is too big for the pointer size of the selected target.

Messages

- `error: array is too large`
- `error: struct or union is too large`
- `error: class is too large for pointer type of %`*`d`*`-bits`

Code Used

```
struct S
{
  char tab[32728];
}s;
```

When using a 16-bit target (for example: `-target mcpu`)

Solution

Use a larger pointer size.

To select a larger pointer:

- If you are using `-target mcpu`, specify `-pointer-is-32bits`.
- If you are using a specific target, specify `-pointer-is-`*`xx`*`bits` if available, otherwise use a larger target.

Initialization of Global Variables (C++)

When a data member of a class is declared static in the class definition, it is a *static member* of the class. Static data members are initialized and destroyed outside the class, as they exist even when no instance of the class has been created.

```
class Test
{
public:

 static int m_number = 0;
};
```

Error message:

```
Verifying test_ko.cpp
/sources/test_ko.cpp, line 4: error: a member with an in-class
initializer must be const
|  static int m_number = 0;
|                      ^

1 error detected in the compilation of "test_ko.cpp".
```

Corrected code:

in file Test.h	**in file Test.cpp**

`class Test`
`{`
`public:`
`static int m_number;`
`};`
	`int Test::m_number = 0;`

Note: Some dialects, other than those supported by Polyspace Code Prover, accept the default initialization of static data member during the declaration.

Double Declarations of Standard Template Library Functions

Consider the following code.

```
#include <list>

void f(const std::list<int*>::const_iterator it) {}
void f(const std::list<int*>::iterator it) {}
void g(const std::list<int*>::const_reverse_iterator it) {}
void g(const std::list<int*>::reverse_iterator it) {}
```

The declared functions belong to `list` container classes with different iterators. However, the software generates the following compilation errors:

```
error: function "f" has already been defined
error: function "g" has already been defined
```

You would also see the same error if, instead of `list`, the specified container was `vector`, `set`, `map`, or `deque`.

To avoid the double declaration errors, use the following Polyspace preprocessing directives:

- `__PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__`
- `__PST_STL_VECTOR_CONST_ITERATOR_DIFFER_ITERATOR__`
- `__PST_STL_SET_CONST_ITERATOR_DIFFER_ITERATOR__`
- `__PST_STL_MAP_CONST_ITERATOR_DIFFER_ITERATOR__`
- `__PST_STL_DEQUE_CONST_ITERATOR_DIFFER_ITERATOR__`

For example, for the given code, run the verification with the directive for the `list` container:

```
-D __PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__
```

Large Static Initializer

If you see a large initializer warning during the compilation phase, for example,

```
lsi_eg.c, line 86: warning: initializer is too large, this may cause scaling troubles.
Please refer to the "Troubleshooting" section of the User Guide
|  const unsigned char CJK_S_MKT_CTS_BE_HDB[] = {
|                                               ^
```

the compilation might:

- Fail with an error if no memory space is available:
  - Windows — `Error: A segmentation fault occurred in "edgcpfe.x86-mingw32.exe"`
  - Linux — `Error: The process "edgcpfe.x86-linux" received the signal 11`
- Take a long time to complete.

To avoid this issue, rerun your verification with the following option:

```
-cfe-extra-flags --truncate_huge_initializer
```

This option is valid:

- For a C verification only.
- Only if no variable or function address is referenced within the initializer

Compilation Messages

Phrase Found in Message	See
`syntax error`	"Syntax Error" on page 8-17
`undeclared identifier`	"Undeclared Identifier" on page 8-18
`unknown prototype`	"Unknown Prototype" on page 8-20

`No such file or folder`
or
`Catastrophic error: could not open source file`
	"No Such File or Folder" on page 8-21

`#error: directive`	"`#error` Directive" on page 8-22

C++ Dialect Issues

In this section...

ISO versus Default Dialects

The ISO® dialect strictly follows the ISO/IEC 14882:1998 ANSI C++ standard. If you specify the -dialect iso option, the Polyspace compiler might produce permissiveness errors. The following code contains five common permissiveness errors that occur if you specify the -dialect iso option. These errors are explained in detail following the code.

If you do not specify the -dialect option, the Polyspace compiler uses a default dialect that many C++ compilers use; the default dialect is more permissive with regard to the C ++ standard.

Original code (file permissive.cpp):

```

class B {} ;
class A
{
friend B ;
enum e ;
void f() { long float ff = 0.0 ;}
enum e { OK = 0, KO } ;
};
template <class T>
struct traits
{
typedef T * pointer ;
typedef T * pointer ;
} ;
template<class T>
```

```
17  struct C
18  {
19  typedef traits<T>::pointer pointer ;
20  } ;
21  int main()
22  {
23  C<int> c ;
23  }
```

- Using `-dialect iso`, line 5 should be: `friend class B`:

```
"./sources/permissive.cpp", line 5: error: omission of "class"
is nonstandard
  friend B ;
```

- Using `-dialect iso`, the line 6 must be removed:

```
"./sources /permissive.cpp", line 6: error: forward declaration
of enum type
is nonstandard
  enum  e ;
    ^
```

- Using `-dialect iso`, line 7 should be: `double ff = 0.0`:

```
"./sources/permissive.cpp", line 7: error: invalid combination
of type
specifiers
  long float ff = 0.0 ;
  ^
```

- Using `-dialect iso`, line 14 needs to be removed:

```
"./sources/permissive.cpp", line 14: error: class member typedef
may not be
redeclared
  typedef T *  pointer ; // duplicate !
       ^
```

- Using `-dialect iso`, line 21 needs to be changed by: `typedef` *`typename`* `traits<T>::pointer pointer`

```
"./sources/permissive.cpp", line 21: error: nontype
"traits<T>::pointer [with T=T]" is not a type name
  typedef traits<T>::pointer pointer ;
```

These error messages disappear if you specify the `-dialect` default option.

CFront2 and CFront3 Dialects

The cfront2 and cfront3 dialects were being used before the publication of the ANSI C ++ standard in 1998. Nowadays, these two dialects are used to compile legacy C++ code.

If the cfront2 or cfront3 options are not selected, you may get the common error messages below.

Variable Scope Issues

The ANSI C++ standard specifies that the scope of the declarations occurring inside loop definition is local to the loop. However some compilers may assume that the scope is local to the bloc ({ }) that contains the loop.

Original code:

```
for (int i = 0; i < maxval; i++) {...}
if (i == maxval) {
 ...
}
```

Error message:

```
Verifying Test.cpp
"../sources/Test.cpp", line 26: error: identifier "i" is undefined
  if (i == maxval) {
    ^
```

Note: This kind of construction has been allowed by compilers until 1999, before the standard became more strict.

"bool" Issues

Standard type may need to be turned into boolean type.

Original code:

```
enum bool
 {
  FALSE=0,
  TRUE
 };
```

```
class CBool
{
public:
 CBool ();
 CBool (bool val);
 bool m_val;
};
```

Error message:

```
Verifying C++ sources ...
Verifying CBool.cpp
"../sources/CBool.h", line 4: error: expected either a definition
or a tag name
 enum  bool
   ^
```

Visual Dialects

The following messages appears if the compiler is based on a Visual® dialect (including `visual8`).

Import Folder

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file that contains some definitions. These header files have a `.tlh` extension, and Polyspace for C/C++ requires the folder containing those files.

Original code:

```
#include "stdafx.h"
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
 return 0;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file "./MsXml.tlh"
```

```
#import <MsXml.tlb>
        ^
```

The Visual C++ compiler generates these files in its “build-in” folder (usually Debug or Release). Therefore, in order to provide those files, the application needs to be built first. Then, the option `-import-dir=<build folder>` must be set with the path folder.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp

`#pragma pack(4)`
 	
`#include "type.h"`
	`struct A`
	`{`
	` char c ;`
	` int i ;`
	`} ;`
		`#pragma pack(2)`
		
		`#include "type.h"`

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
 struct A
    ^
   detected during compilation of secondary translation unit
"test2.cpp"
```

The option `-ignore-pragma-pack` is mandatory to continue the verification.

GNU Dialect

For the GNU dialect, you can select the GCC 3.4 or GCC 4.6 version. If you use this dialect, Polyspace does not produce an error during the **Compile** phase because of assembly language keywords such as `__asm__ __volatile__`. However, for verification, Polyspace ignores the content of the assembly-language code.

Polyspace software supports the following GNU elements:

- Variable length arrays

- Anonymous structures:

  ```
  void f(int n) { char tmp[n] ; /* ... */ }

  union A {
   struct {
    double x ;
    double y ;
    double z ;
   };
   double tab[3];
  } a ;


  void main(void) {

   assert(&(a.tab[0]) == &(a.x)) ;

  }
  ```
- Other syntactic constructions allowed by GCC, except as noted below
- Statement expressions:

  ```
  int i = ({ int tmp ; tmp = f() ; if (tmp > 0 ) { tmp = 0 ; } tmp ; })
  ```

Partial Support

Zero-length arrays have the same support as in Visual Mode. They are allowed when used through a pointer, but not in a local variable.

Syntactic Support Only

Polyspace software provides syntactic support for the following options, but not semantic support:

- `__attribute__(...)` is allowed, but generally not taken into account.
- No special stubs are computed for predeclared functions such as `__builtin_cos`, `__builin_exit`, and `__builtin_fprintf`).

Not Supported

The following options are not supported:

- The keyword `__thread`

- Taking the address of a label:

  ```
  { L : void *a = &&L ; goto *a ; }
  ```
- General C99 features supported by default in GCC, such as complex built-in types (`__complex__`, `__real__`, etc.).
- Extended designators initialization:

  ```
  struct X { double a; int b[10] } x = { .b = { 1, [5] =2 },
  .b[3] = 1, .a = 42.0 };
  ```
- Nested functions

Examples

Example 1: `_asm_volatile_` keyword

In the following example, for the `inb_p` function to manage the return of the local variable `_v`, the `__asm__ __volatile__` keyword is used as follows:

```
extern inline unsigned char
inb_p (unsigned short port)
{
  unsigned char _v;

  __asm__ __volatile__ ("inb %w1,%0\noutb %%al,$0x80":"=a"
         (_v):"Nd" (port));
  return _v;
}
...
```

Although Polyspace does not produce an error during the **Compile** phase, it ignores the assembly code. An orange **Non-initialized local variable** error appears on the `return` statement after verification. For more information, see “Local Variables in Functions with Assembly Code”.

Example 2: Anonymous Structure

The following example shows an unnamed structure supported by GNU:

```
class x
{
public:

  struct {
```

```
  unsigned int a;
  unsigned int b;
  unsigned int c;
  };
  unsigned short pcia;
  enum{
  ea = 0x1,
  eb = 0x2,
  ec = 0x3
  };

  struct {
  unsigned int z1;
  unsigned int z2;
  unsigned int z3;
  unsigned int z4;
  };
};

int main(int argc, char *argv[])
{
  class x myx;

  myx.a = 10;
  myx.z1 = 11;
  return(0);
}
```

C Link Errors

In this section...

Link Error Overview (C)

This section describes how to address some common types of link errors for C code.

Link errors result from the checking that Polyspace performs for compliance with ANSI C standards. Link error messages can apply to functions, variables, and varargs.

The error message includes specific information that reflects the code that the Polyspace software is checking, such as the function name and type declaration.

Function: Procedure Multiply Defined

Files Used

`header.h`	`file1.c`	`file2.c`

`#include <stdio.h>`
 	
`void func() {`
` }`
	`#include "header.h"`	`#include "header.h"`

Polyspace Output

```
Error:
```

```
procedure func multiply defined
```

Solution

For C code, to allow such multiple inclusion of the header containing the function body, use the option `-static-headers-object`.

Function: Wrong Argument Type

Polyspace Output

```
Error:
global declaration of 'f' function has incompatible type with its definition
Declared function type has 'arg 1' type incompatible with definition.
```

```
int f(float y)            int f(int *y);
{
  int r;                  void main(void)
  r=12;                   {
}                           int r;
                            r = f(&r);
                          }
```

Solution

The first parameter for the `f` function is either a float or a pointer to an integer. The global declaration must match the definition.

Function: Wrong Argument Number

Polyspace Output

```
Error:
global declaration of 'f' function has incompatible type with its definition
Declared function type has incompatible number of arguments with definition.
```

```
int f(float y)            int f(float y, float x);
{
  int r;                  void main(void)
  r=12;                   {
}                           int a;
                            float b, c;
                            a = f(b, c);
                          }
```

Solution

These two functions have a different number of arguments. This mismatch in the number of arguments results in a nondeterministic execution.

Function: Wrong Return Type

Polyspace Output

```
Verifying cross-files ANSI C compliance ...
Error: global declaration of 'f' function has incompatible type with its definition
declared function type has incompatible return type with definition
declared 'float' (size 64) type incompatible with defined 'int' (size 32) type
```

```
float f(int y)           int f(int y);
{
float r;                 void main(void)
r=1.0;                   {
return r;                int r;
}                        r = f(r);
                         }
```

Solution

Use the same return type for the declaration and definition of function `f`.

Variable: Wrong Type

Polyspace Output

```
Verifying cross-files ANSI C compliance ...
Error: global declaration of 'x' variable has incompatible type with its definition
  declared 'float' (32) type incompatible with defined 'int' (32) type
```

```
extern float x               int x;
                             void main(void)
                             {}
```

Solution

Declare the `x` variable the same way in every file. If a variable `x` is as an integer equal to 1, which is 0x0001, what does this value mean when seen as a float? It could result in a NaN (Not a Number) during execution.

Variable: Signed/Unsigned

Polyspace Output

```
Verifying cross-files ANSI C compliance ...
Error: global declaration of 'x' variable has incompatible type with its definition
  declared 'unsigned' type incompatible with defined 'signed' type
```

```
extern unsigned char x;      char x;
```

```
                             void main(void)
                             {}
```

Solution

Consider the 8-bit binary value 10000010. Given that a `char` is 8 bits, it is not clear whether it is 130 (unsigned), or maybe -126 (signed).

Variable: Different Qualifier

Polyspace Output

```
Verifying cross-files ANSI C compliance ...
Warning: global declaration of 'x' variable has incompatible type with its definition
  declared 'non qualified' type incompatible with defined 'volatile' type
  'volatile' qualifier used
```

```
extern int x;            volatile int x;

                         void main(void)
                         {}
```

Solution

Polyspace software flags the volatile qualifier, because that qualifier has major implications for the verification. Because it is not clear which statement is correct, the verification process generates a warning.

Variable: Array Against Variable

Polyspace Output

```
Verifying cross-files ANSI C compliance ...
Error: global declaration of 'x' variable has incompatible type with its definition
  declared 'array' (384) type incompatible with defined 'int' (32) type
```

```
extern int x[12];          int x;

                           void main(void)
                           {

                           }
```

Solution

The real allocated size for the `x` variable is one integer. Any function attempting to manipulate `x[]` corrupts memory.

Variable: Wrong Array Size

Polyspace Output

```
Verifying cross-files ANSI C compliance ...
Warning: global declaration of 'x' variable has incompatible type with its definition
 declared array type has 'upper bound' 5 inferior to definition 'upper bound' 12
```

```
extern int x[12];          int x[5];

                           void main(void)
                           {

                           }
```

Solution

The real allocated size for the `x` variable is five integers. Any function attempting to manipulate `x[]` between `x[5]` and `x[11]` corrupts memory.

Missing Required Prototype for varargs

Polyspace Output

```
Verifying cross-files ANSI C compliance ...
Error: missing required prototype for varargs. procedure 'g'
```

```
void g(int, ...);          void main(void)
                           {
void f(void)               g(4);
{                          }
g(12,“abcde“,40)
}
```

Solution

Declare the prototype for `g` when `main` executes.

To eliminate this error, you can add the following line to `main`:

```
void g(int, ...)
```

Or, you can avoid modifying `main` by adding that same line in a new file and then when you launch the verification, use the `-include` option:

```
—include c:\Polyspace\new_file.h
```

where `new_file.h` is the new file that includes the line `void g(int, ...)`.

C++ Link Errors

In this section...

STL Library C++ Stubbing Errors

Polyspace software provides an efficient implementation of all functions in the Standard Template Library (STL). The STL and platforms may have different declarations and definitions; otherwise, the following error messages appear:

Original code:

```
#include <map>

struct A
{
 int m_val;
};

struct B
{
 int m_val;
 B& operator=(B &) ;
};

typedef std::map<A, B> MAP ;

int main()
{
 MAP m ;
 A a ;
 B b ;

 m.insert(std::make_pair(a,b)) ;
}
```

Error message:

```
Verifying template.cpp
"<Product>/Verifier/cinclude/new_stl/map", line 205: error: no operator
```

```
"=" matches these operands
 operand types are: pair<A, B> = const map<A, B, less<A>>::value_type
 { volatile int random_alias = 0 ; if (random_alias) *((pair<Key, T> * )
_pst_elements) = x ; } ; // read of x is done here

 detected during instantiation of
"pair<__pst__generic_iterator<bidirectional_iterator_tag, pair<const Key,
T>>, bool> map<Key, T, Compare>::insert(const map<Key, T, Compare>::
value_type &) [with Key=A, T=B, Compare=less<A>]" at line 23 of "/cygdrive/
c/_BDS/Test-Polyspace/sources/template.cpp"
```

Using the `-no-stub-stl` option avoids this error message. Then, you need to add the folder containing definitions of own STL library as a folder to include using the option `-I`.

The preceding message can also appear with the folder names:

```
"<Product>/cinclude/new_stl/map", line 205: error: no operator "="
matches these operands
```

```
"<Product>/cinclude/pst_stl/vector", line 64: error: more than one
operator "=" matches these operands:
```

Be careful that other compile or linking troubles can appear with your own template definitions.

Lib C Stubbing Errors

Extern C Functions

Some functions may be declared inside an `extern “C” { }` bloc in some files, but not in others. In this case, the linkage is different which causes a link error, because it is forbidden by the ANSI standard.

Original code:

```
extern "C" {
 void* memcpy(void*, void*, int);
}
class Copy
{
public:
 Copy() {};
 static void* make(char*, char*, int);
};
void* Copy::make(char* dest, char* src, int size)
{
 return memcpy(dest, src, size);
}
```

Error message:

```
Pre-linking C++ sources ...

<results_dir>/test.cpp, line 2: error: declaration of function "memcpy"
is incompatible with a declaration in another translation unit
(parameters do not match)
|            the other declaration is at line 4096 of "__polyspace__stdstubs.c"
|    void* memcpy(void*, void*, int);
|          ^
|          detected during compilation of secondary translation unit "test.cpp"
```

The function `memcpy` is declared as an external "C" function and as a C++ function. It causes a link problem. Indeed, function management behavior differs whether it relates to a C or a C++ function.

When such error happens, the solution is to homogenize declarations, i.e. add `extern "C" { }` around previous listed C functions.

Another solution consists in using the permissive option `-no-extern-C`. It removes all `extern "C"` declarations.

Functional Limitations on Some Stubbed Standard ANSI Functions

- `signal.h` is stubbed with functional limitations: `signal` and `raise` functions do not follow the associated functional model. Even if the function raise is called, the stored function pointer associated to the signal number is not called.
- No jump is performed even if the `setjmp` and `longjmp` functions are called.
- `errno.h` is partially stubbed. Some math functions do not set `errno`, but instead, generate a red error when a range or domain error occurs with **ASRT** checks.

You can also use the compile option `POLYSPACE_STRICT_ANSI_STANDARD_STUBS` (`-D` flag). This option only deactivates extensions to ANSI C standard libC, including the functions `bzero`, `bcopy`, `bcmp`, `chdir`, `chown`, `close`, `fchown`, `fork`, `fsync`, `getlogin`, `getuid`, `geteuid`, `getgid`, `lchown`, `link`, `pipe`, `read`, `pread`, `resolvepath`, `setuid`, `setegid`, `seteuid`, `setgid`, `sleep`, `sync`, `symlink`, `ttyname`, `unlink`, `vfork`, `write`, `pwrite`, `open`, `creat`, `sigsetjmp`, `__sigsetjmp`, and `siglongjmpare`.

Standard Library Function Stubbing Errors

In this section...

Conflicts Between Library Functions and Polyspace Stubs

A code set compiles successfully for a target, but during the `__polyspace_stdstubs.c` compilation phase for the same code, Polyspace software generates an error message.

The error message highlights conflicts between:

- A standard library function that the application includes
- One of the standard stubs that Polyspace software uses in place of the function

For more information about errors generated during automatic stub creation, see "Automatic Stubbing Errors" on page 8-50.

_polyspace_stdstubs.c Compilation Errors

Here are examples of the errors relating to stubbing standard library functions. The code uses standard library functions such as `sprintf` and `strcpy`, illustrating possible problems with these functions.

Example 1

```
C-STUBS/__polyspace__stdstubs.c:1117: string.h: No such file or
folder
```

```
Verifying C-STUBS/__polyspace__stdstubs.c

C-STUBS/__polyspace__stdstubs.c:1118: syntax error; found `strlen'
expecting `;'

C-STUBS/__polyspace__stdstubs.c:1120: syntax error; found `i'
expecting `;'

C-STUBS/__polyspace__stdstubs.c:1120: undeclared identifier `i'
```

Example 2

```
Verifying C-STUBS/__polyspace__stdstubs.c

Error: missing required prototype for varargs. procedure 'sprintf'
```

Example 3

```
Verifying C-STUBS/__polyspace__stdstubs.c

C-STUBS/__polyspace__stdstubs.c:3027: missing parameter 4 type

C-STUBS/__polyspace__stdstubs.c:3027: syntax error; found `n'
expecting `)'

C-STUBS/__polyspace__stdstubs.c:3027: skipping `n'

C-STUBS/__polyspace__stdstubs.c:3037: undeclared identifier `n'”
```

Troubleshooting Approaches for Standard Library Function Stubs

You can use a range of techniques to address errors relating to stubbing standard library functions. These techniques reflect different balances for the verification between:

- Precision
- Amount of time preparing the code
- Execution time

Try the techniques in any order. Consider trying the simplest approaches first, and trying other techniques as required to achieve the balance of the trade-offs that you seek.

Here are the techniques, listed in order of estimated simplicity, from simplest to most thorough:

- "Restart with the -I option" on page 8-46
- "Replace Automatic Stubbing with Include Files" on page 8-46
- "Create _polyspace_stdstubs.c File with Required Includes" on page 8-47

  (Use when precision is important enough to justify extensive code preparation time)
- "Provide .c file Containing Prototype Function" on page 8-48

  (Use when you do not want to invest much time for code preparation time)
- "Ignore _polyspace_stdstubs.c" on page 8-49

If the problem persists after trying these solutions, contact MathWorks support.

Restart with the -I option

Generally you can best address stubbing errors by restarting the verification. Include the header file containing the prototype and the required definitions, as used during compilation for the target.

The least invasive way of including the header file containing the prototype is to use the `-I` option.

Replace Automatic Stubbing with Include Files

The Polyspace software provides a selection of files that contain stubs for most standard library functions. You can use those stubs in place of automatic stubbing.

For replacement of stubbing to work effectively, provide the include file for the function. In the following example, the standard library function is `strlen`. This example assumes that you have included `string.h`. Because the `string.h` file can differ between targets, there are no default include folders for Polyspace stub files.

If the compiler has implicit include files, manually specify those include files, as shown in this example.

```
(_polyspace_stdstubs.c located in <<results_dir>>
_polyspace_stdstubs.c
#if defined(_polyspace_strlen) || ... || defined(_polyspace_strtok)
#include <string.h>
size_t strlen(const char *s)
{
 size_t i=0;
 while (s[i] != 0)
  i++;
 return i;
}
#endif /* _polyspace_strlen */
```

If problems persist, try one of these solutions:

- "Create _polyspace_stdstubs.c File with Required Includes" on page 8-47
- "Provide .c file Containing Prototype Function" on page 8-48
- "Ignore _polyspace_stdstubs.c" on page 8-49

Create _polyspace_stdstubs.c File with Required Includes

1. Copy `<<results_dir>>/C-ALL/C-STUBS/ _polyspace_stdstubs.c` to the `sources` folder and rename it `polyspace_stubs.c`.

   This file contains the whole list of stubbed functions, user functions, and standard library functions. For example:

   ```
   #define _polyspace_strlen
   #define a_user_function
   ```

2. Find the problem function in the file. For example:

   ```
   #if defined(_polyspace_strlen) || ... || defined(_polyspace_strtok)
   #include <string.h>
    size_t strlen(const char *s)
    {
     size_t i=0;
     while (s[i] != 0)
      i++;
     return i;
    }
   #endif /* __polyspace_strlen */
   ```

   The verification requires you to include the `string.h` file that the application uses.

3 Provide the `string.h` file that contains the real prototype and type definitions for the stubbed function.

Alternatively, extract the relevant part of that file for inclusion in the verification.

For example, for `strlen`:

```
string.h
 // put it in the /homemade_include folder
 typedef int size_t;
 size_t strlen(const char *s);
```

4 Specify the path for the include files and relaunch Polyspace, using one of these commands:

```
polyspace-code-prover-nodesktop -I /homemade_include
```

or

```
polyspace-code-prover-nodesktop -I /our_target_include_path
```

Provide .c file Containing Prototype Function

1 Identify the function causing the problem (for example, `sprintf`).

2 Add a `.c` file to your verification containing the prototype for this function.

3 Restart the verification either from the user interface or from the command line.

You can find other `__polyspace_no`*function_name* options in `_polyspace__stdstubs.c` files, such as:

```
__polyspace_no_vprintf
__polyspace_no_vsprintf
__polyspace_no_fprintf
__polyspace_no_fscanf
__polyspace_no_printf
__polyspace_no_scanf
__polyspace_no_sprintf
__polyspace_no_sscanf
__polyspace_no_fgetc
__polyspace_no_fgets
__polyspace_no_fputc
__polyspace_no_fputs
__polyspace_no_getc
```

Note: If you are considering defining multiple project generic `-D` options, using the `-include` option can provide a more efficient solution to this type of error. Refer to "Gather Compilation Options Efficiently".

Ignore _polyspace_stdstubs.c

When all other troubleshooting approaches have failed, you can try ignoring `_polyspace_stdstubs.c`. To ignore `_polyspace_stdstubs.c`, but still see which standard library functions are in use:

1. Do one of the following:

   - Deactivate all standard stubs using `-D POLYSPACE_NO_STANDARD_STUBS` option. For example:

     ```
     polyspace-code-prover-nodesktop -D POLYSPACE_NO_STANDARD_STUBS
     ```
   - Deactivate all stubbed extensions to ANSI C standard by using `-D POLYSPACE_STRICT_ANSI_STANDARD_STUBS`. For example:

     ```
     polyspace-code-prover-nodesktop -D
     POLYSPACE_STRICT_ANSI_STANDARD_STUBS
     ```

   This approach presents a list of functions Polyspace software tries to stub. It also lists the standard functions in use (most probably without a prototype), and generates the following type of message:

   ```
   * Function strcpy may write to its arguments and may
   return parts of them. Does not model pointer effects.
   Returns an initialized value.

   Fatal error: function 'strcpy' has unknown prototype
   ```
2. Add an include file in the C file that uses your standard library function. If you restart Polyspace with the same options, the default behavior results for these stubs for this particular function.

Consider the example `size_t strcpy(char *s, const char *i)` stubbed to

- Write anything in `*s`
- Return any possible `size_t`

Automatic Stubbing Errors

In this section...

Three Types of Error Messages

The Polyspace software generates three different types of error messages during the automatic creation of stubs.

For more information about stubbing errors, see “Standard Library Function Stubbing Errors” on page 8-44.

Unknown Prototype Error

Message

```
Fatal error: function 'f' has unknown prototype
-------------------------
Error message explanation:
- "function has wrong prototype" means that either the function
 has no prototype or its prototype is not ANSI compliant.
- "task is undefined" means that a function has been declared
 to be a task but has no known body
```

Solution

Provide an ANSI-compliant prototype.

Parameter `-entry-points` Error

Message

```
*** Verifier found an error in parameter -entry-points: task "w"
must be a userdef function
------------------------------------------------------------- ---
---                      ---
--- Found some errors in launching command.        ---
--- Please consult rte-kernel -h to correct them      ---
--- and launch the verification again.          ---
---                      ---
------------------------------------------------------------ ---
```

Solution

A function or procedure declared to be an `-entry-points` cannot be an automatically stubbed function.

Reduce Verification Time

In this section...

Factors Affecting Verification Time

These factors affect how long it takes to run a verification:

- The size of the code
- The number of global variables
- The nesting depth of the variables (the more nested they are, the longer it takes)
- The depth of the call tree of the application
- The intrinsic complexity of the code, particularly with regards to pointer manipulation

Because many factors affect verification time, there is no precise formula for calculating verification duration. Instead, Polyspace software provides graphical and textual output to indicate how the verification is progressing.

Techniques to Improve Verification Performance

This section suggests methods to reduce the duration of a particular verification, with minimal compromise for the launch parameters or the precision of the results.

You can increase the size of a code sample for effective analysis by tuning the tool for that sample. Beyond that point, subdividing the code or choosing a lower precision level offers better results (`-O1`, `-O0`).

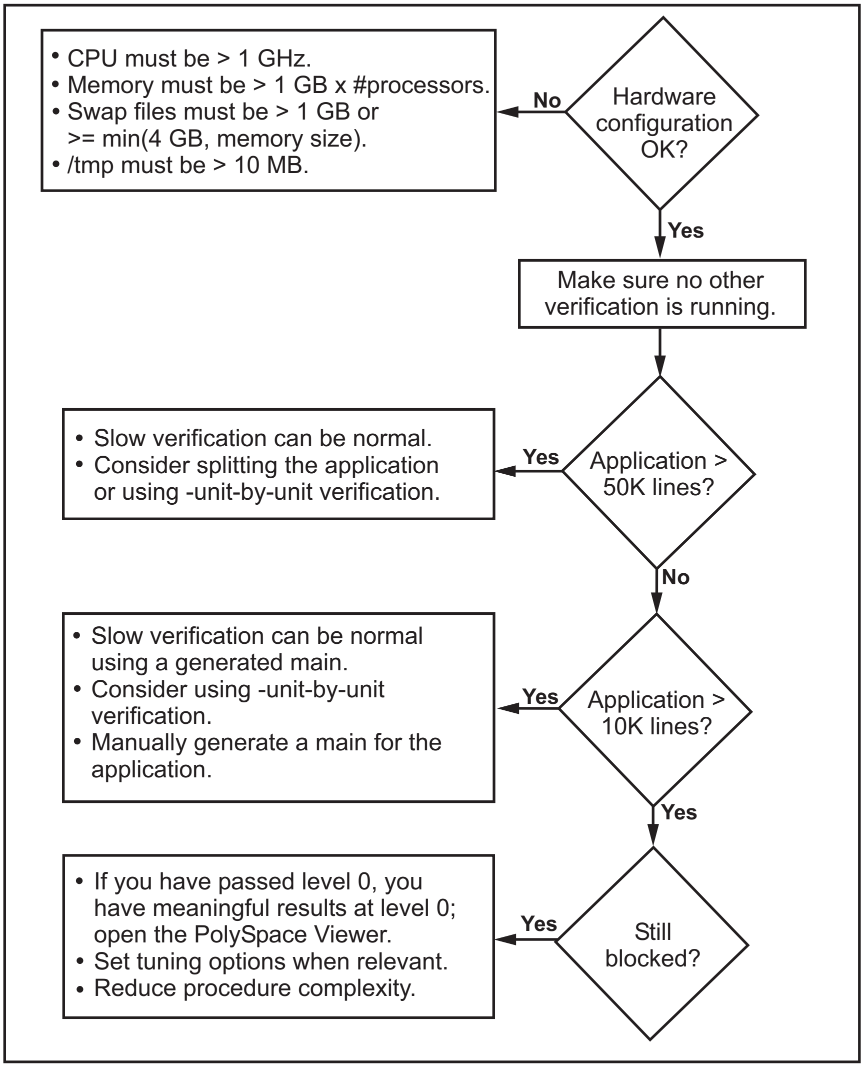
You can use several techniques to reduce the amount of time required for a verification, including

- “Errors From Disk Defragmentation and Antivirus Software” on page 8-11
- “Tune Polyspace Parameters” on page 8-55
- “Subdivide Code” on page 8-55
- “Reduce Procedure Complexity” on page 8-63
- “Reduce Task Complexity” on page 8-66
- “Reduce Variable Complexity” on page 8-66
- “Choose Lower Precision” on page 8-67

You can combine these techniques. See the following performance-tuning flow charts:

- “Standard Scaling Options Flow Chart” on page 8-54
- “Reduce Code Complexity” on page 8-54

Standard Scaling Options Flow Chart



- CPU must be > 1 GHz.
- Memory must be > 1 GB x #processors.
- Swap files must be > 1 GB or >= min(4 GB, memory size).
- /tmp must be > 10 MB.

Make sure no other verification is running.

- Slow verification can be normal.
- Consider splitting the application or using -unit-by-unit verification.

- Slow verification can be normal using a generated main.
- Consider using -unit-by-unit verification.
- Manually generate a main for the application.

- If you have passed level 0, you have meaningful results at level 0; open the PolySpace Viewer.
- Set tuning options when relevant.
- Reduce procedure complexity.

Reduce Code Complexity

To reduce code complexity, try the following techniques, in the order listed:

- "Reduce Procedure Complexity" on page 8-63

- “Reduce Task Complexity” on page 8-66
- “Reduce Variable Complexity” on page 8-66

After you use any of these techniques, restart the verification.

Tune Polyspace Parameters

Impact of Parameter Settings

Compromise to balance the time required to perform a verification and the time required to review the results. Launching Polyspace verification with the following options reduces the time taken for verification. However, these parameter settings compromise the precision of the results. The less precise the results of the verification, the more time you can spend reviewing the results.

Recommended Parameter Tuning

Use the parameters in the sequence listed. If the first suggestion does not increase the speed of verification sufficiently, then introduce the second, and so on.

- Switch from `-O2` to a lower precision.
- Set the `-respect-types-in-globals` and `-respect-types-in-fields` options.
- Set the `-k-limiting` option to 2, then 1, or 0.
- Manually stub missing functions which write into their arguments.
- If some code uses some large arrays, use the `-no-fold` option.

For example:

```
polyspace-code-prover-nodesktop -O0 -respect-types-in-globals -k-
limiting 0
```

Subdivide Code

- “An Ideal Application Size” on page 8-56
- “Benefits of Subdividing Code” on page 8-56
- “Possible Issues with Subdividing Code” on page 8-56
- “Approach” on page 8-58

- "Select a Subset of Code" on page 8-59

An Ideal Application Size

People have used Polyspace software to analyze numerous applications with greater than 100,000 lines of code.

There is a trade-off between the time and resources required to analyze an application, and the resulting selectivity. The larger the project size, the broader the approximations Polyspace software makes. Broader approximations produce more oranges. Large applications can require you to spend much more time analyzing the results and your application.

These approximations enable Polyspace software to extend the range of project sizes it can manage, to perform the verification further, and to solve traditionally incomputable problems. Balance the benefits derived from verifying a whole large application against the loss of precision that results.

Benefits of Subdividing Code

Subdividing a large application into smaller subsets of code provides several benefits. You:

- Quickly isolate a meaningful subset
- Keep all functional modules
- Can maintain a high precision level (for example, level O2)
- Reduce the number of orange items
- Do not have to remove threads that affect shared data
- Reduce the code complexity considerably

Possible Issues with Subdividing Code

Subdividing code can lead to these problems:

- Orange checks can result from a lack of information regarding the relationship between modules, tasks, or variables.
- Orange checks can result from using too wide a range of values for stubbed functions.
- Some loss of precision; the verification consider all possible values for a variable.

When the Application is Incomplete

When the code consists of a small subset of a larger project, Polyspace software automatically stubs many procedures. Polyspace bases the stubbing on the specification or prototype of the missing functions. Polyspace verification assumes that all possible values for the parameter type are returnable.

Consider two 32-bit integers `a` and `b`, which are initialized with their full range due to missing functions. Here, `a*b` causes an overflow, because `a` and `b` can be equal to 2^31. Precise stubbing can reduce the number of incidences of these data set issue orange checks.

Now consider a procedure `f` that modifies its input parameters `a` and `b`. `f` passes both parameters by reference. Suppose `a` can be from 0 through 10, and `b` any value between -10 and 10. In an automatically stubbed function, the combination `a=10` and `b=10` is possible, even if it is not possible with the real function. This situation introduces orange checks in a code snippet such as `1/(a*b - 100)`, where the division would be orange.

- So, even with precise stubbing, verification of a small section of code can introduce extra orange checks. However, the net effect from reducing the complexity is to reduce the total number of orange checks.
- With default stubbing, the increase in the number of orange checks as the result of this phenomenon tends to be more pronounced.

Considering the Effects of Application Code Size

Polyspace can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation use a superset of the actual possible values.

For instance, in a relatively small application, Polyspace software can retain detailed information about the data at a particular point in the code. For example, the variable `VAR` can take the values
{ –2 ; 1 ; 2 ; 10 ; 15 ; 16 ; 17 ; 25 }

If the code uses `VAR` to divide, the division is green (because 0 is not a possible value).

If the program is large, Polyspace software simplifies the internal data representation by using a less precise approximation, such as:

```
[-2 ; 2] U {10} U [15 ; 17] U {25}
```

Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, Polyspace can further simplify the VAR range to (for example):

```
[-2 ; 20]
```

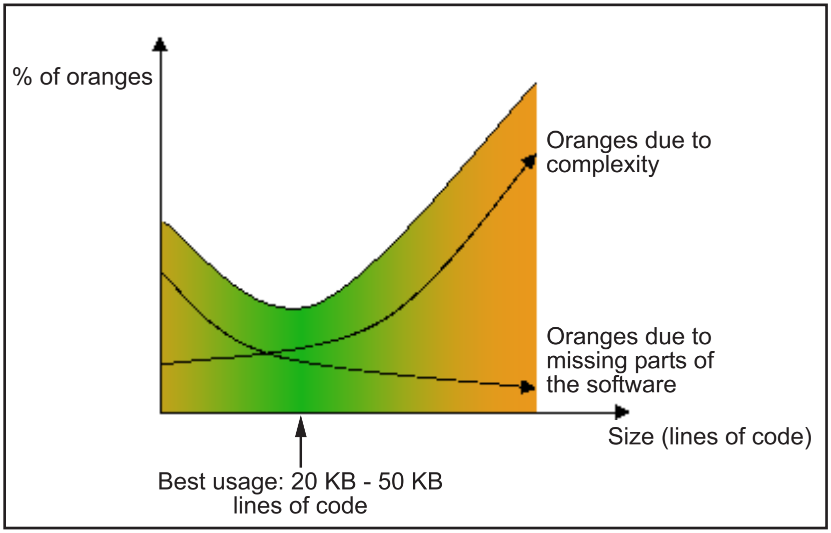
This phenomenon increases the number of orange warnings when the size of the program becomes large.

Approach

Begin with file-by-file verifications (when dealing with C language), package-by-package verifications (when dealing with Ada language), and class-by-class verifications (when dealing with C++ language).

The maximum application size is between 20,000 (for C++) and 50,000 lines of code (for C and Ada). For such applications of that size, approximations are not too significant. However, sometimes verification time is extensive.

Experience suggests that subdividing an application before verification normally has a beneficial impact on selectivity. The verification produces more red, green and gray checks, and fewer unproven orange checks. This subdivision approach makes bug detection more efficient.



A compromise between selectivity and size

Polyspace verification is most effective when you use it as early as possible in the development process, before any other form of testing.

When you analyze a small module (for example, a file, piece of code, or package) using Polyspace software, focus on the red and gray checks. orange unproven checks at this stage are interesting, because most of them deal with robustness of the application. The orange checks change to red, gray, or green as the project progresses and you integrate more modules.

In the integration process, code can become so large (50,000 lines of code or more). This amount of code can cause the verification to take an unreasonable amount of time. You have two options:

- Stop using Polyspace verification at this stage (you have gained many benefits already).
- Analyze subsets of the code.

Select a Subset of Code

Subdividing a project for verification takes considerably less verification time for the sum of the parts than for the whole project considered in one pass. Consider data flow when you subdivide the code.

Consider two distinct concepts:

- Function entry-points — Function entry-points refer to the Polyspace execution model, because they start concurrently, without assumptions regarding sequence or priority. They represent the beginning of your call tree.
- Data entry-points — Regard lines in the code that acquire data as data entry points.

Example 1

```
int complete_treatment_based_on_x(int input)
{
 thousand of line of computation...
}
```

Example 2

```
void main(void)
{
 int x;
 x = read_sensor();
 y = complete_treatment_based_on_x(x);
}
```

Example 3

```
#define REGISTER_1 (*(int *)0x2002002)
void main(void)
{
 x = REGISTER_1;
 y = complete_treatment_based_on_x(x);
}
```

In each case, the x variable is a data entry point and y is the consequence of such an entry point. y can be formatted data, due to a complex manipulation of x.

Because x is volatile, a probable consequence is that y contains all possible formatted data. You could remove the procedure complete_treatment_based_on_x completely, and let automatic stubbing work. The verification process considers y as potentially taking any value in the full range data.

```
//removed definition of complete_treatment_based_on_x
void main(void)
{
 x = ... // what ever
 y = complete_treatment_based_on_x(x); // now stubbed!
}
```

Typical Examples of Removable Components, According to the Logic of the Data

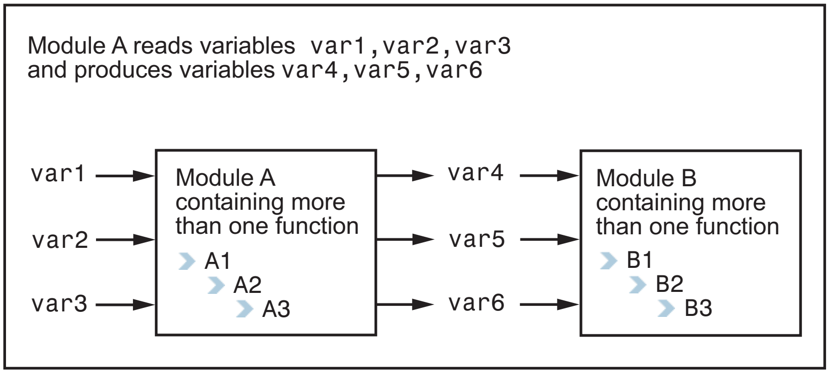
Here are some examples of removable components, based on the logic of the data:

- **Error management modules** often contain a large array of structures accessed through an API, but return only a Boolean value. Removing the API code and retaining the prototype causes the automatically generated stub to return a value in the range [-2^31, 2^31-1], which includes 1 and 0. Polyspace considers the procedure able to return all possible values.
- **Buffer management for mailboxes coming from missing code** – Suppose an application reads a huge buffer of 1024 char. The application then uses the buffer to populate three small arrays of data, using a complicated algorithm before passing it to the main module. If the verification excludes the buffer, and initializes the arrays with random values instead, then the verification of the remaining code is just the same.

- Display modules

Subdivision According to Data Flow

Consider the following example.


Module A reads variables var1,var2,var3
and produces variables var4,var5,var6
var1
var2
var3
Module A containing more than one function
A1
A2
A3
var4
var5
var6
Module B containing more than one function
B1
B2
B3

In this application, `var1`, `var2`, and `var3` can vary between the following ranges:

`var1`	From 0 through 10
`var2`	From 1 through 100
`var3`	From –10 through 10

Module A consists of an algorithm that interpolates between `var1` and `var2`. That algorithm uses `var3` as an exponential factor, so when `var1` is equal to 0, the result in `var4` is also equal to 0.

As a result, `var4`, `var5`, and `var6` have the following specifications:

Ranges	`var4var5var6`	Between –60 and 110From 0 through 12From 0 through 100
Properties	And a set of properties between variables	• If `var2` is equal to 0, then `var4 > var5 > 5`. • If `var3` is greater than 4, then `var4 < var5 < 12` • ...

Subdivision in accordance with data flow allows you to analyze modules `A` and `B` separately:

- `A` uses `var1`, `var2`, and `var3`, initialized respectively to `[0;10]`, `[1;100]`, and `[-10;10]`.
- `B` uses `var4`, `var5`, and `var6`, initialized respectively to `[-60;110]`, `[0;12]`, and `[-10;10]`.

The consequences are:

- A slight loss of precision on the `B` module verification, because now Polyspace considers all combinations for `var4`, `var5`, and `var6`. It includes all possible combinations, even those combinations that the module `A` verification restricts.

  For example, if the `B` module included the test

  ```
  If var2 is equal to 0, then var4 > var5 > 5
  ```

  then the dead code on any subsequent `else` clause is undetected.
- An in-depth investigation of the code is not required to isolate a meaningful subset. It means that a logical split is possible for an application, in accordance with the logic of the data.
- The results remain valid, because there is no requirement to remove, for example, a thread that changes shared data.
- The code is less complex.
- You can maintain the maximum precision level.

Typical examples of removable components:

- Error management modules. A function `has_an_error_already_occurred` can return `TRUE` or `FALSE`. Such a module can contain a large array of structures accessed through an API. Removing API code with the retention of the prototype results in the Polyspace verification producing a stub that returns `[-2^31, 2^31-1]`. That result clearly includes 1 and 0 (yes and no). The procedure `has_an_error_already_occurred` returns all possible values.
- Buffer management for mailboxes coming from missing code. Suppose the code reads a large buffer of 1024 char and then collates the data into three small arrays of data, using a complicated algorithm. It then gives this data to a main module for treatment. For the verification, Polyspace can remove the buffer and initialize the arrays with random values.

- Display modules.

Subdivide According to Real-Time Characteristics

Another way to split an application is to isolate files which contain only a subset of tasks, and to analyze each subset separately.

If a verification initiates using only a few tasks, Polyspace loses information regarding the interaction between variables.

Suppose an application involves tasks `T1` and `T2`, and variable `x`.

If `T1` modifies `x` and reads it at a particular moment, the values of `x` affect subsequent operations in `T2`.

For example, consider that `T1` can write either 10 or 12 into `x` and that `T2` can both write 15 into `x` and read the value of `x`. Two ways to achieve a sound standalone verification of `T2` are:

- You could declare `x` as volatile to take into account all possible executions. Otherwise, `x` takes only its initial value or `x` variable remains constant, and verification of `T2` is a subset of possible execution paths. You can get precise results, but it includes one scenario among all possible states for the variable `x`.
- You could initialize `x` to the whole possible range `[10;15]`, and then call the `T2` entry-point. Use this approach if `x` is calibration data.

Subdivide According to Files

This method is simple, but it can produce good results when you are trying to find defects in gray code.

Simply extract a subset of files and perform a verification using one of these approaches:

- Use entry points.
- Create a `main` that calls randomly functions that the subset of the code does not call.

Reduce Procedure Complexity

If the log file does not display any messages for several hours, you probably have a scaling issue. You can reduce the complexity of some of the procedures by cloning the calling context for specific procedures. One way to reduce complexity is to specify the -inline option on procedures whose names appear in the log file in one or both of two lists.

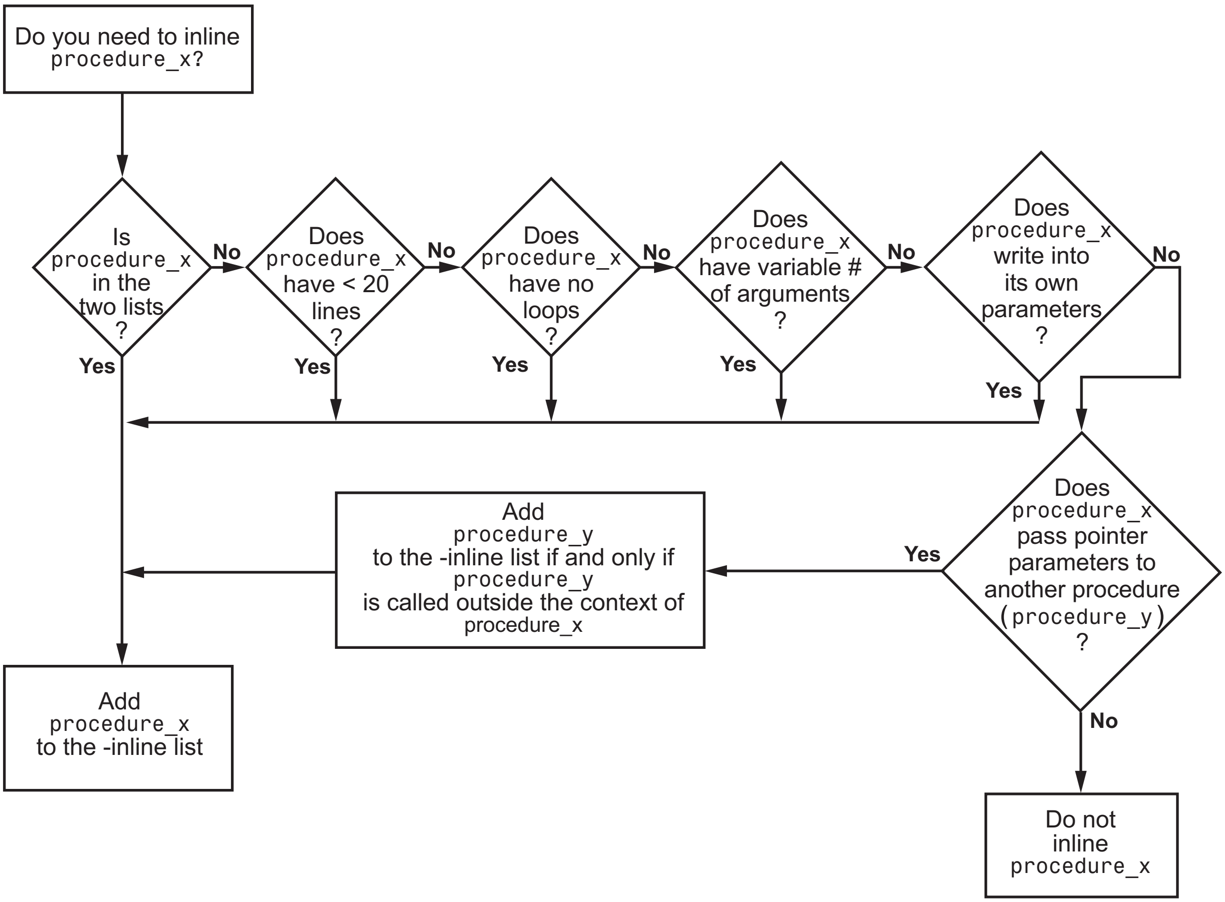
The -inline option creates clones of each specified procedure for each call to it. This option reduces the number of aliases in a procedure, and can improve precision in some situations.

Suppose that the log file contains two lists that look like the following:

```
%%% BEGIN PRE%%%
* inlining procedure_1 could decrease the number of aliases of parameter #3 from 752
                                                                                to 3
* inlining procedure_2 could decrease the number of aliases of parameter #3 from 2687
                                                                                to 3
* inlining procedure_3 could decrease the number of aliases of parameter #4 from 1542
                                                                                to 4
…
%%%END PRE%%%

%%% BEGIN PRE%%%
…
procedures that write the biggest sets of aliases: procedure_4  (2442),
                                                          procedure_2 (1120),
procedure_5 (500)
…
%%%END PRE%%%
```

Looking at this example log file, procedure_1 through procedure_5 are good candidates to be inlined.

Follow the steps on this flow chart to determine which procedure_*x* must be inlined, that is, for which procedure_*x* you need to specify the -inline option.


Do you need to inline procedure_x?
Is procedure_x in the two lists ?
No
Does procedure_x have < 20 lines ?
No
Does procedure_x have no loops ?
No
Does procedure_x have variable # of arguments ?
No
Does procedure_x write into its own parameters ?
No
Yes
Yes
Yes
Yes
Yes
Does procedure_x pass pointer parameters to another procedure (procedure_y) ?
Yes
Add procedure_y to the -inline list if and only if procedure_y is called outside the context of procedure_x
No
Add procedure_x to the -inline list
Do not inline procedure_x

Here are three example situations:

- Using the preceding log file, inline `procedure_2` because it appears in both lists. In addition, if it has no loops, inline `procedure_5`.
- Inline procedures that have a variable number of arguments, such as `printf` and `sprintf`.
- In the following examples, consider whether each procedure, `procedure_x`, passes its pointer parameters to another procedure.

Does this procedure pass pointer parameters?		
Yes	No	No

`void procedure_x(int *p)`
`{`
` procedure_y(p)`
`}`
	`void procedure_x(int q)`	`void procedure_x(int *r)`
		`{`
		` *r = 12`
		`}`

Exercise caution when you inline procedures. Inlining duplicates code and can drastically increase the number of lines of code, resulting in increased computation time.

For example, suppose `procedure_2` has 30 lines of codes and is called 30 times; `procedure_5` has 100 lines of code and is called 50 times. The number of lines of code becomes more than 5000 lines, so computation time increases.

Reduce Task Complexity

If the code contains two or more tasks, and particularly if there are more than 10,000 alias reads, set the option **Reduce task complexity** (`-lightweight-thread-model`). This option reduces:

- Task complexity
- Verification time

However, using this option causes more oranges and a loss of precision on reads of shared variables through pointers.

Reduce Variable Complexity

Variable Characteristic	Action
The types are complex.	Set the `-k-limiting [0-2]` option. Begin with 0. Go up to 1, or 2 in order to gain precision.
There are large arrays	Set the `-no-fold` option.

Choose Lower Precision

The amount of simplification applied to the data representations depends on the required precision level (O0, O2), Polyspace software adjusts the level of simplification. For example:

- `-O0` — shorter computation time
- `-O2` — less orange warnings
- `-O3` — less orange warnings and longer computation time. Use this option for projects containing less than 1,000 lines of code.

Allow Long Verification Time

If the verification does not progress sufficiently within a certain time, it stops. For complex code, if the verification stops at a level lower than what you selected, to allow the verification to continue to the selected level, use the command-line option `-no-inactivity-monitoring`.

To enter command-line options in the user interface:

1. On the **Configuration** pane, select the **Advanced Settings** node.
2. In the **Other** field, enter your command-line option.

Note: This option prevents the verification from stopping at a lower level. However, verification at higher levels can take an extremely long time. If the verification does not progress significantly after 2 days, stop the verification and try alternate ways of reducing the verification time. For more information, see:

- "Run File-by-File Local Verification"
- "Modularize Project Automatically"
- "Troubleshooting in Polyspace Code Prover"

Storage of Temporary Files

By default, Polyspace uses the standard `/tmp` or `C:\Temp` folder to store temporary files. If you do not have write permissions for your temporary folder, you can encounter the error, `Unable to create folder "C:\Temp\Polyspace\`*`foldername`*. There are two possible solutions to this error:

- Change the permissions of your standard temporary folder so you have full read and write privileges.
- Specify the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder. Using this option may affect processing speed if the results folder is mounted on a network drive. Use this option only when the temporary folder partition is not large enough and troubleshooting is required. You can specify `-tmp-dir-in-results-dir` through a line command or the **Advanced Settings** > **Other** field.

Reviewing Verification Results

Review Red Checks

During verification, Polyspace Code Prover checks each operation in your code for certain run-time errors. After verification, the software displays the checks on the **Results Summary** pane.

A red check indicates that the operation fails the check on all execution paths. For instance, a red **Division by Zero** check on a division operation indicates that a division by zero occurs every time the operation takes place. Therefore, you must fix the code containing a red check.

For details of the steps for each check type, see “Check Types”.

In this section...

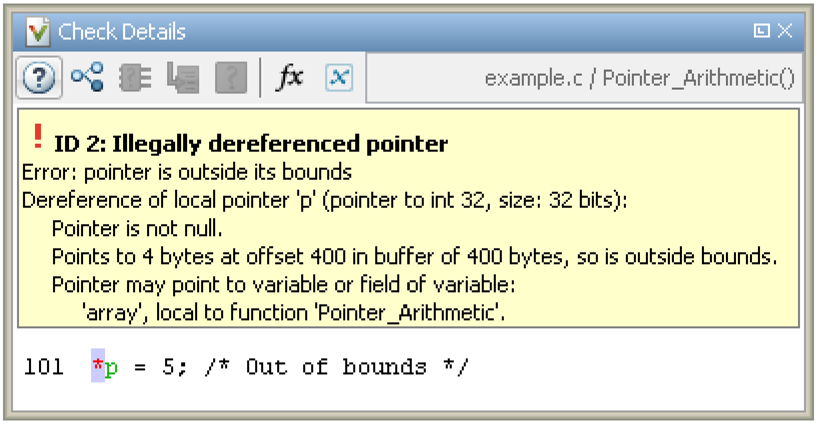
Step 1: Interpret Check Information

1 Select a check on the **Results Summary** pane.

  - On the **Check Details** pane, view further information about the check.
  - On the **Source** pane, the operation containing the check is highlighted.

    If you place your cursor on the operation, the tooltip provides further information about the check.

  Sometimes, this information is sufficient to understand the root cause of the check. If you can determine a fix for your code from this information, you do not have to proceed further with this procedure.

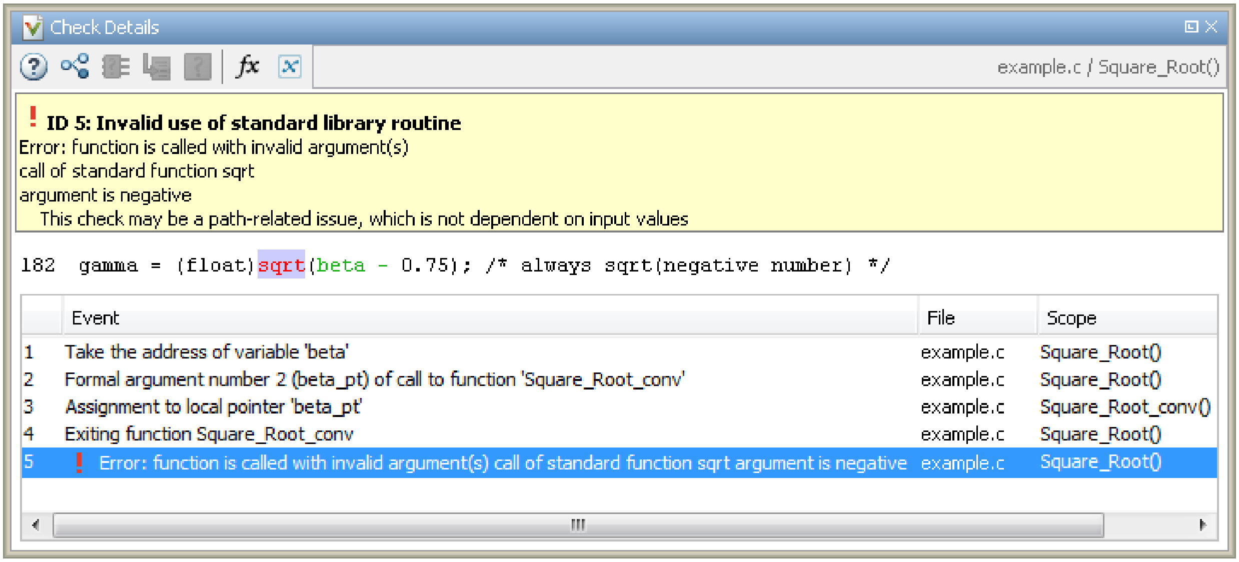


ID 2: Illegally dereferenced pointer
Error: pointer is outside its bounds
Dereference of local pointer 'p' (pointer to int 32, size: 32 bits):
  Pointer is not null.
  Points to 4 bytes at offset 400 in buffer of 400 bytes, so is outside bounds.
  Pointer may point to variable or field of variable:
    'array', local to function 'Pointer_Arithmetic'.

```
101  *p = 5; /* Out of bounds */
```

For details on the information available for each check type, see “Check Types”.

2 Sometimes, the **Check Details** pane lists the sequence of instructions that led to the check. If you see this sequence, select each instruction to trace back to the root cause of the check in your source code.



ID 5: Invalid use of standard library routine
Error: function is called with invalid argument(s)
call of standard function sqrt
argument is negative
  This check may be a path-related issue, which is not dependent on input values

```
182  gamma = (float)sqrt(beta - 0.75); /* always sqrt(negative number) */
```

	Event	File	Scope
1	Take the address of variable 'beta'	example.c	Square_Root()
2	Formal argument number 2 (beta_pt) of call to function 'Square_Root_conv'	example.c	Square_Root()
3	Assignment to local pointer 'beta_pt'	example.c	Square_Root_conv()
4	Exiting function Square_Root_conv	example.c	Square_Root()
5	! Error: function is called with invalid argument(s) call of standard function sqrt argument is negative	example.c	Square_Root()

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, using navigation shortcuts in the user interface, navigate to the root cause.

For details on the navigation process for each type of check, see “Check Types”. The high-level workflow is:

1. Using the tooltips on variables or operations, identify the variable `var` that causes the check. For instance, for an **Out of bound array index** error, `var` can be the array index.
2. Trace the data flow for `var`.

   a. Browse through the previous instances of `var`. On the **Source** pane, place your cursor on each instance of `var` to see its values.

      Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. i Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. ii On the **Search** pane, select the previous instances. • Browse the source code. i Double-click the variable on the **Source** pane. All instances of the variable are highlighted. ii Scroll up to find the previous instances.
Global Variable	i Select the option **Show In Variable Access View**.

Variable	How to Find Previous Instances of Variable
Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	On the **Variable Access** pane, the current instance of the variable is shown. **ii** On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◂ and read operations with ▸.

Function argument
`void func(..,int arg,`
`.`
`.`
`}`
	i On the **Check Details** pane, select the *fx* button.
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◂.
	ii Select a calling function name. You go to the call to `func` in your source.
	iii Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Function return value `ret=func();`	**i** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. **ii** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

b Find the instance where `var` acquires the value that can cause the run-time error.

3 If `var` obtains values from another variable, trace the data flow for the second variable.

Continue this process until you identify the root cause of the check.

For details on the navigation process for each check type, see “Check Types”.

Step 3: Look for Common Causes of Check

Try to methodically determine the root cause for each check. Polyspace Code Prover lists the vulnerabilities in your code. Unless you methodically address these code vulnerabilities, you can miss possible run-time errors.

If you are aware of typical coding errors that cause a red or orange check, root cause investigation is often easier. You can browse through your source code to look for those errors.

For details on the common coding errors for each check type, see “Check Types”.

Related Examples

- “Review Gray Checks”
- “Review Orange Checks”
- “Review Coding Rule Violations”
- “Review Code Metrics”
- “Review Global Variable Usage”

More About

- “Results Management”
- “Result Views in Polyspace User Interface”

Review Gray Checks

Gray checks indicate code that cannot be reached during run-time. Polyspace Code Prover runs three checks for unreachable code:

- Unreachable code
- Function not called

  This check is not turned on by default.
- Function not reachable

  This check is not turned on by default.

If the gray check indicates defensive code, ignore the check. For instance, you can have error handling tests in your code. If the errors do not occur, the test blocks appear gray. However, you might want to retain the error handling test.

In some cases, unreachable code results from coding errors. Therefore, you must review the gray checks. Also, if you do not want to retain unnecessary code, review and fix gray checks.

Note: Following a red check, Polyspace does not verify the remaining code in the same scope as the check. However, this code does not appear gray on the **Source** pane.

Review and fix the red checks so that Polyspace can verify the remaining code. For more information, see “Review Red Checks”.

---

1. After verification, see the code coverage metrics on the **Dashboard** pane.

   The coverage metrics are displayed through the **Code covered by verification** graph. The graph displays:

   - Percentage of code covered by verification.
   - Percentage of functions or procedures covered by verification.
2. If the percentage of functions covered is less than 100, investigate why there are unreachable functions.

   - Select the column graph to see a list of unreachable functions.

- If you want to justify uncalled and unreachable functions, rerun verification using appropriate values for the option **Detect uncalled functions**. The uncalled and unreachable functions appear as gray checks on the **Results Summary** pane.

  For more information, see “Detect uncalled functions (C/C++)”.

3. Investigate the checks further. For more information, see:

   - “Review and Fix Unreachable Code Checks”
   - “Review and Fix Function Not Called Checks”
   - “Review and Fix Function Not Reachable Checks”

4. If you determine that the check represents defensive code, ignore the check. Add a comment and justification in your result or code explaining the rationale.

   For more information, see:

   - “Add Review Comments to Results”
   - “Add Review Comments to Code”

Related Examples

- “Review Red Checks”
- “Review Orange Checks”
- “Review Coding Rule Violations”
- “Review Code Metrics”
- “Review Global Variable Usage”

More About

- “Results Management”
- “Result Views in Polyspace User Interface”

Review Orange Checks

During verification, Polyspace Code Prover checks each operation in your code for certain run-time errors. After verification, the software displays the checks on the **Results Summary** pane.

An orange check indicates that the operation fails the check only on certain execution paths. For more information, see “Sources of Orange Checks”.

Investigate whether the execution paths can occur during run time. If you determine that the execution paths can occur, you must fix the code containing the check.

For details of the steps for each check type, see “Check Types”.

In this section...

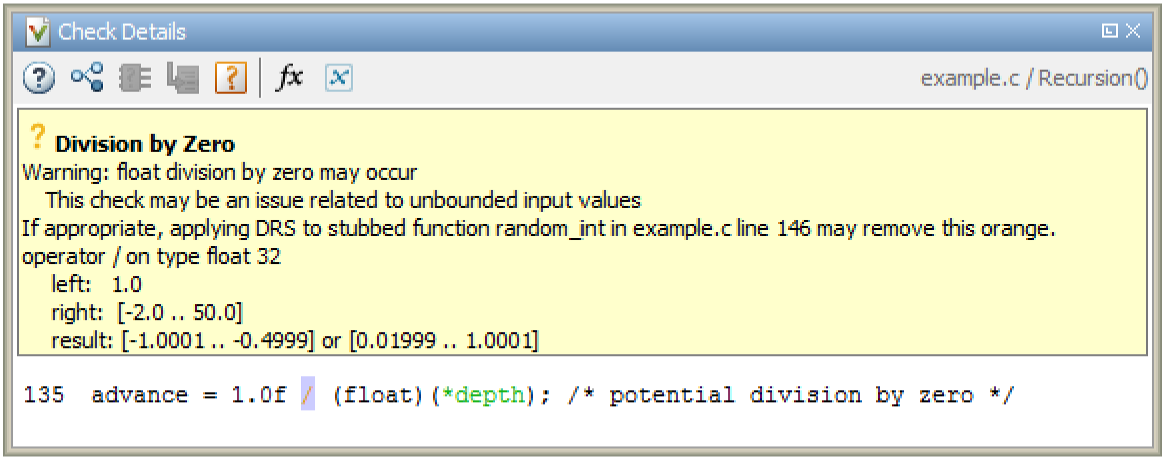
Step 1: Interpret Check Information

1 Select a check on the **Results Summary** pane.

   - On the **Check Details** pane, view further information about the check.
   - On the **Source** pane, the operation containing the check is highlighted.

     If you place your cursor on the operation, the tooltip provides further information about the check.

   Sometimes, this information is sufficient to understand the root cause of the check. If you can determine a fix for your code from this information, you do not have to proceed further with this procedure.



Division by Zero

Warning: float division by zero may occur

This check may be an issue related to unbounded input values

If appropriate, applying DRS to stubbed function random_int in example.c line 146 may remove this orange.

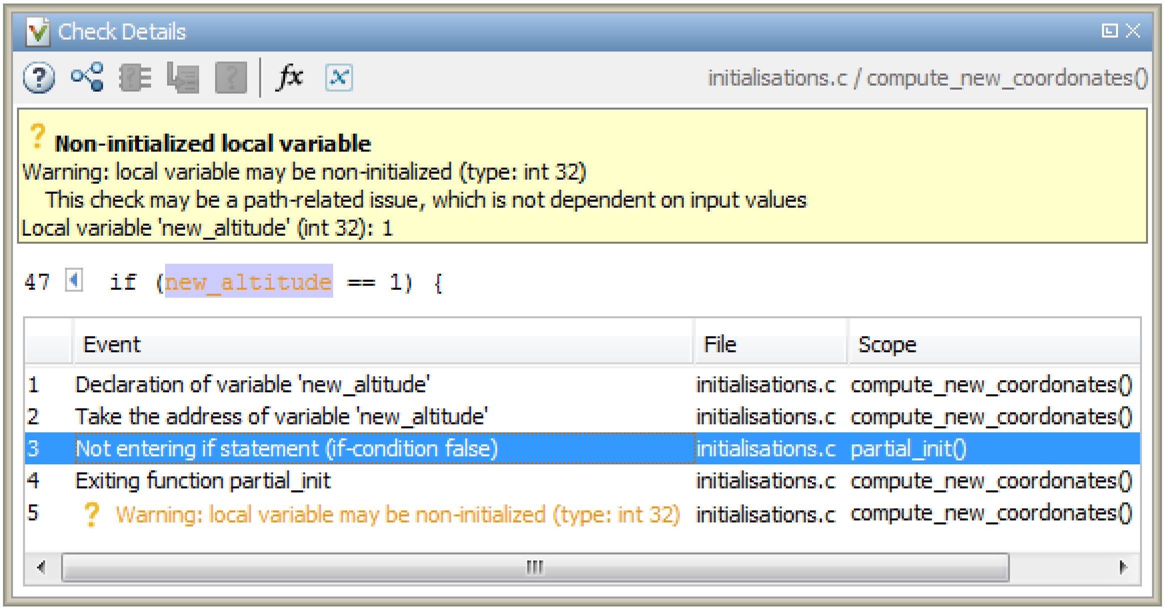
operator / on type float 32

```
    left:   1.0
    right:  [-2.0 .. 50.0]
    result: [-1.0001 .. -0.4999] or [0.01999 .. 1.0001]
```

```
135  advance = 1.0f / (float)(*depth); /* potential division by zero */
```

For details on the information available for each check type, see "Check Types".

2 Sometimes, the **Check Details** pane lists the sequence of instructions that led to the check. If you see this sequence, select each instruction to trace back to the root cause of the check in your source code.



initialisations.c / compute_new_coordonates()

? Non-initialized local variable
Warning: local variable may be non-initialized (type: int 32)
This check may be a path-related issue, which is not dependent on input values
Local variable 'new_altitude' (int 32): 1

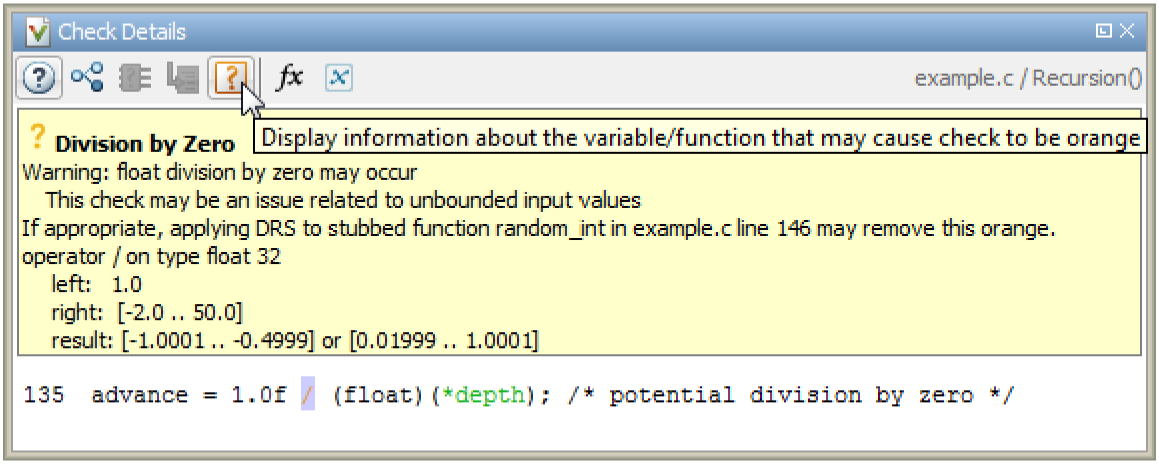
```
if (new_altitude == 1) {
```

	Event	File	Scope
1	Declaration of variable 'new_altitude'	initialisations.c	compute_new_coordonates()
2	Take the address of variable 'new_altitude'	initialisations.c	compute_new_coordonates()
3	Not entering if statement (if-condition false)	initialisations.c	partial_init()
4	Exiting function partial_init	initialisations.c	compute_new_coordonates()
5	? Warning: local variable may be non-initialized (type: int 32)	initialisations.c	compute_new_coordonates()

3 Sometimes, the **Check Details** pane or tooltip shows you the probable cause for the check.

- If a line number is specified for the probable cause, right-click in the **Source** pane. Select **Go To Line**. Enter the line number.
- If the probable cause is an undefined function, click the ? icon. On the **Orange Sources** pane, you can specify constraints on the arguments and return values of the function.

  For more information, see “Constrain Function Arguments and Return Values”.


Check Details
example.c / Recursion()
Display information about the variable/function that may cause check to be orange

? **Division by Zero**
Warning: float division by zero may occur
This check may be an issue related to unbounded input values
If appropriate, applying DRS to stubbed function random_int in example.c line 146 may remove this orange.
operator / on type float 32
left: 1.0
right: [-2.0 .. 50.0]
result: [-1.0001 .. -0.4999] or [0.01999 .. 1.0001]

```
135  advance = 1.0f / (float)(*depth); /* potential division by zero */
```

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, using navigation shortcuts in the user interface, navigate to the root cause.

For details on the navigation process for each type of check, see “Check Types”. The high-level workflow is:

1. Using the tooltips on variables or operations, identify the variable `var` that causes the check. For instance, for an **Out of bound array index** error, `var` can be the array index.
2. Trace the data flow for `var`.

   a. Browse through the previous instances of `var`. On the **Source** pane, place your cursor on each instance of `var` to see its values.

      Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable.

Variable	How to Find Previous Instances of Variable
	i Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. ii On the **Search** pane, select the previous instances. • Browse the source code. i Double-click the variable on the **Source** pane. All instances of the variable are highlighted. ii Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	i Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. ii On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.

Function argument
`void func(..,int arg,`
`.`
`.`
`}`
	i On the **Check Details** pane, select the fx button.
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	ii Select a calling function name. You go to the call to `func` in your source.
	iii Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	**i** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. **ii** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

b Find the instance where `var` acquires the value that can cause the run-time error.

3 If `var` obtains values from another variable, trace the data flow for the second variable.

Continue this process until you identify the root cause of the check.

For details on the navigation process for each check type, see “Check Types”.

4 For orange checks, you have additional tools that help with root cause investigation:

- If a function is called several times and an error occurs only on certain calls, you can identify which function call caused the check.

  For a tutorial, see “Identify Function Call Causing Orange Check”.
- You can run dynamic tests to see if an orange check contains a run-time error.

  However, dynamic testing cannot guarantee the absence of errors. If a test failure occurs, the orange check definitely contains a run-time error. However, if a failure does not occur, the orange check can still contain a run-time error.

  For a tutorial, see “Test Orange Checks for Run-Time Errors”.

  The orange does not support certain code constructs. See .“Limitations of Automatic Orange Tester”

Step 3: Look for Common Causes of Check

Try to methodically determine the root cause for each check. Polyspace Code Prover lists the vulnerabilities in your code. Unless you methodically address these code vulnerabilities, you can miss possible run-time errors.

If you are aware of typical coding errors that cause a red or orange check, root cause investigation is often easier. You can browse through your source code to look for those errors.

For details on the common coding errors for each check type, see “Check Types”.

Step 4: Trace Check to Polyspace Assumption

If you cannot determine a coding error, try to trace the check to a Polyspace assumption earlier in the code. For a list of assumptions, see “Polyspace Software Assumptions”.

If the assumption is broader than what you expect, do one of the following:

- If you can use an analysis option to relax the assumption, rerun verification using that option.

  In particular, determine if you must specify constraints outside your code or provide other contextual information. See “Provide Context for Verification”.
- See if you can improve your coding design to avoid the assumption.

  For instance, `goto` statements interrupt the flow and can cause orange checks during verification. Avoid `goto` statements in your code.

  To improve your coding design:

  - Enforce limits on code complexity metrics. See “Review Code Metrics”.
  - Observe coding rules. See “Follow Coding Rules”.
- If possible, try running verification with higher precision. See “Improve Verification Precision”.
- Ignore the orange check. Add a comment and justification in your result or code describing why you ignored the check. See “Add Review Comments to Results” or “Add Review Comments to Code”.

Related Examples

- “Review Red Checks”
- “Review Gray Checks”
- “Review Coding Rule Violations”
- “Review Code Metrics”

- “Review Global Variable Usage”

More About

- “Results Management”
- “Result Views in Polyspace User Interface”

Review Code Metrics

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

- **User interface**: On the **Configuration** pane, select **Advanced Settings**. Select **Calculate Code Metrics**.
- **Command line**: Use the option `-code-metrics` with the `polyspace-code-prover-nodesktop` command.

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

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

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

  See “Impose Limits on Metrics” on page 9-17.
- Justify the value of a metric.

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

  See “Comment and Justify Limit Violations” on page 9-20.

For information on the metrics, see “Code Metrics”.

In this section...

Impose Limits on Metrics

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

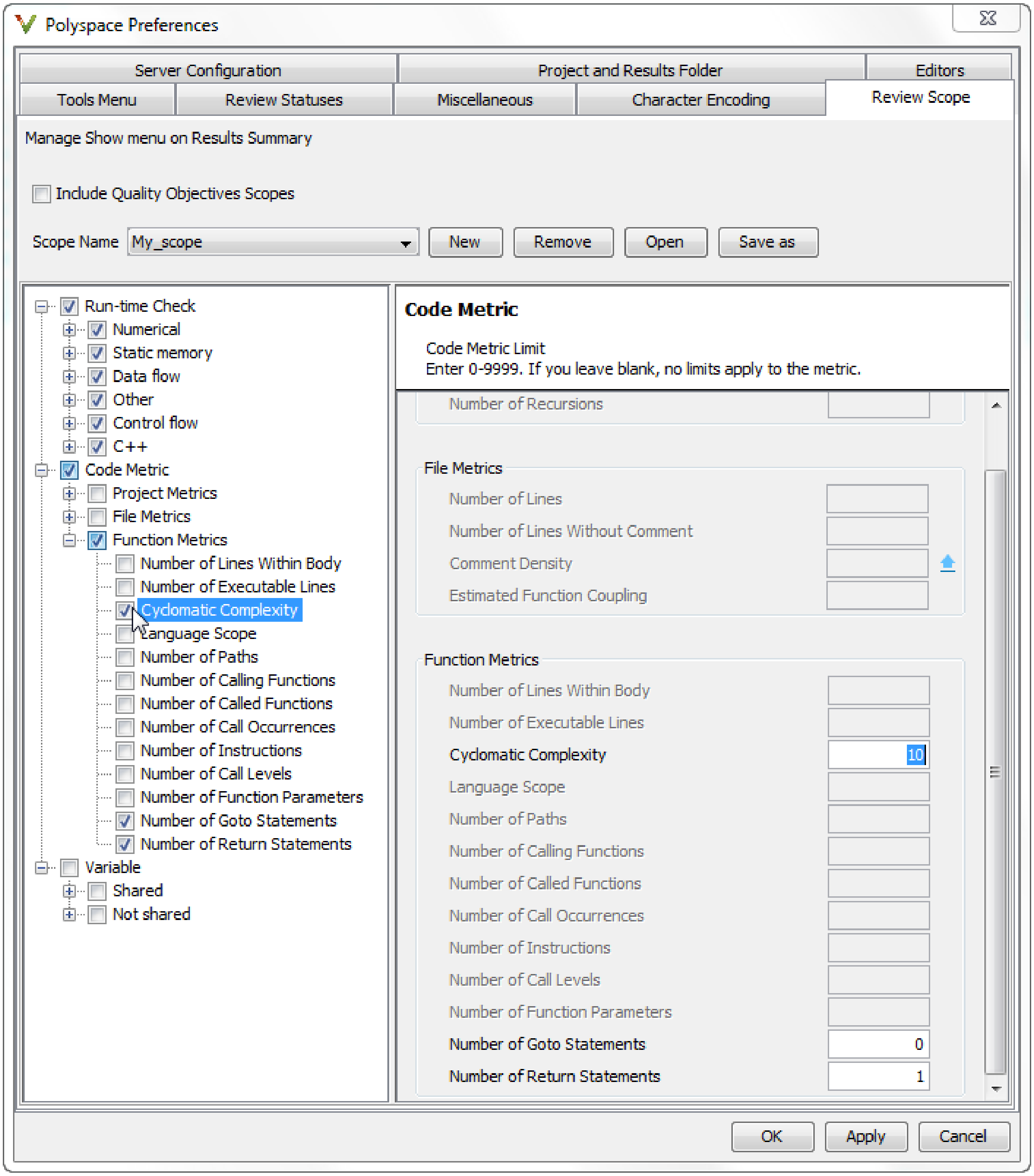
   - To use predefined limits, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows additional options, `HIS`, `SQO-4`, `SQO-5` and `SQO-6`. Select an option to see the limit values.

All the options impose the same limits on code metrics. The option `HIS` displays code metrics only. The other options display other results too and impose different limits on display of orange checks. For more information, see “Limit Display of Orange Checks”. For a detailed explanation of the predefined limits, see “Predefined Software Quality Objectives”.

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

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



Manage Show menu on Results Summary

□ Include Quality Objectives Scopes

Scope Name: My_scope

- ☑ Run-time Check
  - ☑ Numerical
  - ☑ Static memory
  - ☑ Data flow
  - ☑ Other
  - ☑ Control flow
  - ☑ C++
- ☑ Code Metric
  - □ Project Metrics
  - □ File Metrics
  - ☑ Function Metrics
    - □ Number of Lines Within Body
    - □ Number of Executable Lines
    - ☑ Cyclomatic Complexity
    - □ Language Scope
    - □ Number of Paths
    - □ Number of Calling Functions
    - □ Number of Called Functions
    - □ Number of Call Occurrences
    - □ Number of Instructions
    - □ Number of Call Levels
    - □ Number of Function Parameters
    - ☑ Number of Goto Statements
    - ☑ Number of Return Statements
- □ Variable
  - □ Shared
  - □ Not shared

Code Metric

Code Metric Limit
Enter 0-9999. If you leave blank, no limits apply to the metric.

Number of Recursions	

File Metrics

Number of Lines	
Number of Lines Without Comment	
Comment Density	
Estimated Function Coupling	

Function Metrics

Number of Lines Within Body	
Number of Executable Lines	
Cyclomatic Complexity	10
Language Scope	
Number of Paths	
Number of Calling Functions	
Number of Called Functions	
Number of Call Occurrences	
Number of Instructions	
Number of Call Levels	
Number of Function Parameters	
Number of Goto Statements	0
Number of Return Statements	1

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

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

   - If you use predefined limits, the options `HIS`, `SQO-4`, `SQO-5` and `SQO-6` appear.
   - If you define your own limits, the option corresponding to your limits file name appears.
4. Select the option corresponding to the limits that you want. Only metric values that violate your limits appear on the **Results Summary** pane.

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

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

Comment and Justify Limit Violations

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

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

   The code metrics appear together under one node.
2. Expand the node. Select each violation.

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

     For more detailed descriptions and examples, select the ? icon.
3. On the **Results Summary** pane, add a comment and justification describing why the violation occurs. For more information, see “Add Review Comments to Results”.

Related Examples

- “Review Red Checks”

- “Review Gray Checks”
- “Review Orange Checks”
- “Review Coding Rule Violations”
- “Review Global Variable Usage”

More About

- “Results Management”
- “Result Views in Polyspace User Interface”

Review Global Variable Usage

After verification, Polyspace Code Prover displays a list of global variables in your source code. Using this list:

- You can remove variables that you define but do not use.

  Such variables appear gray on the **Results Summary** and **Source** pane.
- For code intended for multitasking, you can see which variables are not protected from concurrent access by multiple tasks.
  - If Polyspace Code Prover proves that a variable is protected, it appears green on the **Results Summary** and **Source** pane.
  - Otherwise, it appears orange.

For more information, see “Global Variables”.

To review global variable usage:

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

   The global variables appear together under one node.
2. Expand the **Global Variable** node. Review each result under the nodes:
   - **Shared** > **Potentially unprotected variable**.
   - **Not shared** > **Unused non-shared variable**.
3. For each potentially unprotected variable, select the variable name.

   a. On the **Check Details** pane, view which tasks can access the variable. Click the button to view a graphical representation of the access.

   b. All read and write operations on the variable appear on this pane. Select each operation to navigate to it in your source code.

      This action also displays more details of the operation on the **Variable Access** pane.

   c. To review your multitasking options, select **Window** > **Show/Hide View** > **Configuration**.

Identify whether you can leverage some of the existing protection mechanisms to protect your variable. For more information on multitasking verification, see “Multitasking”.

Related Examples

- “Review Red Checks”
- “Review Gray Checks”
- “Review Orange Checks”
- “Review Coding Rule Violations”
- “Review Code Metrics”

More About

- “Results Management”
- “Result Views in Polyspace User Interface”

Add Review Comments to Results

This example shows how to comment on results in the Polyspace user interface. When reviewing results, you can assign a status to them, and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same result twice.

In this section...
"Assign and Save Comments" on page 9-24
"Import Review Comments from Previous Verifications" on page 9-25

Assign and Save Comments

1. On the **Results Summary** pane, select the result that you want to review.
2. Investigate the result further.

   For more information, see:

   - "Review Red Checks"
   - "Review Gray Checks"
   - "Review Orange Checks"
   - "Review Code Metrics"
   - "Review Global Variable Usage"
3. On the **Results Summary** or **Check Review** pane, provide the following review information for the result:

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

     To justify the check, select one of the **Status** options, `Justify with annotations` or `No action planned`. You can view the percentage of results justified per file and function. On the **Results Summary** pane, select **Group by** > **File**. View the entries on the **Justified** column.

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

  - **Comment** to describe any other information about the result.

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

   To select the results in a group:

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

     To group together results belonging to a certain category, click the **Check** column header on the **Results Summary** pane.
   - If the results are not contiguous, **Ctrl**-left click each result.
   - If the results belong to the same category and have the same color, right-click one result. From the context menu, select **Select All *`Color Type`* Checks**.

     For instance, select **Select All Orange "Illegally dereferenced pointer" Checks**.
5. To save your review comments, select **File** > **Save**. Your comments are saved with the verification results.

Import Review Comments from Previous Verifications

- “Disable Automatic Comment Import from Last Verification” on page 9-25
- “Import Comments from Another Verification Result” on page 9-26
- “View Imported Comments That Do Not Apply” on page 9-26

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

After you import comments, on the **Results Summary** pane, clicking the ⇨ icon skips justified checks. Using this icon, you can browse through unreviewed checks. You can also filter the justified checks from display. See “Filter and Group Results”.

Disable Automatic Comment Import from Last Verification

1. Select **Tools** > **Preferences**, which opens the Polyspace Preferences dialog box.
2. Select the **Project and Results Folder** tab.

3. Under **Import Comments**, clear **Automatically import comments from last verification**.
4. Click **OK**.

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

Import Comments from Another Verification Result

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

   The review comments from the previous results are imported into the current results, and the Import checks and comments report opens. For more information, see "View Imported Comments That Do Not Apply".

View Imported Comments That Do Not Apply

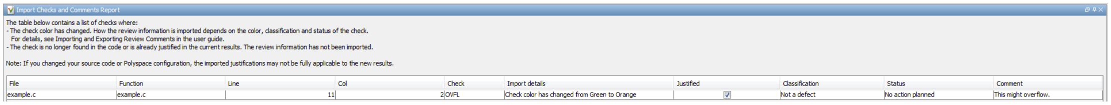
You can directly import review comments from another set of results into the current results. However, it is possible that your review comments do not apply to a subsequent verification because:

- You have changed your source code so that the check is no longer present.
- You have changed your source code so that the check color has changed.
- You have already entered different review comments for the same check.

The Import Checks and Comments Report highlights differences between two verification results. When you import comments from a previous verification, you can see this report. If you have closed the report after an import, to review the report again:

1. Select **Window** > **Show/Hide View** > **Import Comments Report**.

   The Import Checks and Comments Report opens, highlighting differences in the two results.



The table below contains a list of checks where:
- The check color has changed. How the review information is imported depends on the color, classification and status of the check.
  For details, see Importing and Exporting Review Comments in the user guide.
- The check is no longer found in the code or is already justified in the current results. The review information has not been imported.

Note: If you changed your source code or Polyspace configuration, the imported justifications may not be fully applicable to the new results.

File	Function	Line	Col	Check	Import details	Justified	Classification	Status	Comment
example.c	example.c	11	2	OVFL	Check color has changed from Green to Orange	☑	Not a defect	No action planned	This might overflow.

2 Review the differences between the two results.

   - If the check color changes, Polyspace populates the **Comment** field but not the fields **Classification**, **Status** or **Justified**.
   - If a check no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.
   - If you have already entered different review comments for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.

Related Examples

- "Add Review Comments to Code"

More About

- "Result Views in Polyspace User Interface"

Add Review Comments to Code

This example shows how to place review comments in your code for a particular result. If your code comments follow a particular syntax, in a later verification on the same code, Polyspace can read the comments. Using the comments, Polyspace automatically populates the **Classification**, **Status** and **Comment** fields on the **Results Summary** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same result twice.

Tip If you enter review comments in your code, the comments will appear in verification results for all subsequent verifications on that code. Therefore, use code comments for verification results for which you or someone else is not likely to change the code.

In this section...

Enter Code Comments in Specific Syntax

You can manually enter comments in a specific syntax just before the line containing the result.

To comment:

- An individual line of code, use the following syntax:

  ```
  /* polyspace<Type:Kind1[,Kind2] : [Classification] : [Status] >
  [Additional text] */
  ```
- A section of code, use the following syntax:

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

  ... Code section ...

  /* polyspace:end<Type:Kind1[,Kind2] : [Classification] : [Status] > */
  ```

The square brackets `[ ]` indicate optional information.

Replace	Replace with
Type	**Runtime errors:** RTE
	Coding rule violations: • MISRA-C • MISRA-AC-AGC • MISRA-C3 • MISRA-CPP • JSF • Custom
	Global variables: VARIABLE
Kind1,Kind2,...	**Runtime errors:** Acronyms for checks such as ZDV, OVFL, etc.. If you want the comment to apply to all checks on the following line, specify ALL.
	Coding rule violations: Rule number. For more information, see “Coding Rules”. If you want the comment to apply to all coding rule violations on the following line, specify ALL.
	Global variables: ALL. For global variables, the same comment syntax applies irrespective of whether they are shared or used.

Replace	Replace with
`Classification`	Text that indicates the severity of the defect. Enter one of the following: • `Unset` • `High` • `Medium` • `Low` • `Not a defect` This text populates the **Classification** column on the **Results Summary** pane.
`Status`	Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text: • `Fix` • `Improve` • `Investigate` • `Justify with annotations` • `No action planned` • `Restart with different options` • `Other` • `Undecided` This text populates the **Status** column on the **Results Summary** pane.
`Additional text`	Any text. This text populates the **Comment** column on the **Results Summary** pane.

- “Syntax Example: Run-time Checks” on page 9-31
- “Syntax Example: Coding Rule Violations” on page 9-31
- “Syntax Example: Global Variables” on page 9-31

Syntax Example: Run-time Checks

- Non terminating call:

  ```
  /* polyspace<RTE: NTC : Low : No Action Planned > Known issue */
  ```
- Division by zero:

  ```
  /* polyspace<RTE: ZDV : High : Fix > Denominator cannot be zero */
  ```

Syntax Example: Coding Rule Violations

- MISRA C rule violation:

  ```
  /* polyspace<MISRA-C: 6.3 : Low : Justify with annotations> Known issue */
  ```
- JSF C++ rule violation:

  ```
  /* polyspace<JSF: 9 : Low : Justify with annotations> Known issue */
  ```

Syntax Example: Global Variables

```
/* polyspace<VARIABLE: ALL : Low : Justify with annotations> Known issue */
```

Copy Comment Syntax from Polyspace User Interface

Instead of manually entering the comment in a specific syntax, you can copy the comment syntax from the Polyspace user interface and paste in your code.

1. On the **Results Summary** pane, assign a **Classification**, **Status** and **Comment** to a result.

   a. Select the result.
   b. Under the columns, **Classification** and **Status**, select options from the drop down lists.
   c. Under the column **Comment**, enter a comment that helps you recognize the result easily.
2. Copy the **Classification**, **Status** and **Comment**.

   a. On the **Results Summary** pane, right-click the result.
   b. Select **Add Pre-Justification to Clipboard**. The software copies the justification string to the clipboard.
3. Paste the **Classification**, **Status** and **Comment** in your source code.

a On the **Results Summary** pane, right-click the result and select **Open Editor**.

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

b Using the paste option in the text editor, paste the justification template string on the line immediately before the line containing the result.

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

c Save your source file.

4. Run the verification again. Open your results.

   On the **Results Summary** pane, the software populates the **Classification**, **Status** and **Comment** fields for the result. You can either ignore these findings, or filter them from the **Results Summary** pane. For more information on filtering, see “Filter and Group Results”.

For a tutorial, see “Comment Code for Known Defects”.

Related Examples

- “Add Review Comments to Results”

More About

- “Result Views in Polyspace User Interface”

Comment Code for Known Defects

This tutorial shows how to place comments in your code to mark defects that you are already aware of but do not intend to fix immediately. Using your comments, Polyspace populates the defect **Classification**, **Status** and **Comment** fields on the **Results Summary** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same defect twice. The example uses the following code that is stored in a file `divideByDifference.c`.

```
#include <math.h>
int divideByDifference(int num, int x, int y)
{
 if(x >= y)
   return(num/(abs(x)-abs(y)));
  else
   return 0;
}
```

For the high-level workflow, see “Add Review Comments to Code”.

1 Verify Source File and Review Results

1. Create a new Polyspace project. Add the file `divideByDifference.c` to the project.
2. Click ▷ Run to start verification on your project.

   The verification uses the default options on the **Configuration** pane. It uses a generated `main` to call the function, `divideByDifference`.
3. Open the verification results. On the **Results Summary** pane, select:

   - One of the two orange **Invalid use of standard library routine** errors. On the **Check Details** pane, you can see an error message that the orange error on the `abs` functions can be due to unbounded input values.

     Enter the following review information for the error.

Column name	Review Information
Classification	**Not a defect**
Status	**No action planned**

Column name	Review Information
Comment	`Argument of abs is bounded`

- The orange **Division by Zero** error.

  Enter the following review information for the error.

Column name	Review Information
Classification	**High**
Status	**Investigate**
Comment	`To check if x can be equal to y`

2 Comment Code for `STD_LIB` error

1. On the **Results Summary** pane, right-click one of the orange **Invalid use of standard library routine** errors. Select **Add Pre-Justification to Clipboard**.

   This action copies your **Classification**, **Status**, and **Comment** in a form that you can insert in your source code.

2. Using the paste option in your text editor, in the file `divideByDifference.c`, paste what you copied just before the line `return(num/(abs(x)-abs(y)));`.

   Your source code appears as follows:

```
#include <math.h>
int divideByDifference(int num, int x, int y)
{
 if(x >= y)
   /* polyspace<RTE:STD_LIB:Not a defect:No action planned>
Argument of abs is bounded */
   return(num/(abs(x)-abs(y)));
  else
   return 0;
}
```

3. Run the verification again. Open your results.

   On the **Results Summary** pane, both instances of **Invalid use of standard library routine** on the line `return(num/(abs(x)-abs(y)));` have the **Classification**, **Status**, and **Comment** that you entered.

3 Comment Code for OVFL error

1 In the file `divideByDifference.c`, edit the comment that you entered.

Original	Replace with
`STD_LIB`	`STD_LIB,OVFL`
`Not a defect`	`Low`
`Argument of abs is bounded`	`Error does not occur for values of x and y`

2 Run the verification again. Open your results.

On the **Results Summary** pane, the **Overflow** and **Invalid use of standard library routine** checks on the line `return(num/(abs(x)-abs(y)));` have the following review information:

Column name	Review Information
Classification	**Low**
Status	**No action planned**
Comment	`Error does not occur for values of x and y`

4 Comment Code for ZDV error

1 On the **Results Summary** pane, right-click the orange **Division by Zero** error. Select **Add Pre-Justification to Clipboard**.

This action copies your **Classification**, **Status**, and **Comment** in a form that you can insert in your source code.

2 In the file `divideByDifference.c`, paste what you copied after the already existing comment.

Your source code appears as follows:

```
#include <math.h>
int divideByDifference(int num, int x, int y)
{
 if(x >= y)
   /* polyspace<RTE:STD_LIB,OVFL:Not a defect:
No action planned> Error does not occur for values of x and y */
```

```
   /* polyspace<RTE:ZDV:High:Investigate>
To check if x can be equal to y */
   return(num/(abs(x)-abs(y)));
  else
   return 0;
}
```

3 Run the verification again. Open your results.

On the **Results Summary** pane, the **Division by Zero** error on the line `return(num/(abs(x)-abs(y)));` has the **Classification**, **Status**, and **Comment** that you entered. The other errors retain the earlier review information.

Check and Code Metric Acronyms

The following table lists alphabetically the result acronyms that you must use in code comments or custom software quality objectives. For more information on the workflows, see:

- "Add Review Comments to Code"
- "Customize Software Quality Objectives"

In this section...
"Checks" on page 9-37 "Code Complexity Metrics" on page 9-38

Checks

Check	Acronym
Absolute address	`ABS_ADDR`
C++ specific checks	`CPP`
Correctness condition	`COR`
Division by zero	`ZDV`
Exception handling	`EXC`
Function not called	`FNC`
Function not reachable	`FNR`
Function returns a value	`FRV`
Illegally dereferenced pointer	`IDP`
Initialized return value	`IRV`
Inspection points	`IPT`
Invalid use of standard library routine	`STD_LIB`
Non-initialized local variable	`NIVL`
Non-initialized pointer	`NIP`
Non-initialized variable	`NIV`
Non-null this-pointer in method	`NNT`

Check	Acronym
Non-terminating call	NTC
Non-terminating loop	NTL
Object oriented programming	OOP
Out of bounds array index	OBAI
Overflow	OVFL
Shift operations	SHF
Unreachable code	UNR
User assertion	ASRT

Code Complexity Metrics

You cannot add review comments to your code for code metrics. The following acronyms are useful only for defining custom software quality objectives.

Code Metric	Acronym
Comment Density	COMF
Cyclomatic Complexity	VG
Estimated Function Coupling	FCO
Language Scope	VOCF
Number of Call Levels	LEVEL
Number of Call Occurrences	NCALLS
Number of Called Functions	CALLS
Number of Calling Functions	CALLING
Number of Direct Recursions	AP_CG_DIRECT_CYCLE
Number of Executable Lines	FXLN
Number of Files	FILES
Number of Function Parameters	PARAM
Number of Goto Statements	GOTO
Number of Header Files	INCLUDES

Code Metric	Acronym
Number of Instructions	`STMT`
Number of Lines	`TOTAL_LINES`
Number of Lines Within Body	`FLIN`
Number of Lines Without Comment	`LINES_WITHOUT_CMT`
Number of Paths	`PATH`
Number of Protected Shared variables	`PSHV`
Number of Recursions	`AP_CG_CYCLE`
Number of Return Statements	`RETURN`
Number of Unprotected Shared Variables	`UNPSHV`

Result and Source Code Colors

In this section...

Result Colors

Polyspace displays the different verification results with specific icons on the **Results Summary**, **Check Details** and **Check Review** pane.

- **Red checks** – Indicates code that is proven to contain a certain error on all execution paths.

  Polyspace Code Prover verification determines errors with reference to the language standard. Though some of the errors can be acceptable for a particular compilation environment, they violate the language standard. To allow some of the environment-dependent behavior, use appropriate analysis options. For more information, see "Verification Assumptions" and "Check Behavior".

  In the verification results, the software displays red checks with an ! icon.
- **Gray checks** – Indicates unreachable code. Polyspace assigns a lighter gray color to code deactivated due to conditional compilation, for instance in `#ifdef` statements.

  In the verification results, the software displays gray checks with an ✕ icon.
- **Orange checks** – Indicates code that can contain an error on certain execution paths. For more information, see "Sources of Orange Checks".

  In the verification results, the software displays orange checks with an ? icon.
- **Green checks** – Indicates code that is proven to not contain a certain error on all execution paths.

  In the verification results, the software displays green checks with an ✓ icon.
- **Coding rule violations** – Indicates violation of predefined or custom coding rules.

  In the verification results, the software displays coding rule violations with an ▽ icon for predefined rules or ▼ icon for custom rules.
- **Code metrics** – Indicates code complexity metrics.

In the verification results, the software displays code metrics with an ★ icon. If a code metric exceeds the limit that you specified, the software displays the metric with an ! ★ icon.

- **Global variables** – Indicates global variable declaration.

  In the verification results, the software displays global variables with an ⊠ icon. In addition, depending on the nature of the global variable, the software supplements the icon with a check icon. For instance, the software displays shared potentially unprotected global variables with an ? ⊠ icon and non-shared unused global variables with an ✕ ⊠ icon.

Source Code Colors

Polyspace uses the following color scheme for displaying code on the **Source** pane.

- For every check on the **Results Summary** pane, Polyspace assigns the check color to the corresponding section of code.
  - For lines containing macros, if the macro is collapsed, then Polyspace colors the entire line with the color of the most severe check on the line. The severity decreases in this order: red, gray, orange, green.

    If there is no check in a line containing a macro, Polyspace underlines the line in black when the macro is collapsed.
  - For all other lines, Polyspace colors only the keyword or identifier associated with the check.
- For every coding rule violation on the **Results Summary** pane, Polyspace assigns to the corresponding keyword or identifier:
  - A ▽ symbol if the coding rule is a predefined rule. The predefined rules available are MISRA C, MISRA AC AGC, MISRA C++, or JSF C++.
  - A ▼ symbol if the coding rule is a custom rule.
- If a tooltip is available for a keyword or identifier on the **Source** pane, Polyspace:
  - Uses solid underlining for the keyword or identifier if it is associated with a check.
  - Uses dashed underlining for the keyword or identifier if it is not associated with a check.

- When a function is defined, Polyspace colors the function name in blue.
- Polyspace assigns a lighter shade of gray to code deactivated due to conditional compilation. Such code occurs, for instance, in `#ifdef` statements where the macro for a branch is not defined. This code does not affect the verification.

Verification Following Red and Orange Checks

Polyspace considers that all execution paths that contain a run-time error terminate at the location of the error. Therefore:

- Following a red check, Polyspace does not analyze the remaining code in the same scope as the check.
- Following an orange check, Polyspace analyzes the remaining code. But it considers only the subset of execution paths that did not contain the run-time error.

  Exceptions to this behavior can occur. For example, for an orange overflow, if you specify the appropriate **Overflow computation mode**, Polyspace wraps the result of an overflow and does not terminate the execution paths. See "Overflow computation mode (C/C++)".

Use these two rules to understand your checks. The following examples show how the two rules can result in checks that can be misleading when viewed out of context. Understand the examples below thoroughly to practice reviewing checks in context of the remaining code.

In this section...

Code Following Red Check

The following example shows what happens after a red check:

```
void red(void)
{
int x;
x = 1 / x ;
x = x + 1;
}
```

When Polyspace verification reaches the division by `x`, `x` has not yet been initialized. Therefore, the software generates a red `Non-initialized local variable` check for `x`.

Execution paths beyond division by x are stopped. No checks are generated for the statement x = x + 1;.

Green Check Following Orange Check

The following example shows how a green check can result from a previous orange check.

```
extern int Read_An_Input(void);
void propagate(void)
{
 int x;
 int y[100];
 x = Read_An_Input();
 y[x] = 0;
 y[x] = 0;
}
```

In this function:

- x is assigned the return value of Read_An_Input. After this assignment, the software estimates the range of x as [-2^31, 2^31-1].
- The first y[x]=0; shows an Out of bounds array index error because x can have negative values.
- After the first y[x]=0;, from the size of y, the software estimates x to be in the range [0,99].
- The second y[x]=0; shows a green check because x lies in the range [0,99].

Gray Check Following Orange Check

The following example shows how a gray check can result from a previous orange check.

Consider the following example:

```
extern int read_an_input(void);

void main(void)
{
 int x;
 int y[100];
 x = read_an_input();
 y[x] = 0;
 y[x-1] = (1 / x) + x ;
 if (x == 0)
```

```
  y[x] = 1;
}
```

From the gray check, you can trace backwards as follows:

- The line `y[x]=1;` is unreachable.
- Therefore, the test to assess whether `x = 0` is always false.
- The return value of `read_an_input()` is never equal to 0.

However, `read_an_input` can return any value in the full integer range, so this is not the correct explanation.

Instead, consider the execution path leading to the gray code:

- The orange **Out of bounds array index** check on `y[x]=0;` means that subsequent lines deal with `x` in `[0,99]`.
- The orange **Division by Zero** check on the division by `x` means that `x` cannot be equal to 0 on the subsequent lines. Therefore, following that line, `x` is in `[1,99]`.
- Therefore, `x` is never equal to 0 in the `if` condition. Also, the array access through `y[x-1]` shows a green check.

Red Check Following Orange Check

The following example shows how a red error can hide a bug which occurred on previous lines.

```
%% file1.c %%

void f(int);
int read_an_input(void);

int main()  {
    int x,old_x;
    x = read_an_input();
    old_x = x;
    if (x<0 || x>10)
      return 1;
    f(x);
    x = 1 / old_x;
    // Red Division by Zero
    return 0;
}
```

```
%% file2.c %%

#include <math.h>

void f(int a) {
    int tmp;
    tmp = sqrt(0-a);
}
```

A red check occurs on `x=1/old_x;` in `file1.c` because of the following sequence of steps during verification:

1. When `x` is assigned to `old_x` in `file1.c`, the verification assumes that `x` and `old_x` have the full range of an integer, that is `[-2^31 , 2^31-1]`.
2. Following the `if` clause in `file1.c`, `x` is in `[0,10]`. Because `x` and `old_x` are equal, Polyspace considers that `old_x` is in `[0,10]` as well.
3. When `x` is passed to `f` in `file1.c`, the only possible value that `x` can have is 0. All other values lead to a run-time exception in `file2.c`, that is `tmp = sqrt(0—a);`.
4. A red error occurs on `x=1/old_x;` in `file1.c` because the software assumes `old_x` to be 0 as well.

Related Examples

- "Review Red Checks"
- "Review Gray Checks"
- "Review Orange Checks"

Check Types

Polyspace Code Prover checks each operation in your code for certain run-time errors and displays the result as a red, green or orange check. For more information, see “Result and Source Code Colors”.

You must review a red or orange check and determine whether to fix your code. The tables below list the checks that Polyspace Code Prover performs and how you can review them.

In this section...

Data Flow Checks

Check	How to Review
Function not called	“Review and Fix Function Not Called Checks”
Function not reachable	“Review and Fix Function Not Reachable Checks”
Initialized return value	“Review and Fix Initialized Return Value Checks”
Non-initialized local variable	“Review and Fix Non-initialized Local Variable Checks”
Non-initialized pointer	“Review and Fix Non-initialized Pointer Checks”
Non-initialized variable	“Review and Fix Non-initialized Variable Checks”
Unreachable code	“Review and Fix Unreachable Code Checks”

Numerical Checks

Check	How to Review
Division by zero	"Review and Fix Division by Zero Checks"
Overflow	"Review and Fix Overflow Checks"
Shift operations	"Review and Fix Shift Operations Checks"

Static Memory Checks

Check	How to Review
Absolute address	"Review and Fix Absolute Address Checks"
Illegally dereferenced pointer	"Review and Fix Illegally Dereferenced Pointer Checks"
Out of bounds array index	"Review and Fix Out of Bounds Array Index Checks"

Control Flow Checks

Check	How to Review
Non-terminating call	"Review and Fix Non-Terminating Call Checks"
Non-terminating loop	"Review and Fix Non-Terminating Loop Checks"

C++ Checks

- C++ specific checks
- Exception handling
- Function returns a value
- Non-null this-pointer in method
- Object oriented programming

Other Checks

Check	How to Review
Correctness condition	"Review and Fix Correctness Condition Checks"
Invalid use of standard library routine	"Review and Fix Invalid Use of Standard Library Routine Checks"
User assertion	"Review and Fix User Assertion Checks"

Result Views in Polyspace User Interface

In the Polyspace user interface, you can use the following panes to review your results. If a pane is not open by default, to see the pane, select **Window > Show/Hide View >** ***`Pane Name`***.

In this section...

Results Summary

The **Results Summary** pane lists all checks along with their attributes.

For each check, the **Results Summary** pane contains the check attributes, listed in columns:

Attribute	Description
Family	Group to which the check belongs. For instance, if you choose **Group by > File**, this column contains the name of the file and function containing the check.
ID	Unique identification number of the check.
Type	Check color
Category	Category of the check. For more information on the checks covered by a category, see “Run-Time Checks”.
Check	Check name
Information	For orange checks, this column indicates whether the check is related to path or

Attribute	Description
	input values. For more information, see "Critical Orange Checks". For coding rule violations, this column indicates whether the rule belongs to the `Required` subset. For global variables, this column contains the global variable name.
File	File containing the instruction where the check occurs
Class	Class containing the instruction where the check occurs. If the check is not inside a class definition, then this column contains the entry, **Global Scope**.
Function	Function containing the instruction where the check occurs. If the function is a method of a class, it appears in the format `class_name::function_name`.
Line	Line number of the instruction where the check occurs.
Col	Column number of the instruction where the check occurs. The column number is the number of characters from the beginning of the line.
%	Percentage of checks that are not orange. This column is most useful when you choose the option **Group by** > **File**. The entry in this column against a file or function indicates the percentage of checks in the file or function that are not orange.

Attribute	Description
Classification	Level of severity you have assigned to the check. The possible levels are: • `Unset` • `High` • `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the check. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justify with annotations` • `No action planned` • `Other` • `Restart with different options`
Justified	Check boxes showing whether you have justified the checks. To justify a check, you must assign the status `Justify with annotations` or `No action planned`. If you choose the option **Group by** > **File**, this column indicates the percentage of checks that you have justified per file and function.
Comments	Comments you have entered about the check

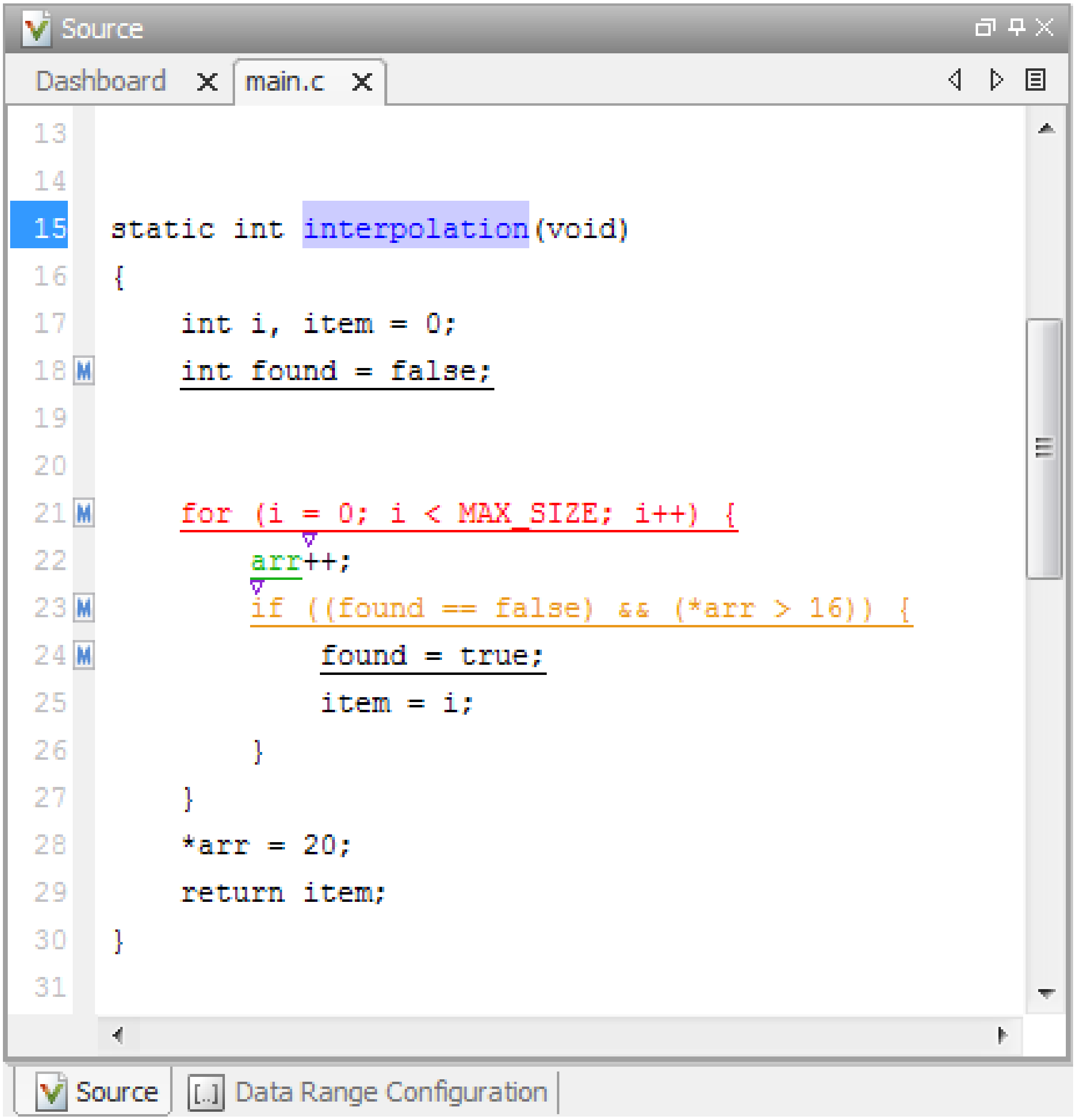
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the checks. For more information, see “Add Review Comments to Results”.
- Organize your check review using filters on the columns. For more information, see “Filter and Group Results”.

Source

The **Source** pane shows the source code with colored checks highlighted.

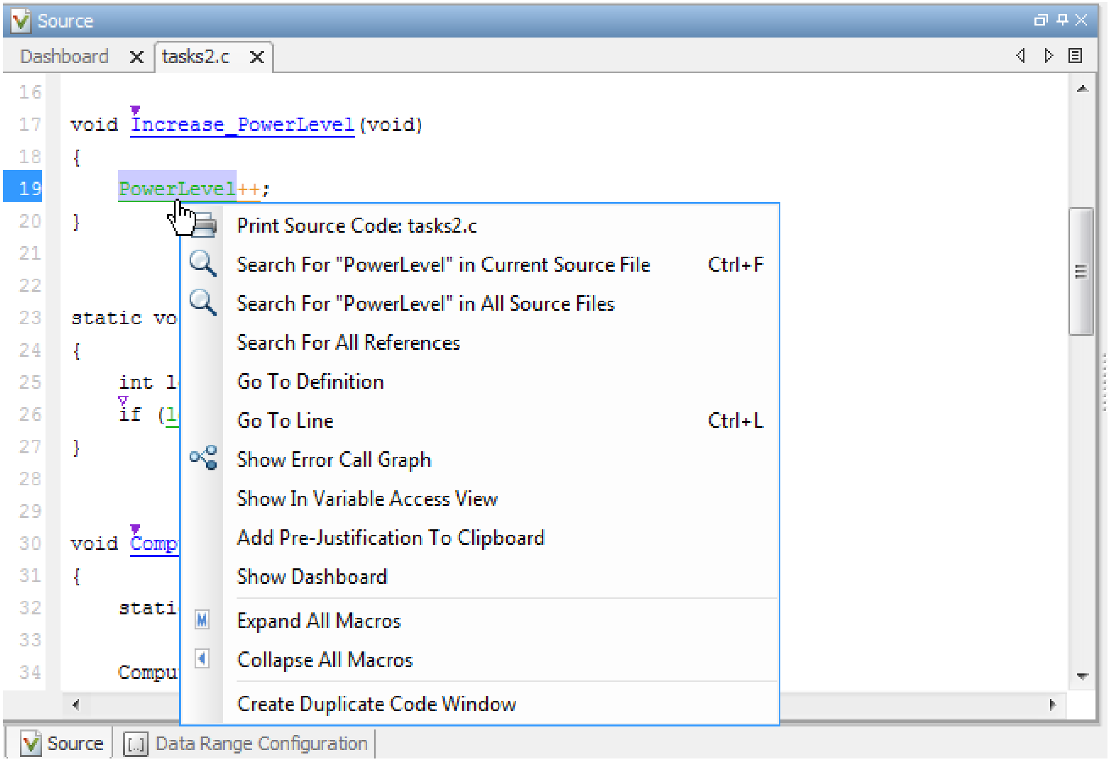


```


static int interpolation(void)
{
    int i, item = 0;
    int found = false;


    for (i = 0; i < MAX_SIZE; i++) {
        arr++;
        if ((found == false) && (*arr > 16)) {
            found = true;
            item = i;
        }
    }
    *arr = 20;
    return item;
}

```

On the **Source** pane, you can:

- **Examine Source Code**

  On the **Source** pane, if you right-click a text string, the context menu provides options to examine your code. For example, right-click the global variable `PowerLevel`:



Source
Dashboard
tasks2.c
Print Source Code: tasks2.c
Search For "PowerLevel" in Current Source File Ctrl+F
Search For "PowerLevel" in All Source Files
Search For All References
Go To Definition
Go To Line Ctrl+L
Show Error Call Graph
Show In Variable Access View
Add Pre-Justification To Clipboard
Show Dashboard
Expand All Macros
Collapse All Macros
Create Duplicate Code Window
Source
Data Range Configuration

```

void Increase_PowerLevel(void)
{
    PowerLevel++;
}


static vo
{
    int l
    if (l
}


void Comp
{
    stati

    Compu
```

Use the following options to examine and navigate through your code:

- **Search "PowerLevel" in Current Source File** — List occurrences of the string within the current source file in the **Search** pane.
- **Search "PowerLevel" in All Source Files** — List occurrences of the string within all source files in the **Search** pane.
- **Search For All References** — List all references in the **Search** pane. The software supports this feature for global and local variables, functions, types, and classes.

  - **Go To Definition** — Go to the line of code that contains the definition of `PowerLevel`. The software supports this feature for global and local variables, functions, types, and classes.
  - **Go To Line** — Open the Go To Line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.
  - **Expand All Macros** or **Collapse All Macros** — Display or hide the content of macros in current source file.

- **View Variable Range**

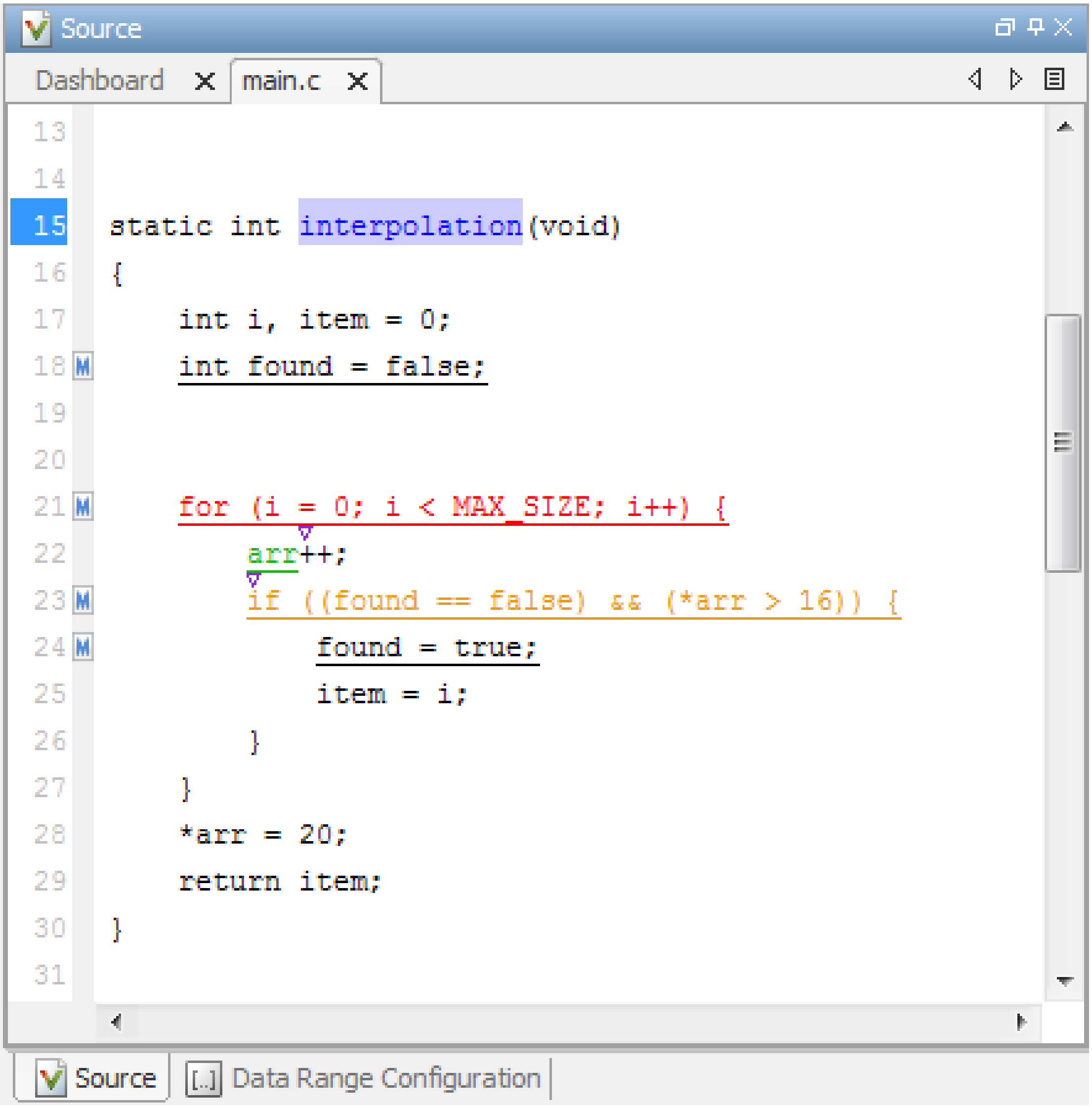
  Place your cursor over a check to view range information for variables, operands, function parameters, and return values.

  If a tooltip is available for a keyword or identifier on the **Source** pane, Polyspace:

  - Uses solid underlining for the keyword or identifier if it is associated with a check.
  - Uses dashed underlining for the keyword or identifier if it is not associated with a check.

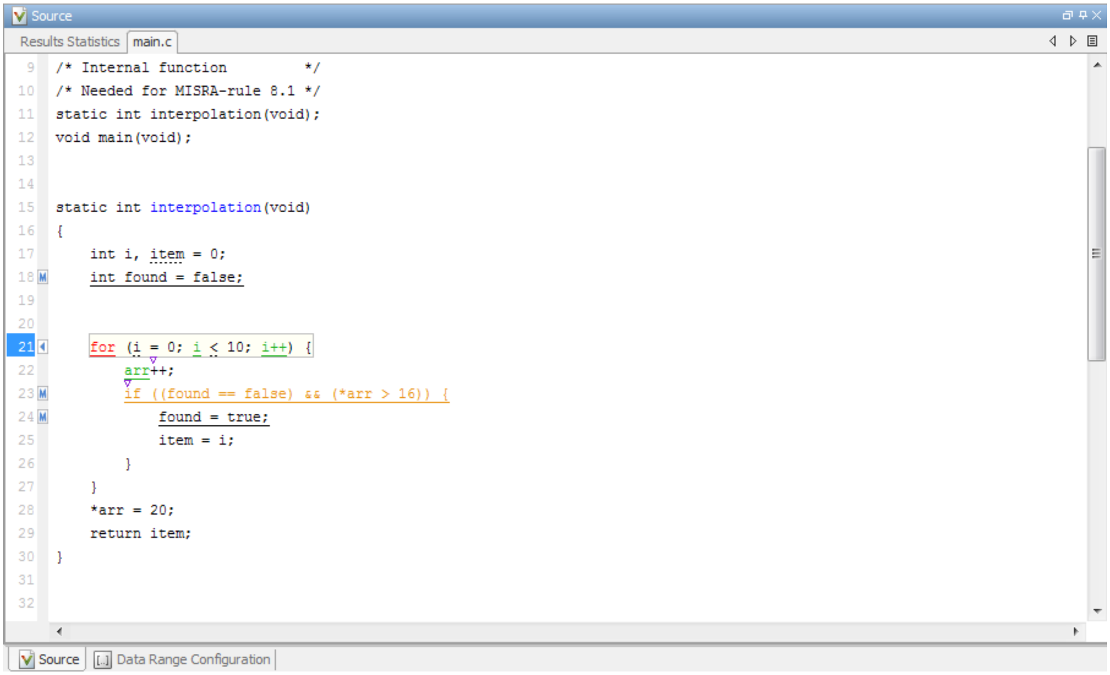
- **Expand Macros**

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



```


static int interpolation(void)
{
    int i, item = 0;
    int found = false;


    for (i = 0; i < MAX_SIZE; i++) {
        arr++;
        if ((found == false) && (*arr > 16)) {
            found = true;
            item = i;
        }
    }
    *arr = 20;
    return item;
}

```

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



```
/* Internal function        */
/* Needed for MISRA-rule 8.1 */
static int interpolation(void);
void main(void);


static int interpolation(void)
{
    int i, item = 0;
    int found = false;

    for (i = 0; i < 10; i++) {
        arr++;
        if ((found == false) && (*arr > 16)) {
            found = true;
            item = i;
        }
    }
    *arr = 20;
    return item;
}
```

To display the normal source code again, click the line away from the shaded region, for example, on the arrow icon.

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

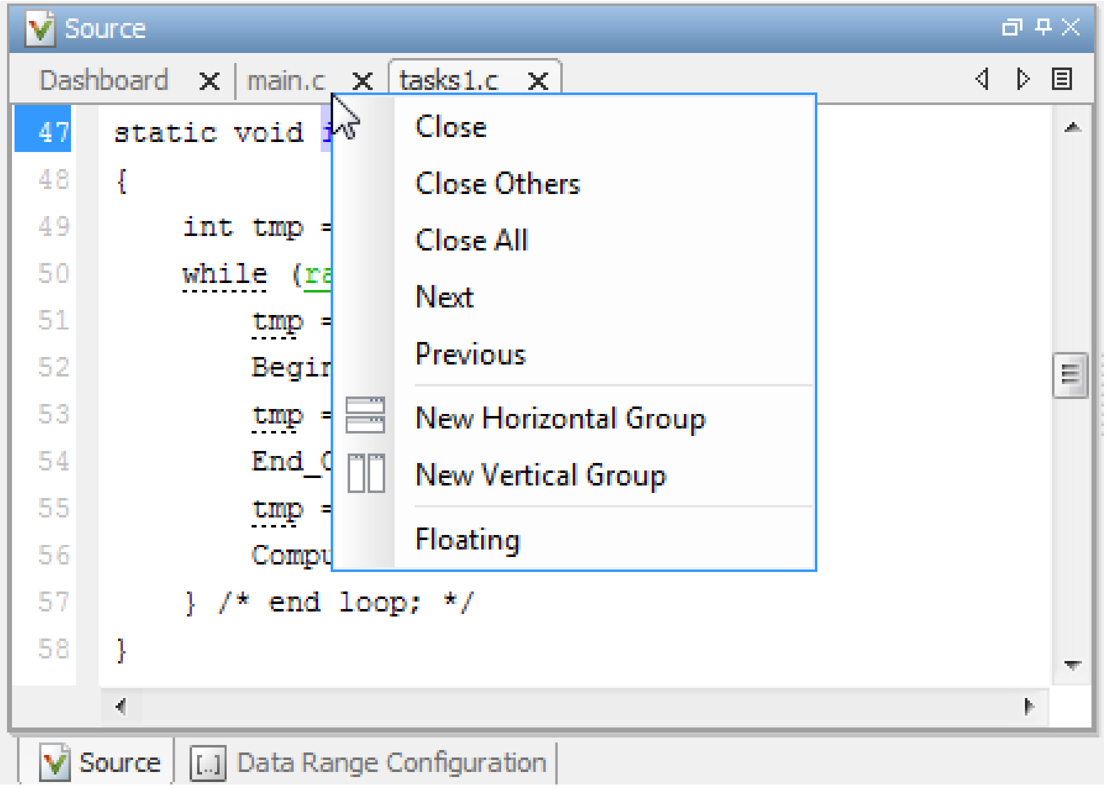
1. Right-click any point within the source code view.
2. From the context menu, select either **Expand All Macros** or **Collapse All Macros**.

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

- **Manage Multiple Files**

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

On the **Source** pane toolbar, right-click a view.



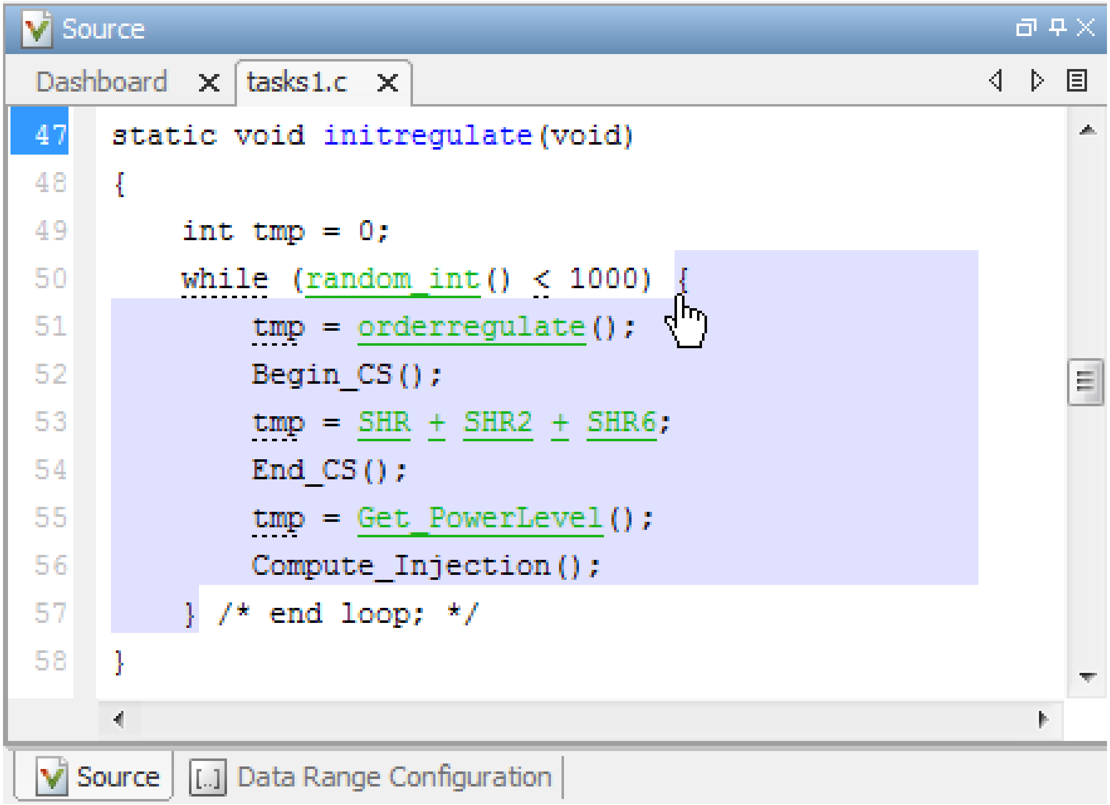
```
static void
{
    int tmp =
    while (ra
        tmp =
        Begir
        tmp =
        End_C
        tmp =
        Compu
    } /* end loop; */
}
```

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

- **Close** – Close the currently selected source file. You can also use the χ button to close the tabs.
- **Close Others** – Close all source files except the currently selected file.
- **Close All** – Close all source files.
- **Next** – Display the next view.
- **Previous** – Display the previous view.
- **New Horizontal Group** – Split the **Source** pane horizontally to display the selected source file below another file.

  - **New Vertical Group** – Split the **Source** pane vertically to display the selected source file side-by-side with another file.
  - **Floating** – Display the current source file in a new window, outside the **Source** pane.
- **View Code Block**

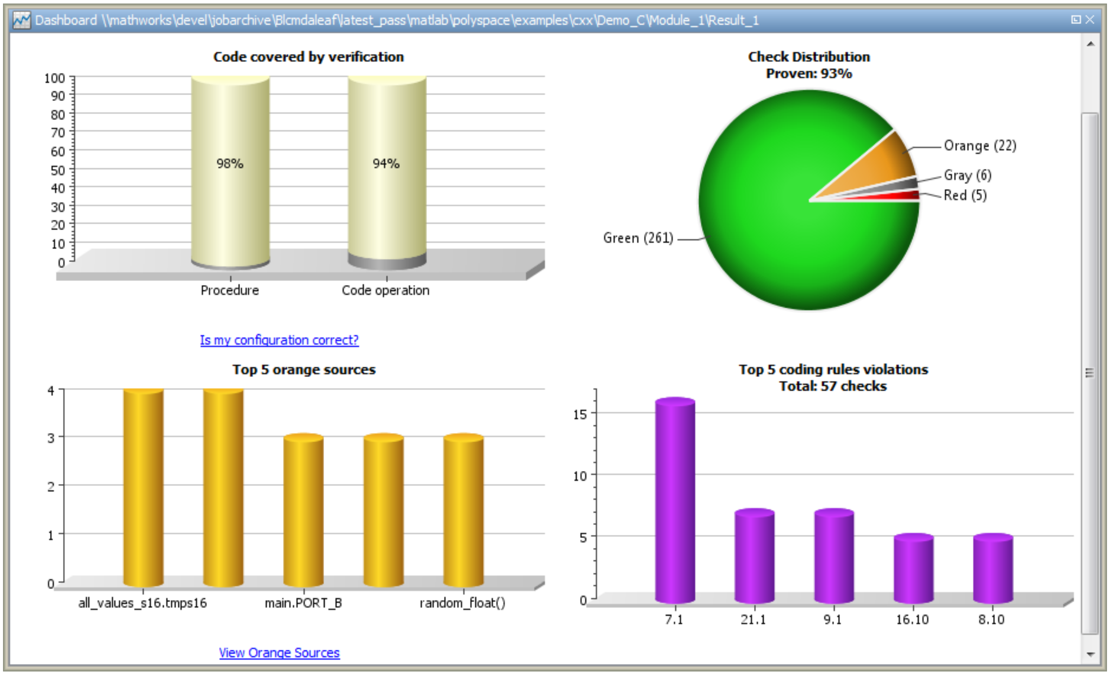
  On the **Source** pane, to highlight a block of code, click either its opening or closing brace.



```
static void initregulate(void)
{
    int tmp = 0;
    while (random_int() < 1000) {
        tmp = orderregulate();
        Begin_CS();
        tmp = SHR + SHR2 + SHR6;
        End_CS();
        tmp = Get_PowerLevel();
        Compute_Injection();
    } /* end loop; */
}
```

Dashboard

The **Dashboard** tab on the **Source** pane provides statistics on the verification results in a graphical format.



On this tab, you can view four graphs and charts:

- **Code covered by verification**

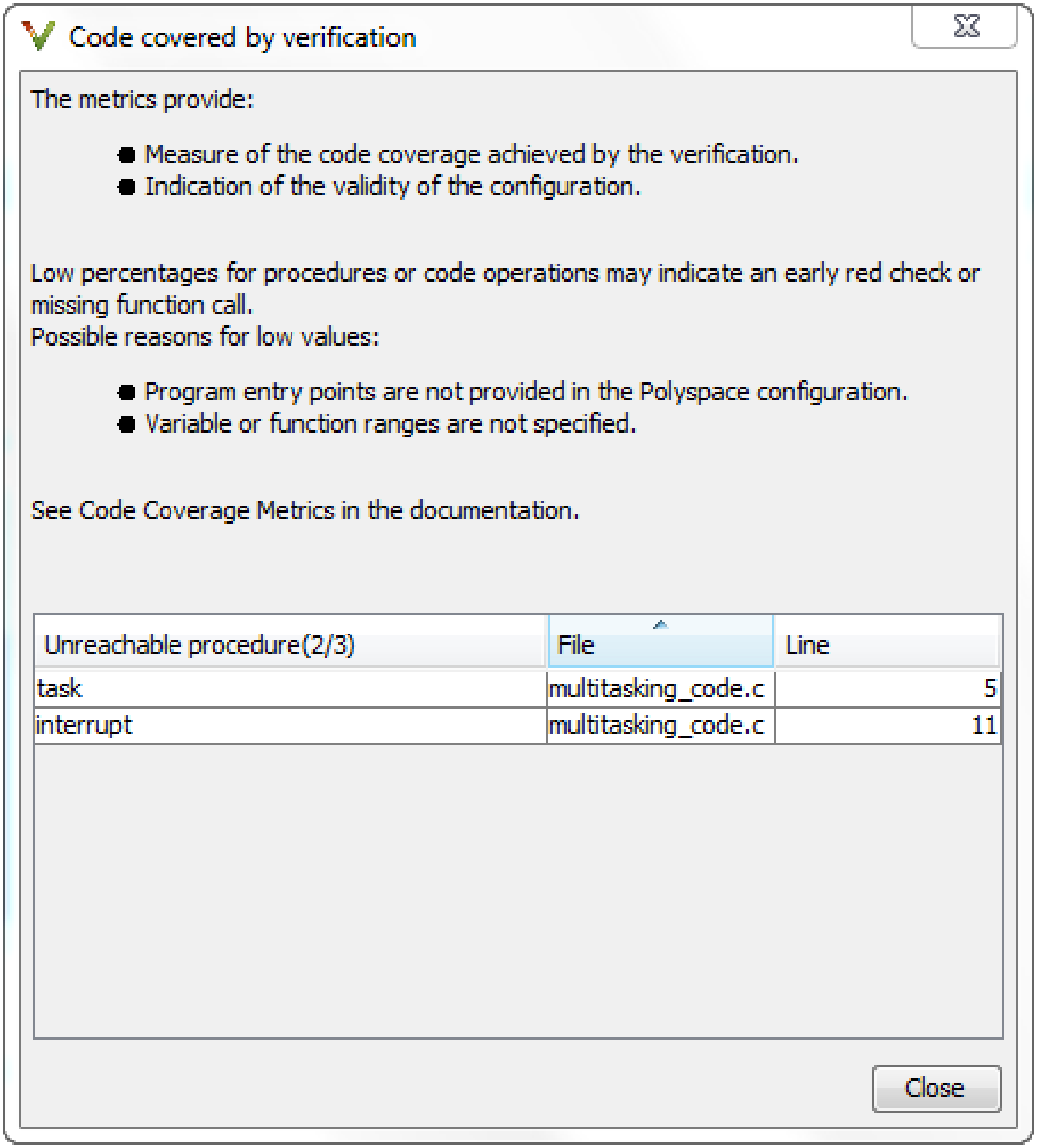
  This column graph displays:

  - The percentage of procedures covered by verification. You can see this percentage in the `Procedure` column.
  - The percentage of elementary operations in executable procedures covered by verification. You can see this percentage in the `Code operation` column.

  These percentages provide a measure of:

  - Code coverage achieved by the Polyspace verification.
  - Validity of your Polyspace configuration.

  Click the column graph to open the Code covered by verification window.



Code covered by verification

The metrics provide:

- Measure of the code coverage achieved by the verification.
- Indication of the validity of the configuration.

Low percentages for procedures or code operations may indicate an early red check or missing function call.
Possible reasons for low values:

- Program entry points are not provided in the Polyspace configuration.
- Variable or function ranges are not specified.

See Code Coverage Metrics in the documentation.

Unreachable procedure(2/3)	File	Line
task	multitasking_code.c	5
interrupt	multitasking_code.c	11

This window contains:

- The fraction of procedures that are unreachable in the format, *`Number of unreachable procedures/Total number of procedures`*.
- A list of unreachable procedures along with the file and line number where they are defined. Selecting a procedure displays the procedure definition in the **Source** pane.

A low coverage can indicate an early red check or missing function call. Consider the following code:

```
1  void coverage_eg(void)
2  {
3    int x;
4
5    x = 1 / x;
6    x = x + 1;
7    propagate();
8  }
```

Verification generates only one red **Non-initialized local variable** check, for a read operation on the variable `x` — see line 5. The software does not display checks for these elementary operations:

- On line 5, for the division operation, a **Division by zero** check.
- On line 5, for the division operation, an **Overflow** check.
- On line 6, for the addition operation, an **Overflow** check.
- On line 6, for another read operation on `x`, a **Non-initialized local variable** check.

As the software displays only one out of the five operation checks for the code, the percentage of elementary operations covered is 1/5 or 20%. The software does not take into account the checks inside the unreachable function `propagate()`.

- **Check distribution**

  This pie chart displays the number of checks of each color. For a description of the check colors, see "Result and Source Code Colors".

  Using this pie chart, you can obtain an estimate of:

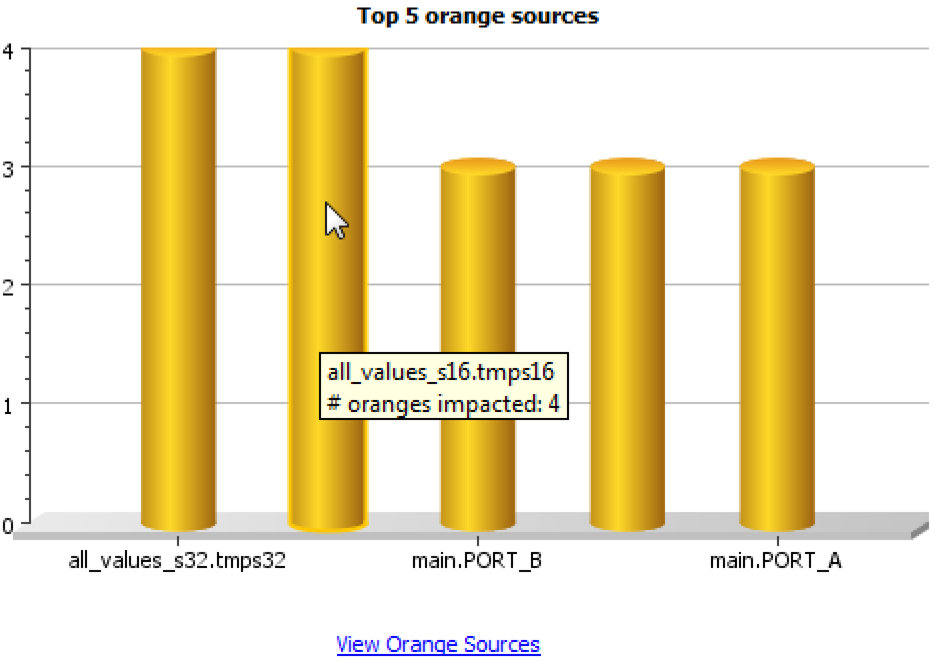
  - The number of checks to review.
  - The selectivity of your verification — the fraction of checks that are not orange.

You can follow certain coding rules or specify certain verification options to reduce the number of orange checks. See “Reduce Orange Checks”.

- **Top 5 orange sources**

  An orange source is a variable or function that leads to an orange check. This column graph displays five orange sources affecting the most number of checks.

  Each column represents an orange source. The columns are arranged in the order of number of checks affected. The height of the column indicates the number of checks affected by the corresponding orange source. Place your cursor on a column to open a tooltip showing the source name and the number of checks affected by the source.



  Using this chart, you can:

  - View the five sources affecting the most number of checks. Select a column to view further details of the corresponding orange source in the **Orange Sources** pane.

  - Prioritize your review of orange checks. If there are sources affecting a large number of orange checks, address those sources if possible before you begin a systematic review of orange checks. See “Create Constraint Template After Verification”.

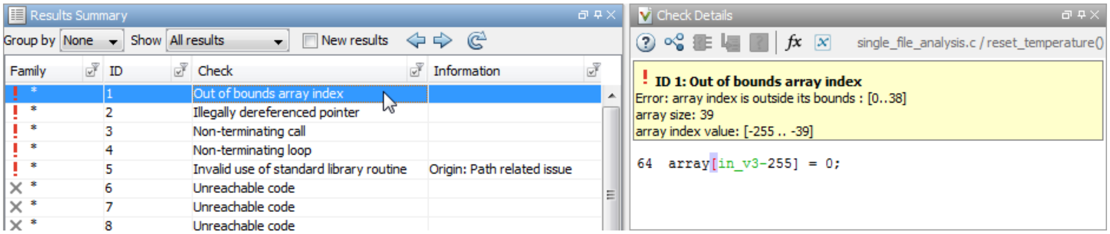
- **Top 5 coding rule violations**

  This column graph displays the five most violated coding rules. Each column represents a coding rule and is indexed by the rule number. The height of the column indicates the number of violations of the coding rule represented by that column.

  For a list of supported coding rules, see “Supported MISRA C:2004 Rules”, “MISRA C:2012 Directives and Rules”, “Supported MISRA C++ Coding Rules”, and “Supported JSF C++ Coding Rules”.

Check Details

On the **Results Summary** pane, if you select a check, you see additional information on the **Check Details** pane.



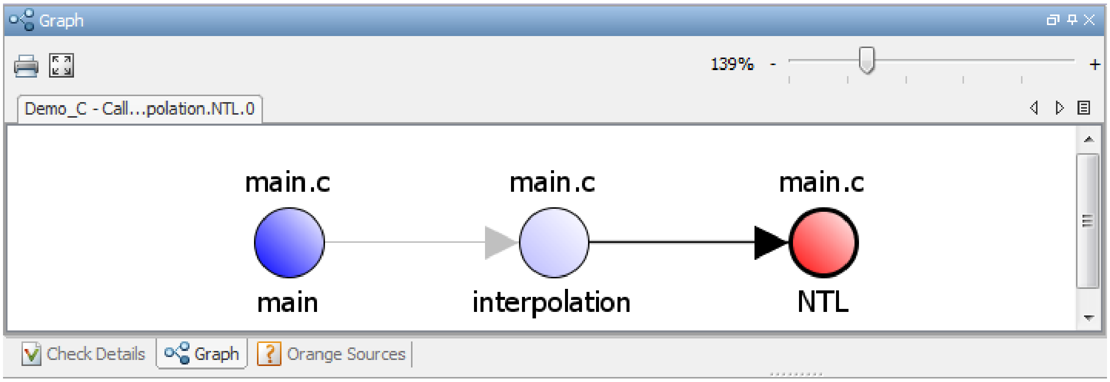
Family		ID	Check	Information
!	*	1	Out of bounds array index	
!	*	2	Illegally dereferenced pointer	
!	*	3	Non-terminating call	
!	*	4	Non-terminating loop	
!	*	5	Invalid use of standard library routine	Origin: Path related issue
✕	*	6	Unreachable code	
✕	*	7	Unreachable code	
✕	*	8	Unreachable code	

! **ID 1: Out of bounds array index**
Error: array index is outside its bounds : [0..38]
array size: 39
array index value: [-255 .. -39]

```
64  array[in_v3-255] = 0;
```

Show Error Call Graph

Click the **Show error call graph** icon, in the **Check Details** pane toolbar to display the call sequence that leads to the code associated with a check.



Show Call Hierarchy and Variable Access

From the **Check Details** pane, you can open the **Call Hierarchy** and **Variable Access** panes.

- Select the *fx* button to open the **Call Hierarchy** pane.

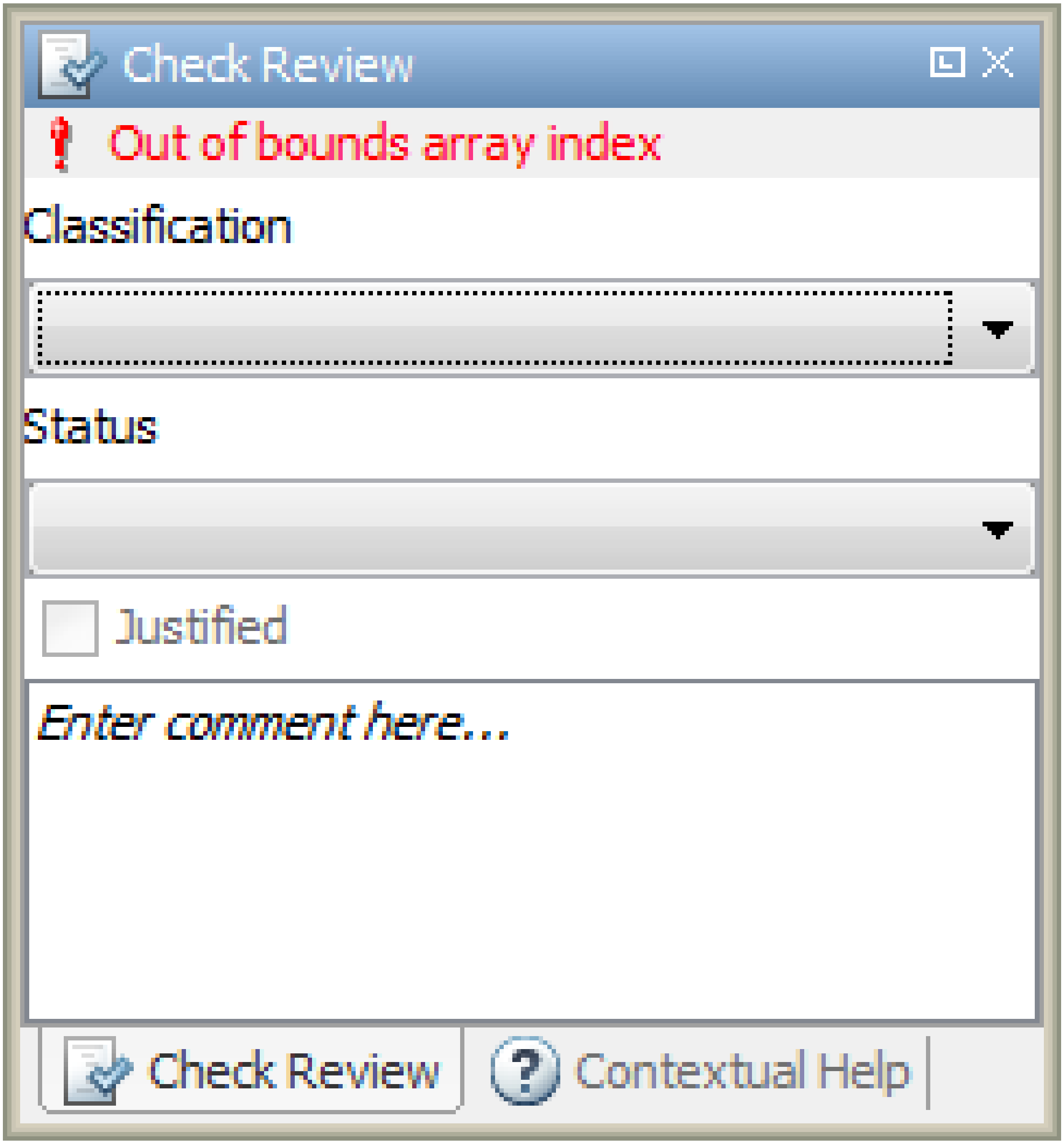
  On this pane, you can see the function in which the current check occurs, along with its callers and callees. For more information, see “Call Hierarchy”.

- Select the button to open the **Variable Access** pane.

  On this pane, you can see the global variables in your code. For more information, see “Variable Access”.

Check Review

When reviewing checks, use the **Check Review** tab to assign a **Classification** and **Status** to each check. You can also enter comments to describe the results of your review. This action helps you track the progress of your review and avoid reviewing the same check twice.



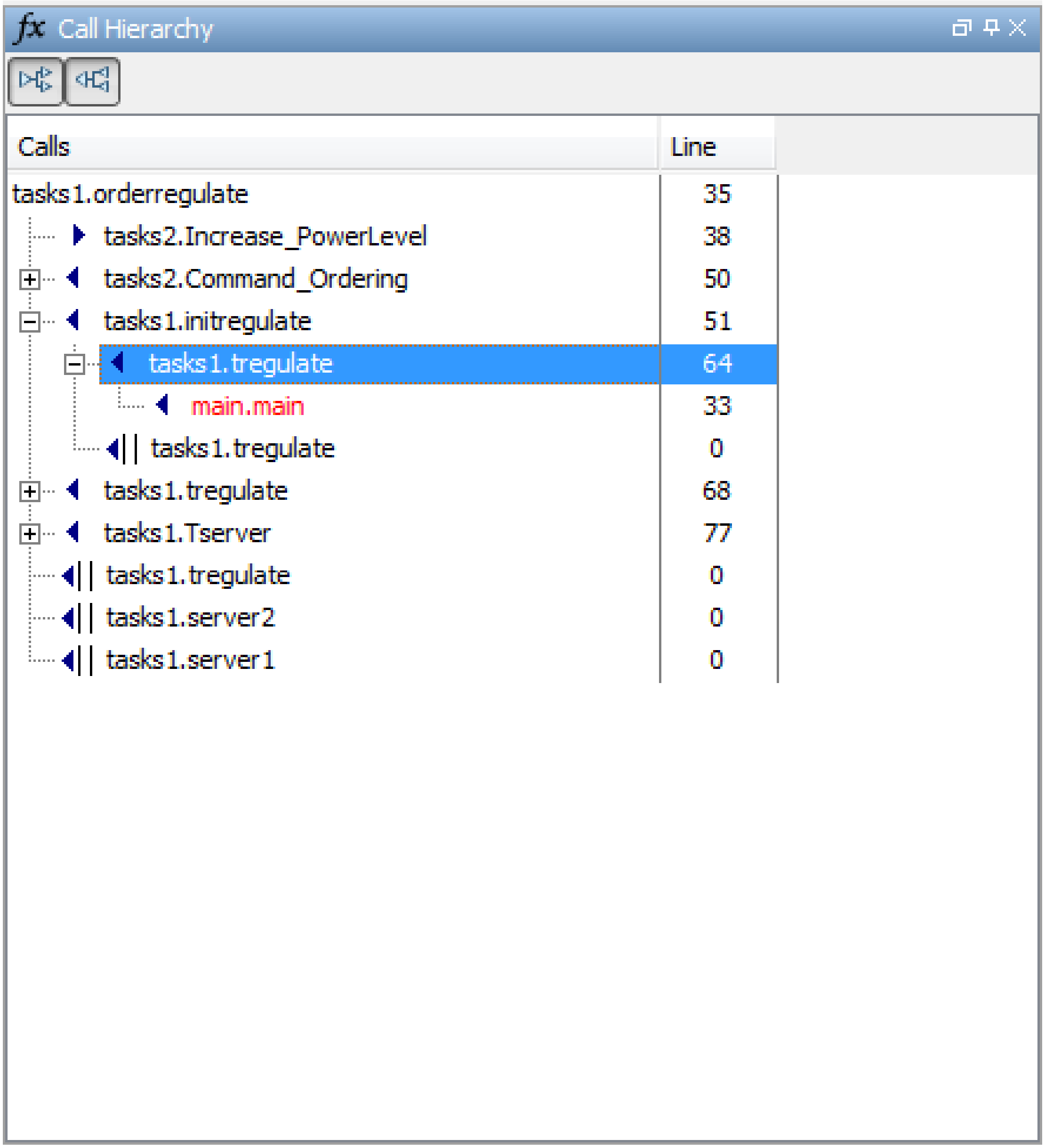
For more information, see “Add Review Comments to Results”.

Call Hierarchy

The **Call Hierarchy** pane displays the call tree of functions in the source code.

For each function,`foo`, the **Call Hierarchy** pane lists the functions and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by ◀ (functions), or ◀|| (tasks). The callees are indicated by ▶ (functions) or ||▶ (tasks). The **Call Hierarchy** pane lists both direct function calls and indirect calls through function pointers.

In the following example, the **Call Hierarchy** pane displays the function, `orderregulate`, in the file, `tasks1.c`. It also displays the callers and the callees of `orderregulate`.



Calls	Line
tasks1.orderregulate	35
tasks2.Increase_PowerLevel	38
tasks2.Command_Ordering	50
tasks1.initregulate	51
tasks1.tregulate	64
main.main	33
tasks1.tregulate	0
tasks1.tregulate	68
tasks1.Tserver	77
tasks1.tregulate	0
tasks1.server2	0
tasks1.server1	0

Depending on the name, the corresponding line number in the **Call Hierarchy** pane refers to a different line in the source code:

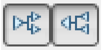
- For the function name, the line number refers to the beginning of the function definition. In the preceding example, the definition of `tasks1.orderregulate` begins on line 35.
- For a callee name, the number refers to the line where the callee is called. In the preceding example, callee, `tasks2.Increase_PowerLevel`, is called by `tasks1.orderregulate` on line 38.
- For a caller name, the number refers to the line where the caller calls the function. In the preceding example, caller, `tasks2.Command_Ordering`, calls `tasks1.orderregulate` on line 50.

Tip Select a caller or callee name to navigate to the call location in the source code.

You can perform the following actions from the **Call Hierarchy** pane:

- **Show/Hide Callers and Callees**

  Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button

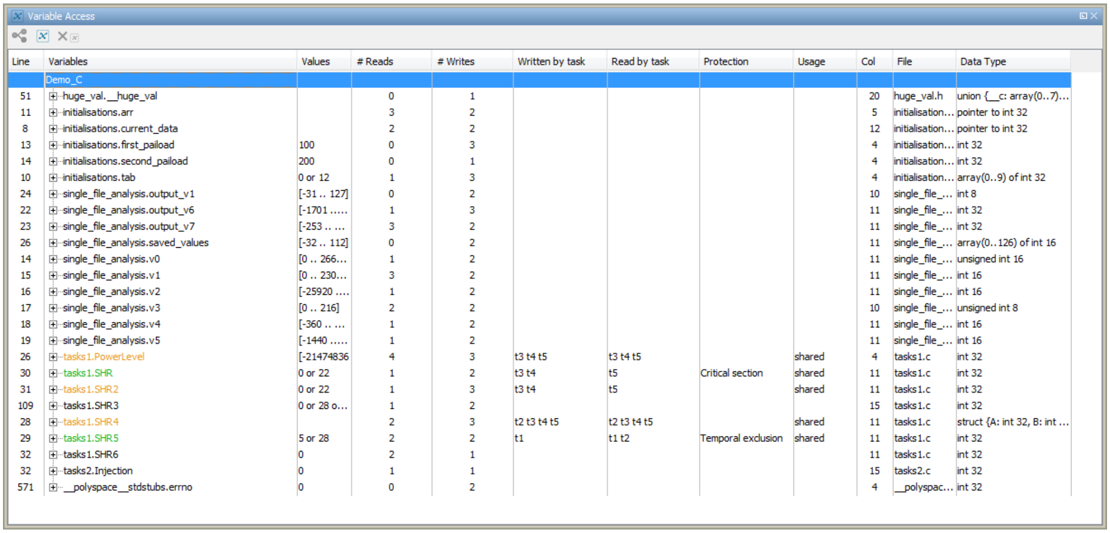
  

- **Navigate Call Hierarchy**

  You can navigate the call hierarchy in your source code using this pane. For a function, double-click a caller or callee name to navigate to the caller or callee definition in the source code.

Variable Access

The **Variable Access** pane displays global variables. For each global variable, the pane lists all functions and tasks performing read/write access on the variables, along with their attributes, such as values, read/write accesses and shared usage.



Line	Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Col	File	Data Type
	Demo_C										
51	huge_val.__huge_val		0	1					20	huge_val.h	union {__c: array(0..7)...
11	initialisations.arr		3	2					5	initialisation...	pointer to int 32
8	initialisations.current_data		2	2					12	initialisation...	pointer to int 32
13	initialisations.first_paiload	100	0	3					4	initialisation...	int 32
14	initialisations.second_paiload	200	0	1					4	initialisation...	int 32
10	initialisations.tab	0 or 12	1	3					4	initialisation...	array(0..9) of int 32
24	single_file_analysis.output_v1	[-31 .. 127]	0	2					10	single_file_...	int 8
22	single_file_analysis.output_v6	[-1701	1	3					11	single_file_...	int 32
23	single_file_analysis.output_v7	[-253	3	2					11	single_file_...	int 32
26	single_file_analysis.saved_values	[-32 .. 112]	0	2					11	single_file_...	array(0..126) of int 16
14	single_file_analysis.v0	[0 .. 266...	1	2					11	single_file_...	unsigned int 16
15	single_file_analysis.v1	[0 .. 230...	3	2					11	single_file_...	int 16
16	single_file_analysis.v2	[-25920	1	2					11	single_file_...	int 16
17	single_file_analysis.v3	[0 .. 216]	2	2					10	single_file_...	unsigned int 8
18	single_file_analysis.v4	[-360	1	2					11	single_file_...	int 16
19	single_file_analysis.v5	[-1440	1	2					11	single_file_...	int 16
26	tasks1.PowerLevel	[-21474836	4	3	t3 t4 t5	t3 t4 t5		shared	4	tasks1.c	int 32
30	tasks1.SHR	0 or 22	1	2	t3 t4	t5	Critical section	shared	11	tasks1.c	int 32
31	tasks1.SHR2	0 or 22	1	3	t3 t4	t5		shared	11	tasks1.c	int 32
109	tasks1.SHR3	0 or 28 o...	1	2					15	tasks1.c	int 32
28	tasks1.SHR4		2	3	t2 t3 t4 t5	t2 t3 t4 t5		shared	11	tasks1.c	struct {A: int 32, B: int ...
29	tasks1.SHR5	5 or 28	2	2	t1	t1 t2	Temporal exclusion	shared	11	tasks1.c	int 32
32	tasks1.SHR6	0	2	1					11	tasks1.c	int 32
32	tasks2.Injection	0	1	1					15	tasks2.c	int 32
571	__polyspace__stdstubs.errno	0	0	2					4	__polyspac...	int 32

For each variable and each read/write access, the **Variable Access** pane contains the relevant attributes. For the variables, the various attributes are listed in this table.

Attribute	Description
Variables	Name of Variable, ***File_Name.*** ***Variable_Name*** ***File_Name***: Name of file where variable is declared
Values	Value (or range of values) of variable
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Written by task	Name of tasks writing on variable using aliases, **t1**, **t2**, **t3** **Tip** To see the full names for aliases, right-click anywhere on the **Variable Access** pane and select **Show Legend**.
Read by task	Name of tasks reading variable using aliases, **t1**, **t2**, **t3**

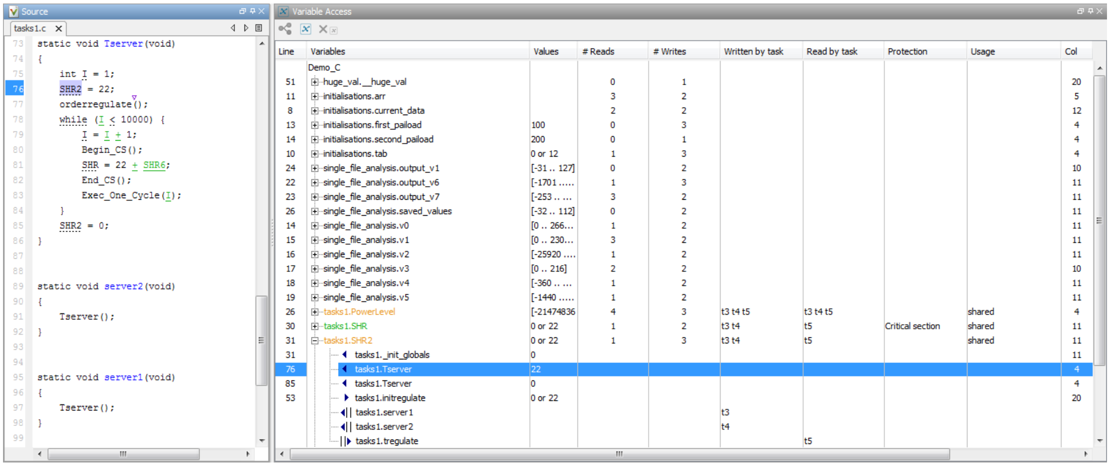
Attribute	Description
	Tip To see the full names for aliases, right-click anywhere on the **Variable Access** pane and select **Show Legend**.
Protection	Whether shared variable is protected from concurrent access (Filled only when **Usage** column has entry, **Shared**) The possible entries in this column are: • **Critical Section**: If variable is accessed in critical section of code • **Temporal Exclusion**: If variable is accessed in mutually exclusive tasks For more details on these entries, see “Multitasking”.
Usage	`Shared`, if variable is shared between tasks; otherwise, blank
Line	Line number of variable declaration
Col	Column number (number of characters from beginning of line) of variable declaration
File	Source file containing variable declaration
Data Type	Data type of variable (C/C++ data types or structures/classes)

Double-click a variable name to view read/write access operations on the variable. The arrowhead symbols ▸ and ◂ in the **Variable Access** pane indicate functions performing read and write access respectively on the global variable. Likewise, tasks performing read and write access are indicated by the symbols ||▸ and ◂|| respectively. For further information on tasks, see “Entry points (C/C++)”.

For access operations on the variables, the various attributes described in the pane are listed in this table.

Attribute	Description
Variables	Names of function (or task) performing read/write access on the variable, ***File_Name.Function_Name*** ***File_Name***: Name of file containing function (or task) definition
Values	Value or range of values of variable in the function or task performing read/write access
Written by task	*Only for tasks*: Name of task performing write access on variable
Read by task	*Only for tasks*: Name of task performing read access on variable
Line	Line number where function or task accesses variable
Col	Column number where function or task accesses variable
File	Source file containing access operation on variable

For example, consider the global variable, SHR2:



```
static void Tserver(void)
{
    int I = 1;
    SHR2 = 22;
    orderregulate();
    while (I < 10000) {
        I = I + 1;
        Begin_CS();
        SHR = 22 + SHR6;
        End_CS();
        Exec_One_Cycle(I);
    }
    SHR2 = 0;
}


static void server2(void)
{
    Tserver();
}


static void server1(void)
{
    Tserver();
}

```

Line	Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Col
	Demo_C								
51	huge_val.__huge_val		0	1					20
11	initialisations.arr		3	2					5
8	initialisations.current_data		2	2					12
13	initialisations.first_paiload	100	0	3					4
14	initialisations.second_paiload	200	0	1					4
10	initialisations.tab	0 or 12	1	3					4
24	single_file_analysis.output_v1	[-31 .. 127]	0	2					10
22	single_file_analysis.output_v6	[-1701	1	3					11
23	single_file_analysis.output_v7	[-253	3	2					11
26	single_file_analysis.saved_values	[-32 .. 112]	0	2					11
14	single_file_analysis.v0	[0 .. 266...	1	2					11
15	single_file_analysis.v1	[0 .. 230...	3	2					11
16	single_file_analysis.v2	[-25920	1	2					11
17	single_file_analysis.v3	[0 .. 216]	2	2					10
18	single_file_analysis.v4	[-360	1	2					11
19	single_file_analysis.v5	[-1440	1	2					11
26	tasks1.PowerLevel	[-21474836	4	3	t3 t4 t5	t3 t4 t5		shared	4
30	tasks1.SHR	0 or 22	1	2	t3 t4	t5	Critical section	shared	11
31	tasks1.SHR2	0 or 22	1	3	t3 t4	t5		shared	11
31	tasks1._init_globals	0							11
76	tasks1.Tserver	22							4
85	tasks1.Tserver	0							4
53	tasks1.initregulate	0 or 22							20
	tasks1.server1				t3				
	tasks1.server2				t4				
	tasks1.tregulate					t5			

The function, `Tserver`, in the file, `tasks1.c`, performs two write operations on `SHR2`. This is indicated in the **Variable Access** pane by the two instances of `tasks1.Tserver()` under the variable, `SHR2`, marked by ◂. Likewise, the two write accesses by tasks, `server1` and `server2`, are also listed under `SHR2` and marked by ◂||.

The color scheme for variables in the **Variable Access** pane is:

- Black: global variable.
- Orange: global variable, shared between tasks with no protection against concurrent access.
- Green: global variable, shared between tasks and protected against concurrent access.
- Gray: global variable, declared but not used in reachable code.

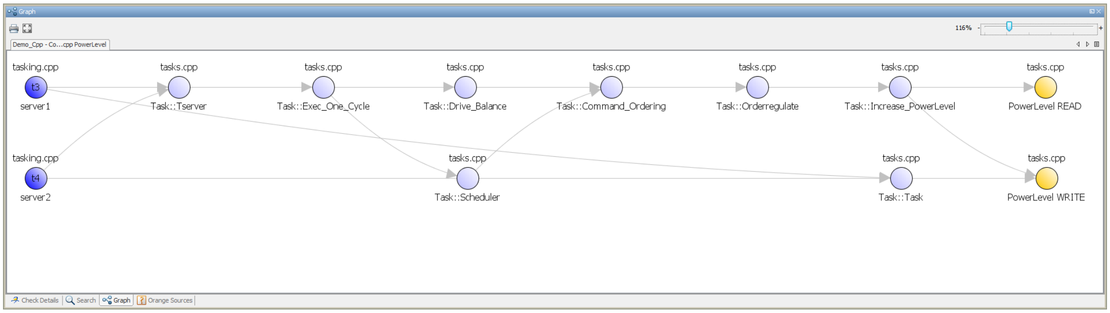
The information about global variables and read/write access operations obtained from the **Variable Access** pane is called the data dictionary.

You can also perform the following actions from the **Variable Access** pane.

- **View Access Graph**

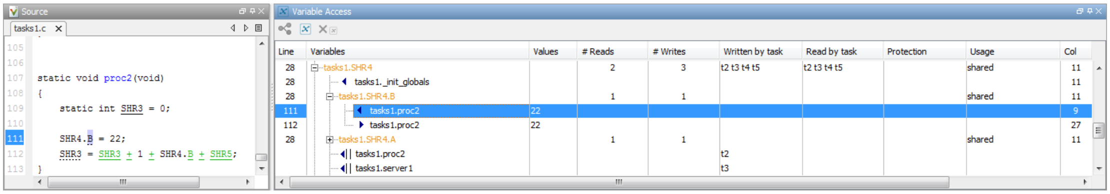
  View the access operations on a global variable in graphical format using the **Variable Access** pane. Select the global variable and click ⚯.

  Here is an example of an access graph:



- **View Structured Variables**

  For structured variables, view the individual fields from the **Variable Access** pane. For example, for the structure, `SHR4`, the pane displays the fields, `SHR4.A` and `SHR4.B`, and the functions performing read/write access on them.



```
static void proc2(void)
{
    static int SHR3 = 0;

    SHR4.B = 22;
    SHR3 = SHR3 + 1 + SHR4.B + SHR5;
}
```

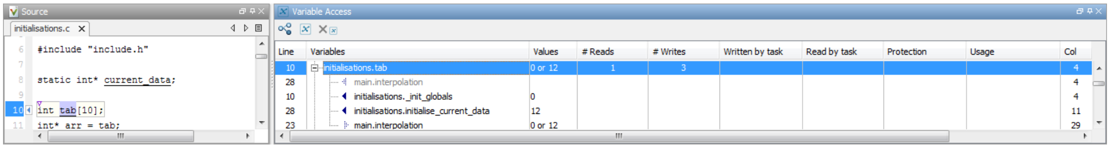
Line	Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Col
28	tasks1.SHR4		2	3	t2 t3 t4 t5	t2 t3 t4 t5		shared	11
28	tasks1._init_globals								11
28	tasks1.SHR4.B		1	1				shared	11
111	tasks1.proc2	22							9
112	tasks1.proc2	22							27
28	tasks1.SHR4.A		1	1				shared	11
	tasks1.proc2				t2				
	tasks1.server1				t3				

View Access Through Pointers

View access operations on global variables performed indirectly through pointers.

If a read/write access on a variable is performed through pointers, then the access is marked by ⁞ (read) or ⁝ (write).

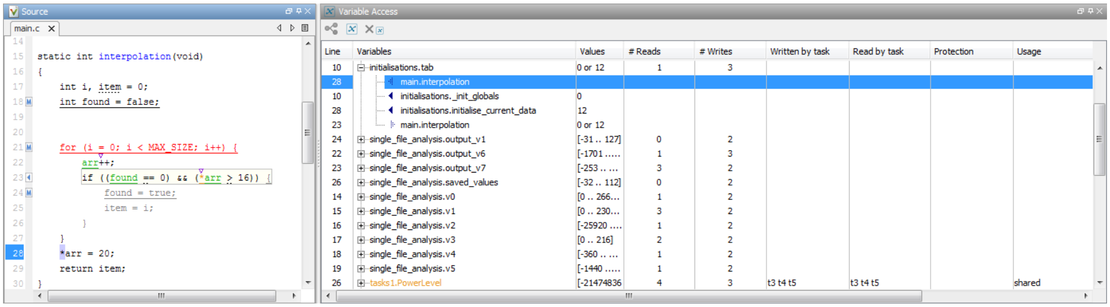
For instance, in the file, `initialisations.c`, the variable, `arr`, is declared as a pointer to the array, `tab`.



```
#include "include.h"

static int* current_data;

int tab[10];
int* arr = tab;
```

Line	Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Col
10	initialisations.tab	0 or 12	1	3					4
28	main.interpolation								4
10	initialisations._init_globals	0							4
28	initialisations.initialise_current_data	12							11
23	main.interpolation	0 or 12							29

In the file `main.c`, `tab` is both read and written in the function, `interpolation()`, through the pointer variable, `arr`. This operation is shown in the **Variable Access** pane by the ⁞ and ⁝ icons respectively.



```

static int interpolation(void)
{
    int i, item = 0;
    int found = false;


    for (i = 0; i < MAX_SIZE; i++) {
        arr++;
        if ((found == 0) && (*arr > 16)) {
            found = true;
            item = i;
        }
    }
    *arr = 20;
    return item;
}
```

Line	Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage
10	initialisations.tab	0 or 12	1	3				
28	main.interpolation							
10	initialisations._init_globals	0						
28	initialisations.initialise_current_data	12						
23	main.interpolation	0 or 12						
24	single_file_analysis.output_v1	[-31 .. 127]	0	2				
22	single_file_analysis.output_v6	[-1701	1	3				
23	single_file_analysis.output_v7	[-253	3	2				
26	single_file_analysis.saved_values	[-32 .. 112]	0	2				
14	single_file_analysis.v0	[0 .. 266...	1	2				
15	single_file_analysis.v1	[0 .. 230...	3	2				
16	single_file_analysis.v2	[-25920	1	2				
17	single_file_analysis.v3	[0 .. 216]	2	2				
18	single_file_analysis.v4	[-360	1	2				
19	single_file_analysis.v5	[-1440	1	2				
26	tasks1.PowerLevel	[-21474836	4	3	t3 t4 t5	t3 t4 t5		shared

- **Show/Hide Callers and Callees**

  Customize the **Variable Access** pane to show only the shared variables. On the **Variable Access** pane toolbar, click the Non-Shared Variables button to show or hide non-shared variables.

- **Hide Access in Unreachable Code**

  Hide read/write access occurring in unreachable code by clicking the filter button .

- **Limitations**

  You cannot see an addressing operation on a global variable or object (in C++) as a read/write operation in the **Variable Access** pane. For example, consider the following C++ code:

```
class CO
{
public:
  CO() {}
  int get_flag()
  {
    volatile int rd;
    return rd;
  }
  ~CO() {}
private:
  int a;                /* Never read/written */
};

CO c0;                  /* c0 is unreachable */

int main()
{
  if (c0.get_flag())    /* Uses address of the method */
    {
      int *ptr = take_addr_of_x();
      return 1;
    }
  else
    return 0;
}
```

  You do not see the method call `c0.get_flag()` in the **Variable Access** pane because the call is an addressing operation on the method belonging to the object `c0`.

Filter and Group Results

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

- Review certain categories of checks in preference to others. For instance, you first want to address checks resulting from **Out of bounds array index**.
- Review only new results found since the last verification.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Not review results you have already justified.

  Typically, in your second or later rounds of review, you would have some results already justified.
- Review only those results that you have already assigned a certain status. For instance, you want to review only those results to which you have assigned the status, `Investigate`.
- Review all results in the body of a particular file or function. Because of continuity of code, reviewing these results together can help you organize your review process.

  You can also review results in a file if you have written the code for that file only and not the entire set of source files used for verification.
- Not review the results in automatically generated functions.
- C++ only: Review all results dealing with a class definition.

In this section...

Filter Results

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

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

- “Review Code Metrics”
- “Limit Display of Orange Checks”

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

Item to Filter	Column
Results in a certain file or function	**File** or **Function**
Results associated with a certain class	**Class**
Results with a certain classification or status	**Classification** or **Status**
Results that you have justified. If you assign the status `No action planned` or `Justify with annotations`, a result is justified.	**Justified**
Checks only	**Family**
Checks of a certain color	**Family**
Global variables of a certain type	**Family**
Code metrics	**Family**

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

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

Note: You can also apply multiple filters.

Group Results

On the **Results Summary** pane:

- To show results without grouping, select **Group by** > **None**.
- To show results grouped by result type, select **Group by** > **Family**.

  The results are organized by type: checks, global variables, coding rule violations, code metrics. Within each type, they are grouped further.

  - The checks are grouped by color. Within each color, the checks are grouped by category. For more information on the categories, see “Run-Time Checks”.
  - The global variables are grouped by their usage. For more information, see “Global Variables”.
  - The coding rule violations are grouped by type of coding rule. For more information, see “Coding Rules”.
  - The code metrics are grouped by scope of metric. For more information, see “Code Metrics”.
- To show results grouped by file, select **Group by** > **File**.

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

  Within each class, the results are grouped by method.

Prioritize Check Review

This example shows how to prioritize your check review. Try the following approach. You can also develop your own procedure for organizing your orange check review.

Tip For easier review, run Polyspace Bug Finder on your source code first. Once you address the defects that Polyspace Bug Finder finds, run Polyspace Code Prover on your code.

1. Before beginning your check review, do the following:

   - See the **Code covered by verification** graph on the **Dashboard** pane. See if the **Procedure** and **Code operation** columns display a value closer to 100%. Otherwise, identify why Polyspace could not cover the code.

     For more information, see “Review Gray Checks”. If a substantial number of functions or code operations were not covered, after identifying and fixing the cause, run verification again.
   - See if you have used the right configuration. With the results open, select **Window** > **Show/Hide View** > **Configuration**.

     Sometimes, especially if you are switching between multiple configurations, you can accidentally use the wrong configuration for the verification.
2. On the **Results Summary** pane, select **Show** > **Critical Checks**.

   This action retains only red, gray and critical orange checks.
3. Click the forward arrow ⇨ to go to the first unreviewed check. Review this check.

   For more information, see “Results Review Process”.

   Continue to click the forward arrow until you have reviewed through all of the checks.
4. Before reviewing orange checks, review red and gray checks.
5. Prioritize your orange check review by:

   - Files and functions: For easier review, begin your orange check review from files and functions with fewer orange checks.

To view the percentage of non-orange checks per file and function, on the **Results Summary** pane, select **Group by** > **File**. Right-click a column header and select %.

- Check type: Review orange checks in the following order. Checks are more difficult to review as you go down this order.

Review Order	Checks
First	• Out of bounds array index • Non-initialized local variable • Division by zero • Shift operations
Second	• Overflow • Illegally dereferenced pointer
Third	Remaining checks

- Orange check sources: Review all orange checks caused by a single variable or function. Orange checks often arise from variables whose values cannot be determined from the code or functions that are not defined.

  To review the top sources, view the **Top 5 orange sources** graph on the **Dashboard** tab or the **Orange Sources** tab.

6. To ensure that you have addressed all red and critical orange checks, run verification again and view your results.
7. If you do not have red or unjustified critical orange checks, on the **Results Summary** pane, select **Show** > **All checks**.

   Depending on the quality level you want, you can choose whether to review the noncritical orange checks or not. For more information, see “Managing Orange Checks”.
8. To see what percentage of checks you have justified, broken down by color and type:

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

   b. For each color and type, view the entries in the **Justified** column.
9. To see what percentage of checks you have justified, broken down by file and function:

a. On the **Results Summary** pane, select **Group by** > **File**.
b. For each file and function, view the entries in the **Justified** column.

Related Examples

- “Review Red Checks”
- “Review Gray Checks”
- “Review Orange Checks”
- “Limit Display of Orange Checks”

Results Folder Contents

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

By default, your results are saved in your project folder in a folder called `Result_#`. To use a different folder, see “Specify Results Folder”.

Files in the Results Folder

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

- `Polyspace_`*`release_project_name_date-time`*`.log` — A log file associated with each analysis.
- `ps_results.pscp` — An encrypted file containing your Polyspace results. Open this file in the Polyspace environment to view your results.
- `ps_sources.db` — A non-encrypted database file listing source files and macros.
- `drs-template.xml` — A template generated when you use constraint specification.
- `ps_comments.db` — An encrypted database file containing your comments and justifications.
- `comments_bak` — A subfolder used to import comments between results.
- `.status` and `.settings` — Two folders used to store files needed to relaunch the analysis.
- `Polyspace-Doc` — When you generate a report, by default, your report is saved in this folder with the name *`ProjectName_ReportType`*. For example, a developer report in Rich Text Format would be, `myProject_Developer.rtf`.
- `Polyspace-Instrumented` — When the software runs the Automatic Orange Tester (AOT) at the end of a static verification, the software creates the `Polyspace-Instrumented` folder. The `Polyspace-Instrumented` folder contains files associated with the configuration and running of the Automatic Orange Tester.

See Also

-results-dir

Related Examples

- “Specify Results Folder”

- “Open Results”

Generate Report

This example shows how to generate a report from your verification results. Using a customizable template, the report presents your results in a concise manner for managerial review or other purposes. To generate a verification report, do one of the following:

- Specify certain options before verification so that the software automatically generates a report.
- Generate a report from your verification results.

In this section...

Specify Report Generation Before Verification

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Reporting**. Specify report generation options. For more information, see "Reporting". **2** Run verification and open your results. **3** Select **Reporting** > **Open Report** **4** Navigate to the `Polyspace-Doc` subfolder in your results folder. You can see the generated report in this subfolder. Click **OK** to open the report.	Use the appropriate option with the `polyspace-code-prover-nodesktop` command. For more information on the options, see the section **Command-Line Information** in "Reporting". Additionally, you can also specify a report name using the option -report-output-name.

Generate Report After Verification

User Interface	Command Line
1 Open your verification results. 2 Select **Reporting** > **Run Report**. The Run Report dialog box opens. 3 In the **Select Reports** section, select the report templates you want to use. For example, you can select **Developer** and **Quality**. For more information, see "Report template (C/C++)". 4 Select an **Output folder** in which to save the reports. 5 Select the **Output format** for the reports. 6 Click **Run Report**. The software creates the specified reports and opens them.	Use the appropriate option with the `polyspace-report-generator` command. The available options are: • `-template` *`path`*: Path to report template file. For more information, see "Report template (C/C++)". The predefined report templates are in *`matlabroot`*`\polyspace\toolbox\psrptgen\templates\Developer.rpt`. Here, *`matlabroot`* is the MATLAB installation folder such as `C:\Program Files\MATLAB\R2015a`. • `-format` *`type`*: Output format of report. The allowed *`type`*s are`HTML`, `PDF`, `RTF`, `WORD`, and `XML`. • `-output-name` *`filename`*: Name of report. • `-results-dir` *`folder_paths`*: Path to folder containing your verification results. To generate a single report for multiple verifications, specify *`folder_paths`* as follows: `"folder1, folder2, ..., folderN"` where *`folder1, folder2, ...`* are paths to the folders that contain verification results. For example, `"C:\Recent\results,C:\Old"`

User Interface	Command Line
	If you do not specify a folder path, the software uses verification results from the current folder.

See Also

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

Related Examples

- "Customize Report Templates"

Customize Report Templates

This example shows how to customize the templates that you use for report generation. To customize the templates, you must have Simulink Report Generator™ software installed on your system.

In this section...

Create Custom Template

If you have Simulink Report Generator software on your system:

1 Open the Report Explorer from the MATLAB command prompt:

```
report
```

2 Select **File** > **Open** to open the template that you want to customize.

3 Navigate to *`Matlab_Install`*`/polyspace/toolbox/psrptgen/templates` where *`Matlab_Install`* is the MATLAB installation folder. Use the `matlabroot` command to find the folder location.

4 Modify the template using the options on the **Report Options** pane.

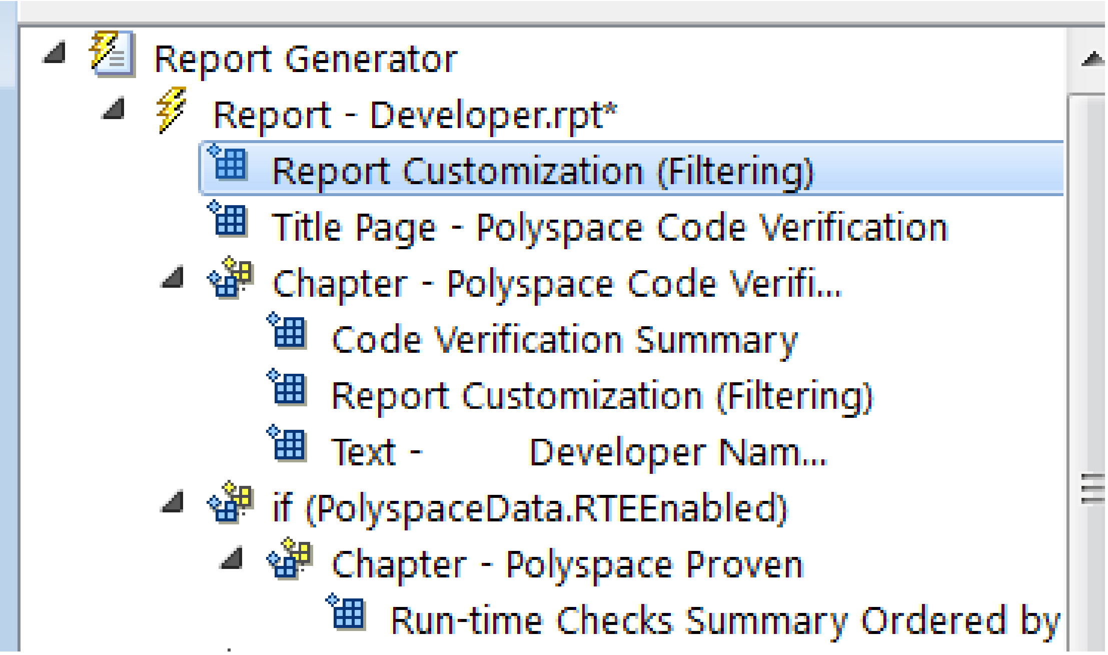
5 Save the modified template as a `.rpt` file.

Apply Global Filters in Template

1 In the Report Explorer, open the template that you want to customize. For instance, **Developer.rpt**.

2 On the **Name** pane, under the **Polyspace** node, select **Report Customization (Filtering)**.

3 Drag this component above the **Title Page** component that is located under the **Report-Developer.rpt** node.



Report Generator
Report - Developer.rpt*
Report Customization (Filtering)
Title Page - Polyspace Code Verification
Chapter - Polyspace Code Verifi...
Code Verification Summary
Report Customization (Filtering)
Text - Developer Nam...
if (PolyspaceData.RTEEnabled)
Chapter - Polyspace Proven
Run-time Checks Summary Ordered by

4 On the **Report Customization (Filtering)** pane in the right side of the Report Explorer, specify your filters. For example:

- To include **Division by zero** checks, under **Advanced filters**, in the **Check types to include** field, enter `ZDV`.
- To exclude **Division by zero** checks, under **Advanced filters**, in the **Check types to include** field, enter the regular expression `^(?!ZDV).*`.
- To include the file `main.c`, under **Advanced filters**, in the **Files to include** field, enter `main.c`.
- To exclude the file `main.c`, under **Advanced filters**, in the **Files to include** field, enter the regular expression `^(?!main.c).*`.

In each text box, specify one filter per line.

For more information, see:

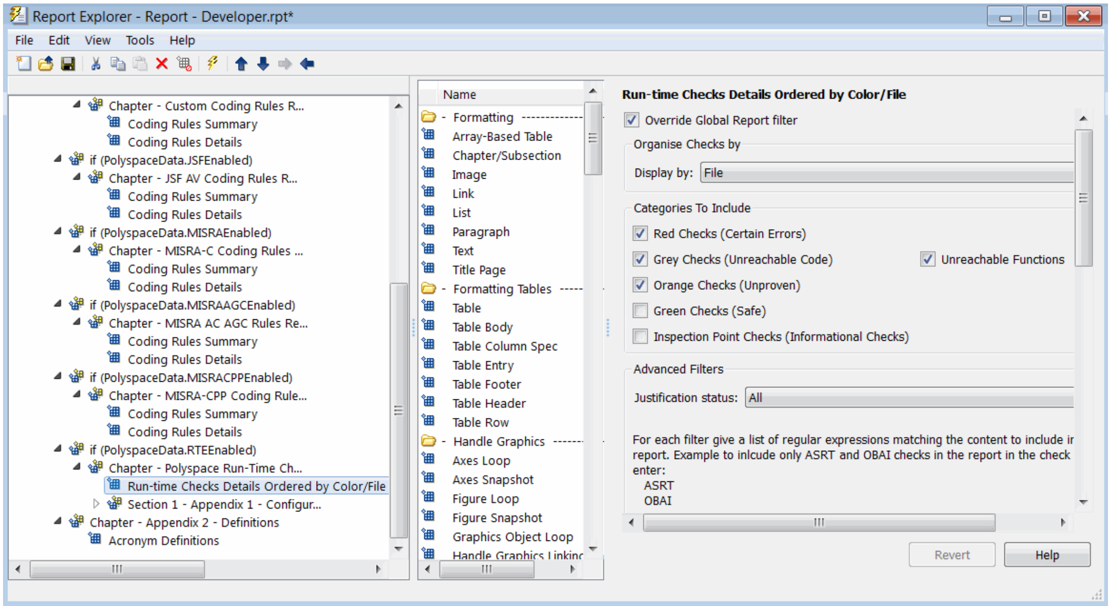
- "Check and Code Metric Acronyms"

- “Regular Expressions”

Override Global Filters

You can override some of the global filters using the **Run-time Check Details Ordered by Color/File** component. For example, you can have a report chapter that contains NIV checks even though NIV checks are excluded by the global filters.

1 Select the **Run-time Check Details Ordered by Color/File** component.



Run-time Checks Details Ordered by Color/File

☑ Override Global Report filter

Organise Checks by

Display by: File

Categories To Include

☑ Red Checks (Certain Errors)

☑ Grey Checks (Unreachable Code) ☑ Unreachable Functions

☑ Orange Checks (Unproven)

☐ Green Checks (Safe)

☐ Inspection Point Checks (Informational Checks)

Advanced Filters

Justification status: All

For each filter give a list of regular expressions matching the content to include in report. Example to inlcude only ASRT and OBAI checks in the report in the check enter:

ASRT
OBAI

2 On the right of the dialog box, select the **Override Global Report filter** check box.

3 Specify your filters for this component. For example, in the **Check types to include** field, enter NIV.

4 Save the template.

For more information on the components available for customizing report, see “Code Verification” in the Simulink Report Generator documentation.

Use Custom Template

1. Open your results in the Polyspace interface.
2. Select **Reporting** > **Run Report**.
3. Click **Browse**.
4. Navigate to the location where you saved your template `.rpt` file.
5. Select the file and click **OK**. Under **Select Reports**, you see your template.
6. Select the template and click **Run Report**.

See Also

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

Related Examples

- "Generate Report"

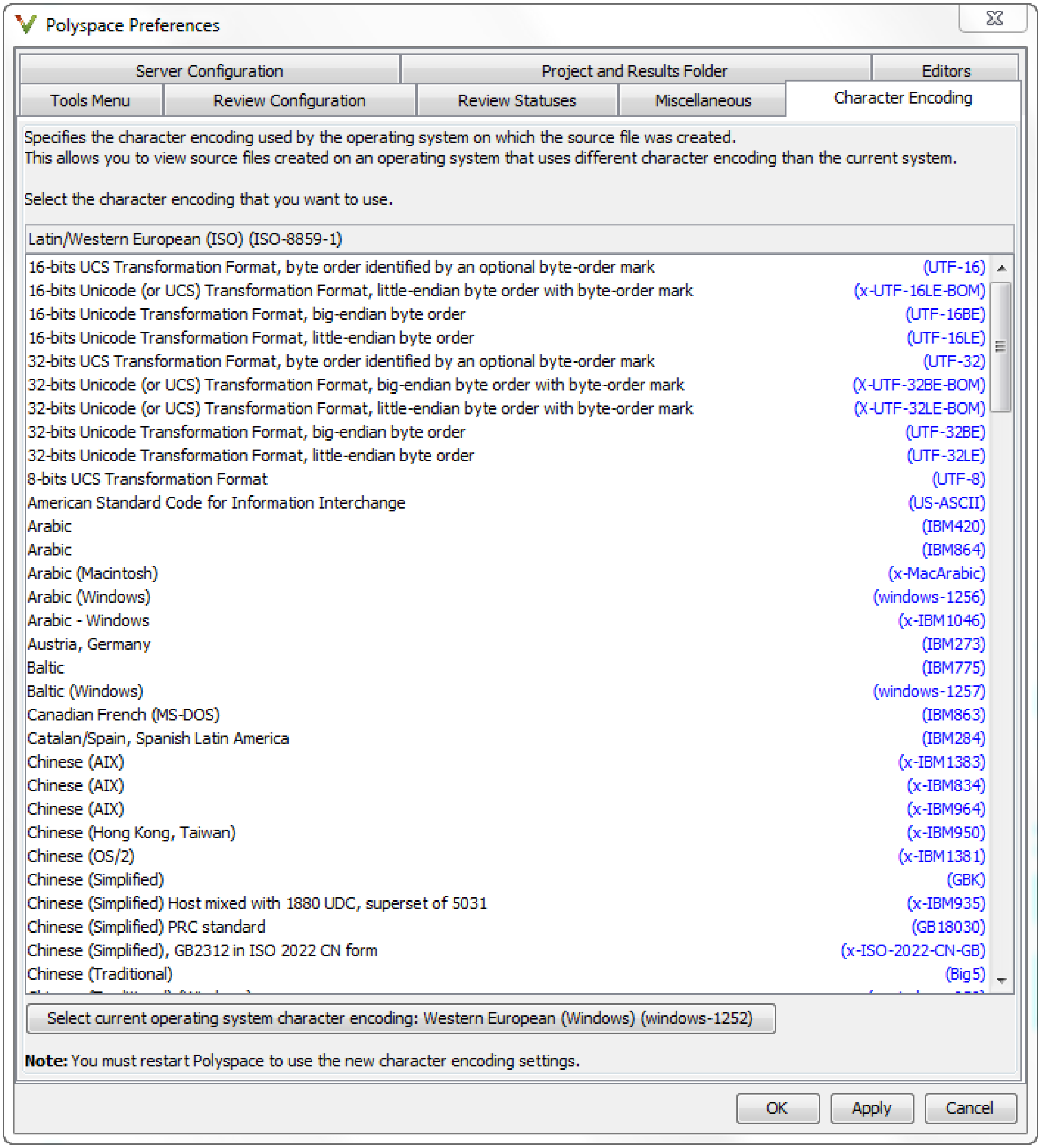
Set Character Encoding Preferences

If the source files that you want to verify are created on an operating system that uses different character encoding than your current system (for example, when viewing files containing Japanese characters), you receive an error message when you view the source file or run certain macros.

The **Character encoding** option allows you to view source files created on an operating system that uses different character encoding than your current system.

To set the character encoding for a source file:

1. Select **Tools** > **Preferences**.
2. In the Polyspace Preferences dialog box, select the **Character encoding** tab.



Specifies the character encoding used by the operating system on which the source file was created.
This allows you to view source files created on an operating system that uses different character encoding than the current system.

Select the character encoding that you want to use.

Latin/Western European (ISO) (ISO-8859-1)

16-bits UCS Transformation Format, byte order identified by an optional byte-order mark	(UTF-16)
16-bits Unicode (or UCS) Transformation Format, little-endian byte order with byte-order mark	(x-UTF-16LE-BOM)
16-bits Unicode Transformation Format, big-endian byte order	(UTF-16BE)
16-bits Unicode Transformation Format, little-endian byte order	(UTF-16LE)
32-bits UCS Transformation Format, byte order identified by an optional byte-order mark	(UTF-32)
32-bits Unicode (or UCS) Transformation Format, big-endian byte order with byte-order mark	(X-UTF-32BE-BOM)
32-bits Unicode (or UCS) Transformation Format, little-endian byte order with byte-order mark	(X-UTF-32LE-BOM)
32-bits Unicode Transformation Format, big-endian byte order	(UTF-32BE)
32-bits Unicode Transformation Format, little-endian byte order	(UTF-32LE)
8-bits UCS Transformation Format	(UTF-8)
American Standard Code for Information Interchange	(US-ASCII)
Arabic	(IBM420)
Arabic	(IBM864)
Arabic (Macintosh)	(x-MacArabic)
Arabic (Windows)	(windows-1256)
Arabic - Windows	(x-IBM1046)
Austria, Germany	(IBM273)
Baltic	(IBM775)
Baltic (Windows)	(windows-1257)
Canadian French (MS-DOS)	(IBM863)
Catalan/Spain, Spanish Latin America	(IBM284)
Chinese (AIX)	(x-IBM1383)
Chinese (AIX)	(x-IBM834)
Chinese (AIX)	(x-IBM964)
Chinese (Hong Kong, Taiwan)	(x-IBM950)
Chinese (OS/2)	(x-IBM1381)
Chinese (Simplified)	(GBK)
Chinese (Simplified) Host mixed with 1880 UDC, superset of 5031	(x-IBM935)
Chinese (Simplified) PRC standard	(GB18030)
Chinese (Simplified), GB2312 in ISO 2022 CN form	(x-ISO-2022-CN-GB)
Chinese (Traditional)	(Big5)

Select current operating system character encoding: Western European (Windows) (windows-1252)

Note: You must restart Polyspace to use the new character encoding settings.

3. Select the character encoding used by the operating system on which the source file was created.
4. Click **OK**.
5. Close and restart the Polyspace verification environment to use the new character encoding settings.

Reviewing Checks

Review and Fix Absolute Address Checks

Follow one or more of these steps until you determine a fix for the **Absolute Address** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see Absolute address.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- “Review Red Checks”
- “Review Orange Checks”

Tip This check can be turned off. To reduce the number of orange checks, if you trust that all absolute addresses in your code are valid, you can turn off this check. For more information, see “Green absolute address checks (C/C++)”.

For best use of this check, turn off this check during initial stages of development. During integration stage, turn on this check and detect all uses of absolute memory addresses. Browse through them and make sure that the addresses are valid.

---

1. Select the check on the **Results Summary** pane.

   The **Source** pane displays the code operation containing the absolute address. Because Polyspace cannot determine if absolute addresses in your code are valid, it produces the orange **Absolute Address** check.
2. If you determine that the address is valid, add a comment and justification in your result or code.

   - To add a justification in your result, see “Add Review Comments to Results”.
   - To add a justification in your code, see “Add Review Comments to Code”.

Review and Fix Correctness Condition Checks

Follow one or more of these steps until you determine a fix for the **Correctness Condition** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see Correctness condition.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

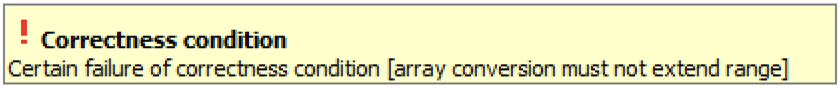
In this section...
"Step 1: Interpret Check Information" on page 10-3
"Step 2: Determine Root Cause of Check" on page 10-6
"Step 4: Trace Check to Polyspace Assumption" on page 10-8

Step 1: Interpret Check Information

On the **Results Summary** pane, select the check. View the cause of check on the **Check Details** pane. The following list shows some of the possible causes:

- An array is converted to another array of larger size.

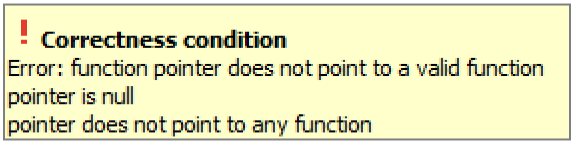
  In the following example, a red check occurs because an array is converted to another array of larger size.

  ! **Correctness condition**
  Certain failure of correctness condition [array conversion must not extend range]

- When dereferenced, a function pointer has value NULL.

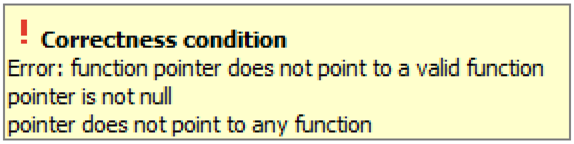
  In the following example, a red check occurs because, when dereferenced, a function pointer has value NULL.



! Correctness condition
Error: function pointer does not point to a valid function
pointer is null
pointer does not point to any function

- When dereferenced, a function pointer does not point to a function.

  In the following example, a red check occurs because Polyspace cannot determine if a function pointer points to a function when dereferenced. This situation can occur if, for instance, you assign an absolute address to the function pointer.



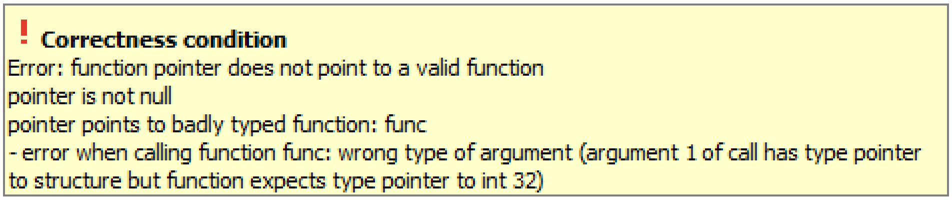
! Correctness condition
Error: function pointer does not point to a valid function
pointer is not null
pointer does not point to any function

- A function pointer points to a function, but the argument types of the pointer and the function do not match. For example:

  ```
  typedef int (*typeFuncPtr) (complex*);
  int func(int* x);
  .
  .
  typeFuncPtr funcPtr = &func;
  ```

  In the following example, a red check occurs because:

  - The function pointer points to a function `func`.
  - `func` expects an argument of type `int`, but the corresponding argument of the function pointer is a structure.



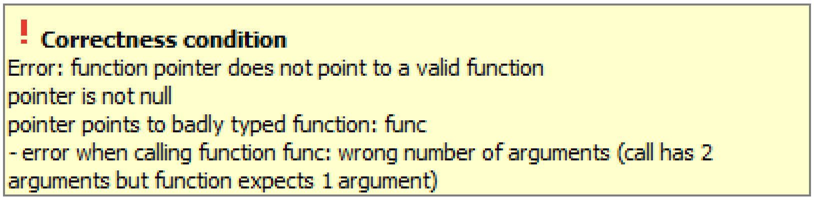
! Correctness condition
Error: function pointer does not point to a valid function
pointer is not null
pointer points to badly typed function: func
- error when calling function func: wrong type of argument (argument 1 of call has type pointer to structure but function expects type pointer to int 32)

- A function pointer points to a function, but the argument numbers of the pointer and the function do not match. For example:

```
typedef int (*typeFuncPtr) (int, int);
int func(int);
.
.
typeFuncPtr funcPtr = &func;.
```

In the following example, a red check occurs because:

- The function pointer points to a function `func`.
- `func` expects one argument but the function pointer has two arguments.



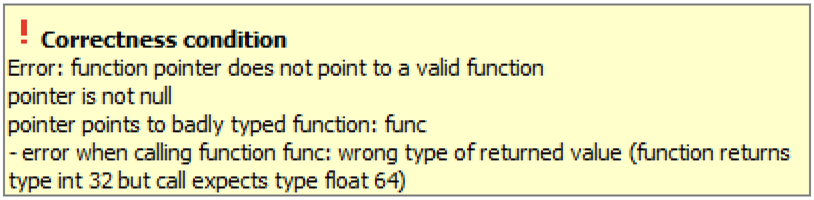
! **Correctness condition**
Error: function pointer does not point to a valid function
pointer is not null
pointer points to badly typed function: func
- error when calling function func: wrong number of arguments (call has 2 arguments but function expects 1 argument)

- A function pointer points to a function, but the return types of the pointer and the function do not match. For example:

```
typedef double (*typeFuncPtr) (int);
int func(int);
.
.
typeFuncPtr funcPtr = &func;
```

In the following example, a red check occurs because:

- The function pointer points to a function `func`.
- `func` returns an `int` value, but the return type of the function pointer is `double`.

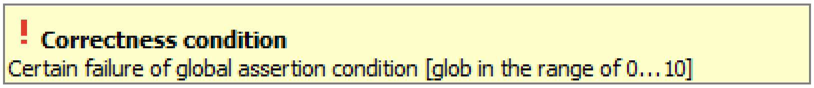


! **Correctness condition**
Error: function pointer does not point to a valid function
pointer is not null
pointer points to badly typed function: func
- error when calling function func: wrong type of returned value (function returns type int 32 but call expects type float 64)

- The value of a variable falls outside the range that you specify through the **Global Assert** mode. See “Constrain Global Variables”.

In the following example, a red check occurs because:

- You specify a range 0...10 for the variable `glob`.
- The value of the variable falls outside this range.



! **Correctness condition**
Certain failure of global assertion condition [glob in the range of 0…10]

Step 2: Determine Root Cause of Check

Based on the check information on the **Check Details** pane, perform further steps to determine the root cause.

Check Information	How to Determine Root Cause
An array is converted to another array of larger size.	**1** To determine the array sizes, see the definition of each array variable. Right-click the variable and select **Go To Definition**. **2** If you dynamically allocate memory to an array, it is possible that their sizes are not available during definition. Browse through all instances of the array variable to find where you allocate memory to the array. **a** Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. **b** On the **Search** pane, select the previous instances.
Issues when dereferencing a function pointer:	**1** Find the location where you assign the function pointer to a function.

Check Information	How to Determine Root Cause
• The function pointer has value `NULL` when dereferenced. • The function pointer does not point to a function when dereferenced. • The function pointer points to a function, but the argument types of the pointer and the function do not match. • The function pointer points to a function, but the argument numbers of the pointer and the function do not match. • The function pointer points to a function, but the return types of the pointer and the function do not match.	**a** Right-click the function pointer. Select **Search For All References**. All instances of the function pointer appear on the **Search** pane with the current instance highlighted. **b** On the **Search** pane, select the previous instances. **2** Determine the argument and return types of the function pointer type and the function. Identify if there is a mismatch between the two. For instance, in the following example, determine the argument and return types of `typeFuncPtr` and `func`. `typeFuncPtr funcPtr = func;` **a** Right-click the function pointer type and select **Go To Definition**. **b** Right-click the function and select **Go To Definition**. If the definition does not exist, this option shows the function stub definition instead. In this case, find the function declaration.
The value of a variable falls outside the range that you specify through the **Global Assert** mode.	Browse through all previous instances of the global variable. Identify a suitable point to constrain the variable. **1** Right-click the variable. Select **Show In Variable Access View**. **2** On the **Variable Access** pane, select each instance of the variable.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

Review and Fix Division by Zero Checks

Follow one or more of these steps until you determine a fix for the **Division by Zero** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see Division by zero.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

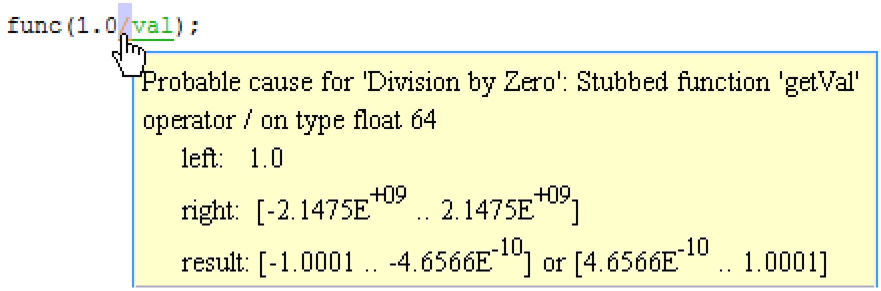
For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

In this section...

Step 1: Interpret Check Information

Place your cursor on the / or % operation that causes the **Division by Zero** error.



```
func(1.0/val);
```

Probable cause for 'Division by Zero': Stubbed function 'getVal'
operator / on type float 64
left: 1.0
right: [$-2.1475E^{+09}$ .. $2.1475E^{+09}$]
result: [-1.0001 .. $-4.6566E^{-10}$] or [$4.6566E^{-10}$ .. 1.0001]

Obtain the following information from the tooltip:

- The values of the right operand (denominator).

In the preceding example, the right operand, `val`, has a range that contains zero.

Possible fix:: To avoid the division by zero, perform the division only if `val` is not zero.

Integer	Floating-point

`if(val != 0)`
`    func(1.0/val);`
`else`
`    /* Error handling */`
	`#define eps 0.0000001`
	`.`
	`.`
	`if(val < -eps || val > eps)`
	`    func(1.0/val);`
	`else`
	`    /* Error handling */`

- The probable root cause for division by zero, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

  Possible fix: To avoid the division by zero, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. To specify a range:

  1. Select the ? icon on the **Check Details** pane.
  2. On the **Orange Sources** pane, find the stubbed function `getVal`. On the **Suggestion** column, click **Add DRS**.
  3. Specify your range on the **Data Range Configuration** pane in the **Init Range** column.

Step 2: Determine Root Cause of Check

Before a `/` or `%` operation, test if the denominator is zero. Provide appropriate error handling if the denominator is zero.

Only if you do not expect a zero denominator, determine root cause of check. Trace the data flow starting from the denominator variable. Identify a point where you can specify a constraint to prevent the zero value.

In the following example, trace the data flow starting from `arg2`:

```
void foo() {
 double time = readTime();
 double dist = readDist();
  .
  .
  bar(dist,time);
}

void bar(double arg1, double arg2) {
 double vel;
 vel=arg1/arg2;
}
```

You might find that:

1. `bar` is called with full-range of values.

   Possible fix: Call `bar` only if its second argument `time` is greater than zero.
2. `time` obtains a full-range of values from `readTime`.

   Possible fix: Constrain the return value of `readTime`, either in the body of `readTime` or through the Polyspace Data Range Specification interface, if you do not have the definition of `readTime`.

   For more information on Data Range Specification, see “Specify Constraints”.

To trace the data flow, select the check and note the information on the **Check Details** pane.

- If the **Check Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Check Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the operand variable.

     Example: The value of `arg2` is written from the value of `time` in `bar`.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

Example: At `bar(dist,time)`, you find that `time` has a full-range of values. Therefore, you trace `time`.

3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

   Example: The previous write operation on `time` is `time=readTime()`. You can choose to specify your constraint on the return value of `readTime`.

Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◂ and read operations with ▸.
Function argument	1 On the **Check Details** pane, select the *fx* button.

Variable	How to Find Previous Instances of Variable

`void func(..,int arg,..`
`.`
`.`
`}`
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	2 Select a calling function name. You go to the call to `func` in your source.
	3 Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

Step 3: Look for Common Causes of Check

1. For a variable that you expect to be non-zero, see if you test the variable in your code to exclude the zero value.

   Otherwise, Polyspace cannot determine that the variable has non-zero values. You can also specify constraints outside your code. See “Specify Constraints”.

2. If you test the variable to exclude its zero value, see if the test occurs in a reduced scope compared to the scope of the division.

   For example, a statement `assert(var !=0)` occurs in an `if` or `while` block, but a division by `var` occurs outside the block. If the code does not enter the `if` or `while` block, the `assert` does not execute. Therefore, outside the `if` or `while` block, Polyspace assumes that `var` can still be zero.

   Possible fix:

   - Investigate why the test occurs in a reduced scope. In the above example, see if you can place the statement `assert(var !=0)` outside the `if` or `for` block.
   - If you expect the `if` or `while` block to always execute, investigate when it does not execute.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

For instance, you are using a volatile variable in your code. Then:

1. Polyspace assumes that the variable is full-range at every step in the code. The range includes zero.
2. A division by the variable can cause **Division by Zero** error.
3. If you know that the variable takes a non-zero value, add a comment and justification describing why you did not change your code.

For more information, see “Polyspace Software Assumptions”.

Note: Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Function Not Called Checks

Follow one or more of these steps until you determine a fix for the **Function Not Called** check. There are multiple ways to fix this check. For a description of the check and code examples, see Function not called.

If you determine that the check represents defensive code or a function that is part of a library, add a comment and justification in your result or code explaining why you did not change your code. To:

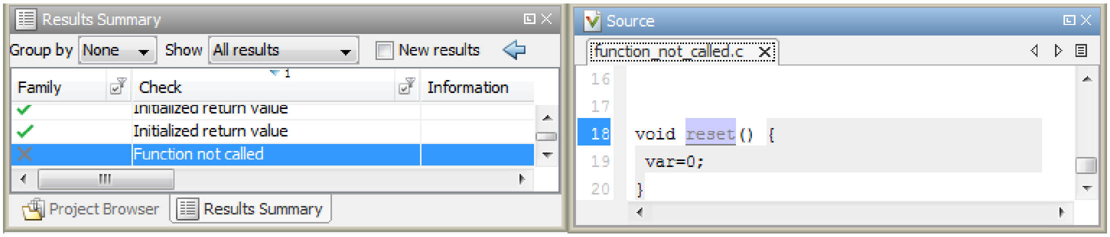
- Add justification in your result, see “Add Review Comments to Results”.
- Add justification in your code, see “Comment Code for Known Defects”.

Note: This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see “Detect uncalled functions (C/C++)”.

In this section...

Interpret Check Information

On the **Results Summary** pane, select the check. On the **Source** pane, the body of the function is highlighted in gray.



Family	Check	Information
✓	Initialized return value	
✓	Initialized return value	
✗	Function not called	

```
void reset() {
 var=0;
}
```

Determine Root Cause of Check

1. Search for the function name and see if you can find a call to the function in your code.

   On the **Search** pane, enter the function name. From the drop-down list beside the search field, select **Source**.

   Possible Fix: If you do not find a call to the function, determine why the function definition exists in your code.
2. If you find a call to the function, see if it occurs in the body of another uncalled function.

   Possible Fix: Investigate why the latter function is not called.
3. See if you call the function indirectly, for example, through function pointers.

   If the indirection is too deep, Polyspace sometimes cannot determine that a certain function is called.

   Possible Fix: If Polyspace cannot determine that you are calling a function indirectly, you must verify the function separately. You do not need to write a new `main` function for this other verification. Polyspace can generate a `main` function if you do not provide one in your source. You can change the `main` generation options if needed. For more information on the options, see:

   - C code: “Code Prover Verification”
   - C++ code: “Code Prover Verification”

Look for Common Causes of Check

1. Determine if you intended to call the function but used another function instead.
2. Determine if you intended to replace some code with a function call. You wrote the function definition, but forgot to replace the original code with the function call.

   If this situation occurs, you are likely to have duplicate code.
3. See if you intend to call the function from yet unwritten code. If so, retain the function definition.
4. For code intended for multitasking, see if you have specified all your entry point functions.

To see the options used for the result, select **Window** > **Show/Hide View** > **Configuration**. View the entry points you specified under the **Multitasking** node.

For more information, see “Entry points (C/C++)”.

5 For code intended for multitasking, see if your `main` function contains an infinite loop. Polyspace Code Prover requires that your `main` function must complete execution before the other entry points begin.

For more information, see “Model Tasks if main Contains Infinite Loop”.

Review and Fix Function Not Reachable Checks

Follow one or more of these steps until you determine a fix for the **Function Not Reachable** check. There are multiple ways to fix this check. For a description of the check and code examples, see Function not reachable.

If you determine that the check represents defensive code, add a comment and justification in your result or code explaining why you did not change your code. To:

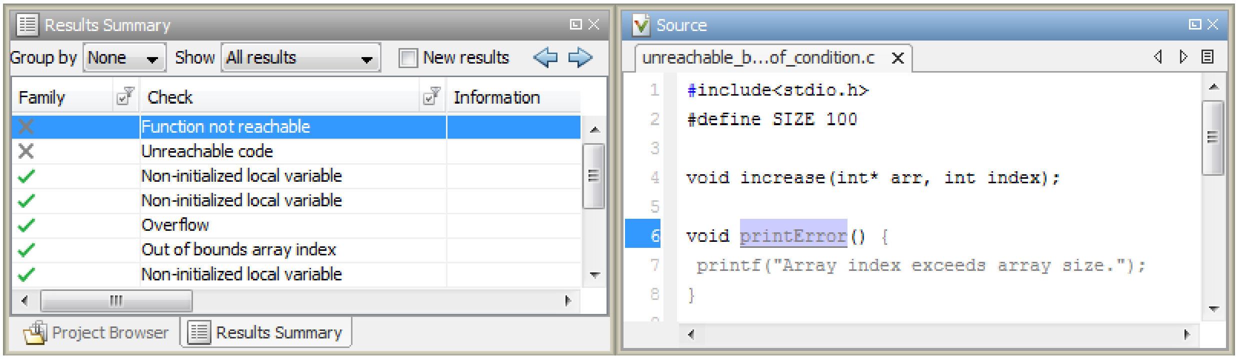
- Add justification in your result, see “Add Review Comments to Results”.
- Add justification in your code, see “Comment Code for Known Defects”.

Note: This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see “Detect uncalled functions (C/C++)”.

In this section...

Step 1: Interpret Check Information

Select the check on the **Results Summary** pane. On the **Source** pane, you can see the function definition in gray.



Family	Check	Information
✗	Function not reachable	
✗	Unreachable code	
✓	Non-initialized local variable	
✓	Non-initialized local variable	
✓	Overflow	
✓	Out of bounds array index	
✓	Non-initialized local variable	

```
#include<stdio.h>
#define SIZE 100

void increase(int* arr, int index);

void printError() {
 printf("Array index exceeds array size.");
}
```

Step 2: Determine Root Cause of Check

Determine where the function is called and review why all the function call sites are unreachable.

1 Select the check on the **Results Summary** pane.

2 On the **Check Details** pane, click the fx button.

On the **Call Hierarchy** pane, you see the callers of the function denoted by ◀ .

3 On the **Call Hierarchy** pane, select each caller

This action takes you to the function call on the **Source** pane.

4 See if the caller itself is called from unreachable code. If the caller definition is entirely in gray on the **Source** pane, it is called from unreachable code. Follow the same investigation process, starting from step 1, for the caller.

5 Otherwise, investigate why the section of code from which you call the function is unreachable.

The code can be unreachable because it follows a red check or because it contains the gray **Unreachable code** check.

- If a red check occurs, fix your code to remove the check. See “Review Red Checks”.
- If a gray **Unreachable code** check occurs, review the check and determine if you must fix your code. See “Review and Fix Unreachable Code Checks”.

Note: If you do not see a caller name on the **Call Hierarchy** pane, determine if you are calling the function indirectly, for example through a function pointer. Determine if a mismatch occurs between the function pointer declaration and the function call through the pointer.

Polyspace places a red **Correctness condition** check on the indirect call if a mismatch occurs. To detect a mismatch in indirect function calls, look for the red **Correctness condition** check on the **Results Summary** pane. For more information, see Correctness condition.

Review and Fix Illegally Dereferenced Pointer Checks

Follow one or more of these steps until you determine a fix for the **Illegally Dereferenced Pointer** check. There are multiple ways to fix this check. For a description of the check and code examples, see Illegally dereferenced pointer.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- “Review Red Checks”
- “Review Orange Checks”

In this section...

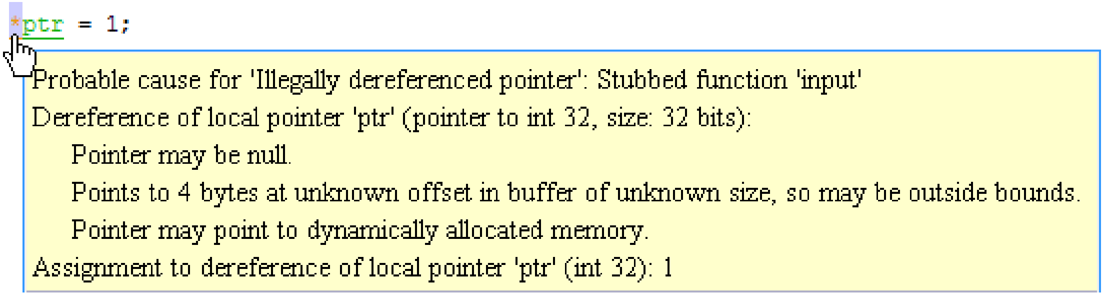
Step 1: Interpret Check Information

Place your cursor on the dereference operator.

Obtain the following information from the tooltip:

- Whether the pointer can be `NULL`.

  In the following example, `ptr` can be `NULL` when dereferenced.



```
*ptr = 1;
```

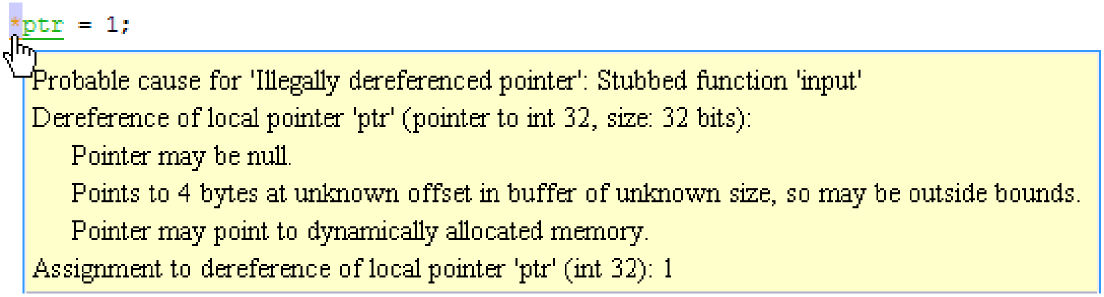
Probable cause for 'Illegally dereferenced pointer': Stubbed function 'input'
Dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    Pointer may be null.
    Points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds.
    Pointer may point to dynamically allocated memory.
Assignment to dereference of local pointer 'ptr' (int 32): 1

Possible fix:: Dereference `ptr` only if it is not `NULL`.

```
if(ptr !=NULL)
    *ptr = 1;
else
    /* Alternate action */
```

- Whether the pointer points to dynamically allocated memory.

  In the following example, `ptr` can point to dynamically allocated memory. It is possible that the dynamic memory allocation operator returns `NULL`.



```
*ptr = 1;
```

Probable cause for 'Illegally dereferenced pointer': Stubbed function 'input'
Dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    Pointer may be null.
    Points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds.
    Pointer may point to dynamically allocated memory.
Assignment to dereference of local pointer 'ptr' (int 32): 1

Possible fix:: Check the return value of the memory allocation operator for `NULL`.

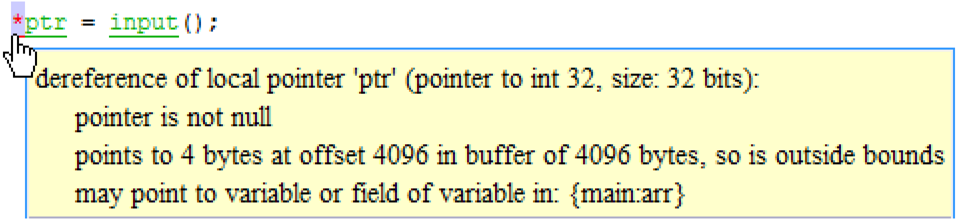
```
ptr = (char*) malloc(i);
if(ptr==NULL)
  /* Error handling*/
else {
  .
 .
 *ptr=0;
```

```
   .
   .
}
```

- Whether pointer points outside allowed bounds. A pointer points outside bounds when the sum of pointer size and offset is greater than buffer size.

  In the following example, the offset size (4096 bytes) together with pointer size (4 bytes) is greater than the buffer size (4096 bytes). If the pointer points to an array:

  - The buffer size is the array size.
  - The offset is the difference between the beginning of the array and the current location of the pointer.

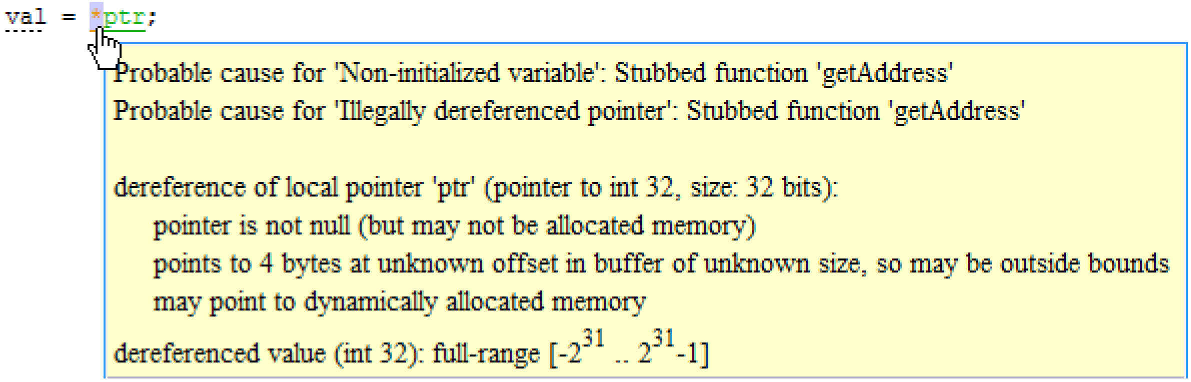
  ```
  *ptr = input();
  ```

  dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
      pointer is not null
      points to 4 bytes at offset 4096 in buffer of 4096 bytes, so is outside bounds
      may point to variable or field of variable in: {main:arr}

  Possible fix: Investigate why the pointer points outside the allowed buffer.

- Whether pointer can point outside allowed bounds because buffer size is unknown.

  In the following example, the buffer size is unknown.



```
val = *ptr;
```

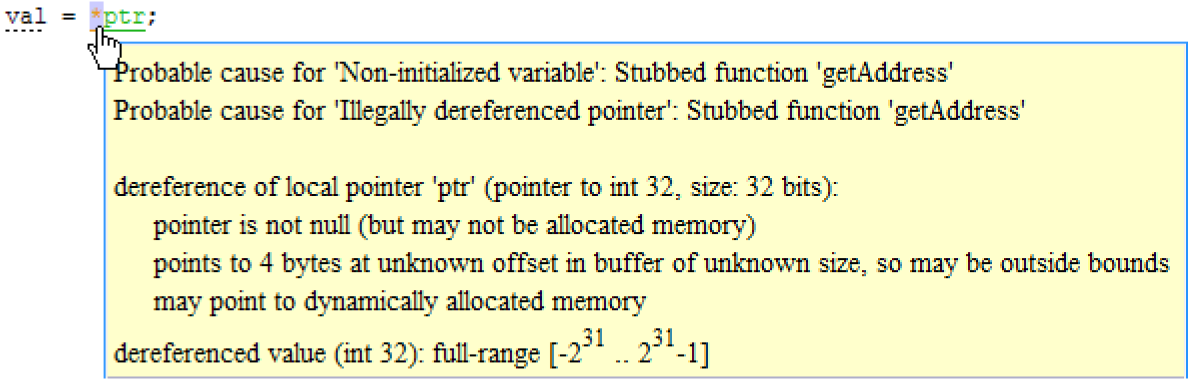
Probable cause for 'Non-initialized variable': Stubbed function 'getAddress'
Probable cause for 'Illegally dereferenced pointer': Stubbed function 'getAddress'

dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    pointer is not null (but may not be allocated memory)
    points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds
    may point to dynamically allocated memory
dereferenced value (int 32): full-range $[-2^{31} .. 2^{31}-1]$

  Possible fix: Investigate whether the pointer is assigned:

   - The return value of an undefined function.
   - The return value of a dynamic memory allocation function. Sometimes, Polyspace cannot determine the buffer size from the dynamic memory allocation.
   - Another pointer of a different type, for instance, `void*`.
- The probable root cause for illegal pointer dereference, if indicated in the tooltip.

  In the following example, the software identifies a stubbed function, `getAddress`, as probable cause.



```
val = *ptr;
```

Probable cause for 'Non-initialized variable': Stubbed function 'getAddress'
Probable cause for 'Illegally dereferenced pointer': Stubbed function 'getAddress'

dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    pointer is not null (but may not be allocated memory)
    points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds
    may point to dynamically allocated memory
dereferenced value (int 32): full-range [-2^{31} .. $2^{31}-1$]

Possible fix: To avoid the illegally dereferenced pointer, constrain the return value of `getAddress`. For instance, specify that `getAddress` returns a pointer to a 10-element array. To specify this constraint:

1. On the **Check Details** pane, select the ? icon.
2. On the **Orange Sources** pane, in the **Suggestion** column for `getAddress`, click **Add DRS**.
3. Specify your constraint on the **Data Range Configuration** pane in the **# Allocated Objects** column.

   For more information on the columns in this pane, see “Constraints”.

Step 2: Determine Root Cause of Check

Select the check and note the information on the **Check Details** pane.

- If the **Check Details** pane shows the sequence of instructions that lead to the check, select each instruction and trace back to the root cause.
- If the **Check Details** pane shows the line number of probable cause for the check, right-click the **Source** pane. Select **Go To Line**.
- Otherwise, based on the nature of the error, use one of the following methods to find the root cause.

Error	How to Find Root Cause

Pointer can be `NULL`.	Find an execution path where the pointer is assigned the value NULL or not assigned a definite address.
	1 Right-click the pointer and select **Search For All References**.
	2 Find each previous instance where the pointer is assigned an address.
	3 For each instance, on the **Source** pane, place your cursor on the pointer. The tooltip indicates whether the pointer can be `NULL`.
	Possible fix: If the pointer can be `NULL`, place a check for `NULL` immediately after the assignment.
	`if(ptr==NULL)`
	`  /* Error handling*/`
	`else {`
	`  .`
	`  .`
	` }`
	4 If the pointer is not `NULL`, see if the assignment occurs only in a branch of a conditional statement. Investigate when that branch does not execute.
	Possible fix: Assign a valid address to the pointer in all branches of the conditional statement.

Pointer can point to dynamically allocated memory.	Identify where the allocation occurs. **1** Right-click the pointer and select **Search For All References**.

Error	How to Find Root Cause
	2 Find the previous instance where the pointer receives a value from a dynamic memory allocation function such as `malloc`. **Possible fix:** After the allocation, test the pointer for `NULL`.
Pointer can point outside bounds allowed by the buffer.	**1** Find the allowed buffer. **a** On the **Search** tab, enter the name of the variable that the pointer points to. You already have this name from the tooltip on the check. **b** Search for the variable definition. Typically, this is the first search result. If the variable is an array, note the array size. If the variable is a structure, search for the structure type name on the **Search** tab and find the structure definition. Note the size of the structure field that the pointer points to. **2** Find out why the pointer points outside the allowed buffer. **a** Right-click the pointer and select **Search For All References**. **b** Identify any increment or decrement of the pointer. See if you intended to make the increment or decrement. **Possible fix:** Remove unintended pointer arithmetic. To avoid pointer arithmetic that takes a pointer outside allowed buffer, use a reference pointer to store its initial value. After every arithmetic operation on your pointer, compare it with the reference pointer to see if the difference is outside the allowed buffer.

Step 3: Look for Common Causes of Check

1. If you use pointers for moving through an array, see if you can use an array index instead.

   To avoid use of pointer arithmetic in your code, look for violations of MISRA C: 2004 rule 17.4 or MISRA C: 2012 rule 18.4. For more information, see “Select Specific MISRA or JSF Coding Rules”.
2. See if you use pointers for moving through the fields of a structure.

   Polyspace does not allow the pointer to one field of a structure to point to another field. To allow this behavior, use the option “Enable pointer arithmetic across fields (C)”.
3. See if you are dereferencing a pointer that points to a structure but does not have sufficient memory for all its fields. Such a pointer usually results from type-casting a pointer to a smaller structure.

   Polyspace does not allow such dereference. To allow this behavior, use the option “Allow incomplete or partial allocation of structures (C)”.
4. If an orange check occurs in a function body, see if you are passing arrays of different sizes in different calls to the function.

   See if one particular call causes the orange check. For a tutorial, see “Identify Function Call Causing Orange Check”.
5. See if you are performing a cast between two pointers of incompatible sizes.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

For instance, the pointer receives an address from an undefined function. Then:

1. Polyspace assumes that the function can return `NULL`.

   Therefore, the pointer dereference is orange.

2. Polyspace also assumes an allowed buffer size based on the type of the pointer.

   If you increment the pointer, you exceed the allowed buffer. The pointer dereference that follows the increment is orange.
3. If you know that the function returns a non-`NULL` value or if you know the true allowed buffer, add a comment and justification in your code describing why you did not change your code.

For more information, see “Polyspace Software Assumptions”.

Note: Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Initialized Return Value Checks

Follow one or more of these steps until you determine a fix for the **Initialized Return Value** check. There are multiple ways to fix this check. For a description of the check and code examples, see Initialized return value.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

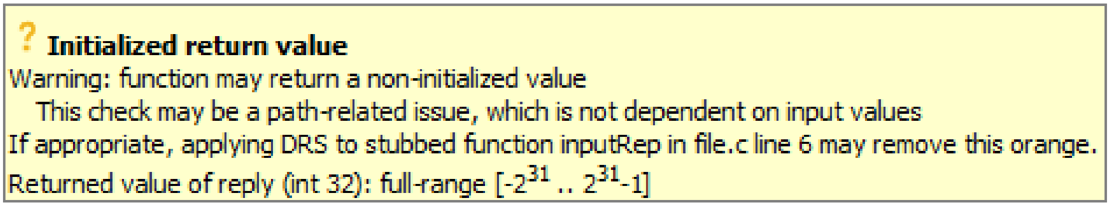
For the general workflow that applies to all checks, see:

- “Review Red Checks”
- “Review Orange Checks”

In this section...

Step 1: Interpret Check Information

Select the check on the **Results Summary** pane. On the **Check Details** pane, view further information about the check.



? **Initialized return value**

Warning: function may return a non-initialized value

This check may be a path-related issue, which is not dependent on input values

If appropriate, applying DRS to stubbed function inputRep in file.c line 6 may remove this orange.

Returned value of reply (int 32): full-range [-2^{31} .. 2^{31}-1]

View the probable cause of check, if mentioned on the **Check Details** pane.

In the preceding example, the software identifies a stubbed function, `inputRep`, as probable cause.

Possible fix: To avoid the check, constrain the argument or return value of `inputRep`. For instance, specify that `inputRep` returns values in a certain range, for example, `1..10`. To specify a range:

1. Select the ? icon on the **Check Details** pane.
2. On the **Orange Sources** pane, find the stubbed function `inputRep`. On the **Suggestion** column, click **Add DRS**.
3. Specify your range on the **Data Range Configuration** pane in the **Init Range** column.

Step 2: Determine Root Cause of Check

Determine the root cause of the check in the function body.

1. Navigate to the function definition.

   Right-click the function call containing the check. Select **Go To Definition**, if the option exists.
2. In the function body, check if a `return` statement occurs before the closing brace of the function.
3. If a `return` statement does not exist:

   a. On the **Search** pane, search for the word `return`, or manually scroll through the function body and look for `return` statements.

   b. For each `return` statement, determine if the statement appears in a scope smaller than function scope.

      For instance, a `return` statement occurs only in one branch of an `if-else` statement.

   Possible fix: See if you can place the `return` statement at the end of the function body. For instance, replace the following code

```
int func(int ch) {
    switch(ch) {
        case 1: return 1;
        break;
        case 2: return 2;
        break;
    }
}
```

with

```
int func(int ch) {
    int temp;
    switch(ch) {
        case 1: temp = 1;
        break;
        case 2: temp = 2;
        break;
    }
    return temp;
}
```

For information on how to enforce this practice, see Number of Return Statements.

Step 3: Look for Common Causes of Check

1. See if the `return` statements appear in `if-else`, `for` or `while` blocks. Identify conditions when the function does not enter the block.

   For instance, the function might not enter a `while` block for certain function inputs.

   Possible fix:

   - See if you can place the `return` statement at the end of the function body.
   - Otherwise, determine how to avoid the condition when the function does not enter the block.

     For instance, if a function does not enter a block for certain inputs, see if you must pass different inputs.
2. See if you have code constructs such as `goto` that interrupt the normal control flow. See if there are conditions when `return` statements in your function cannot be reached because of these code constructs.

   Possible fix:

   - Avoid `goto` statements. For information on how to enforce this practice, see Number of Goto Statements.

- Otherwise, determine how to avoid the condition when `return` statements in your function cannot be reached.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

For instance, you have a `return` statement in branches of an `if-elseif` block but you do not have the final `else` block. The condition you are testing in the `if-elseif` blocks involve variables obtained from an undefined function. Then:

1. Polyspace assumes that for certain values of those variables, none of the `if-elseif` blocks are entered.
2. Therefore, it is possible that the `return` statements are not reached.
3. If you know that those values of the variables do not occur, add a comment and justification describing why you did not change your code.

For more information, see “Polyspace Software Assumptions”.

Review and Fix Invalid Use of Standard Library Routine Checks

Follow one or more of these steps until you determine a fix for the **Invalid Use of Standard Library Routine** check. For a description of the check and code examples, see Invalid use of standard library routine.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

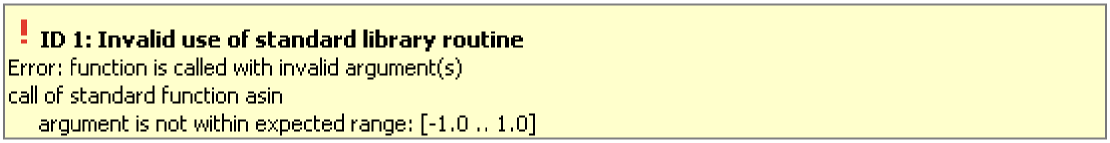
For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

In this section...
"Step 1: Interpret Check Information" on page 10-32
"Step 2: Trace Check to Polyspace Assumption" on page 10-34

Step 1: Interpret Check Information

Select the check on the **Results Summary** pane. View further information about the check on the **Check Details** pane. The check is red or orange because of invalid function arguments.



ID 1: Invalid use of standard library routine

```
Error: function is called with invalid argument(s)
call of standard function asin
    argument is not within expected range: [-1.0 .. 1.0]
```

The cause of a red or orange check depends on the standard library function that you use. The following table shows the possible causes for some of the commonly used functions.

Function	Cause of Red or Orange Check
`islower`, `isdigit`, and other character-handling functions in `ctype.h`	The value of the argument can be outside the range allowed for an `unsigned char` variable. **Note:** You can use the macro `EOF` as argument.

Function	Cause of Red or Orange Check	
Functions in `math.h`	`sqrt`	The value of the argument can be negative.
	`pow`	The first argument can be negative while the second argument is a non-integer.
	`exp`, `exp2`, or the hyperbolic functions	The argument can be so large that the result exceeds the value allowed for a `double`.
	`log`	The argument can be zero or negative.
	`asin` or `acos`	The argument can be outside the range [-1,1].
	`tan`	The argument can have a value of the form $\frac{\pi}{2} + n\pi$
	`acosh`	The argument can be less than 1.
	`atanh`	The argument can be greater than 1 or less than -1.
`fprintf`, `fscanf`, and other file handling functions	The file pointer argument can be non-readable. For example, it can be `NULL`.	
Functions that take string arguments	The string argument can be an invalid string. For example, it does not end with a terminating `'\0'`.	
`memmove` or `memcpy`	The third argument of this function specifies the number of bytes to copy from the second to the first argument. This number can exceed the memory allocated to the first or second argument.	

Step 2: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

For instance, you obtain a value from an undefined function and perform the `sqrt` operation on it. Then:

1. Polyspace assumes that the function can return a negative value.
2. Therefore, the software produces an orange **Invalid Use of Standard Library Routine** check on the `sqrt` function call.
3. If you know that the function returns a positive value, to avoid the orange, you can specify a constraint on the return value of your function. See “Constrain Stubbed Functions”. Alternately, add a comment and justification describing why you did not change your code.

For more information, see “Polyspace Software Assumptions”.

Review and Fix Non-initialized Local Variable Checks

Follow one or more of these steps until you determine a fix for the **Non-initialized Local Variable** check. There are multiple ways to fix this check. For a description of the check and code examples, see Non-initialized local variable.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

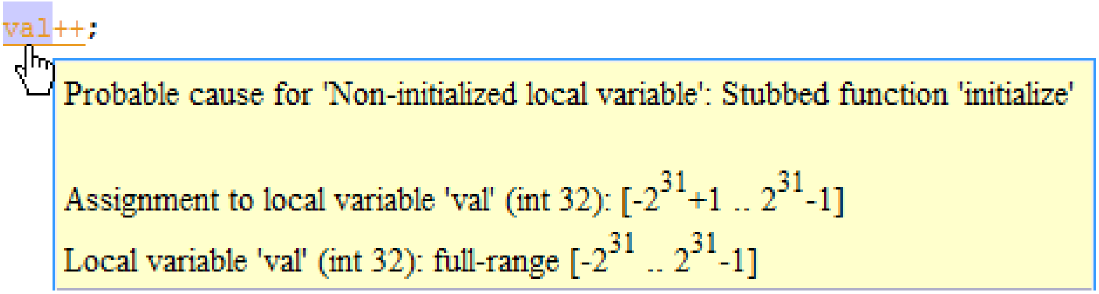
For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

In this section...

Step 1: Interpret Check Information

Place your cursor on the variable on which the **Non-initialized local variable** error appears.



```
val++;
```

Probable cause for 'Non-initialized local variable': Stubbed function 'initialize'

Assignment to local variable 'val' (int 32): $[-2^{31}+1 \; .. \; 2^{31}-1]$

Local variable 'val' (int 32): full-range $[-2^{31} \; .. \; 2^{31}-1]$

Obtain the probable root cause for the variable being non-initialized, if indicated in the tooltip.

In the preceding example, the software identifies a stubbed function, `initialize`, as probable cause.

Possible fix: To avoid the check, you can specify that `initialize` writes to its arguments.

1. Select the ? icon on the **Check Details** pane.
2. On the **Orange Sources** pane, find the stubbed function `initialize`. On the **Suggestion** column, click **Add DRS**.
3. Specify your range on the **Data Range Configuration** pane in the **Init Allocated** column.

Step 2: Determine Root Cause of Check

1. Search for the variable definition. See if you initialize the variable when you define it.

   Right-click the variable and select **Go To Definition**, if the option exists.
2. If you do not want to initialize the variable during definition, browse through all instances of the variable. Determine if you initialize the variable in any of those instances.

   Do one of the following:

   - On the **Source** pane, double-click the variable.

     Previous instances of the variable are highlighted. Scroll up to find them.
   - On the **Source** pane, right-click the variable. Select **Search For All References**.

     Select the previous instances on the **Search** pane.

   Possible Fix: If you do not initialize the variable, identify an instance where you can initialize it.
3. If you find an instance where you initialize the variable, determine if you perform the initialization in the scope where the **Non-initialized local variable** error appears.

For instance, you initialize the variable only in some branches of an `if ... elseif ... else` statement. If you use the variable outside the statement, the variable can be non-initialized.

Possible Fix:

- Perform the initialization in the same scope where you use it.

  In the preceding example, perform the initialization outside the `if ... elseif ... else` statement.
- Perform the initialization in a block with smaller scope but make sure that the block always executes.

  In the preceding example, perform the initialization in all branches of the `if ... elseif ... else` statement. Make sure that one branch of the statement always executes.

Step 3: Look for Common Causes of Check

1. See if you pass the variable to another function by reference or pointers before using it. Determine if you initialize the variable in the function body.

   To navigate to the function body, right-click the function and select **Go To Definition**, if the option exists.
2. Determine if you initialize the variable in code that is not reachable.

   For instance, you initialize the variable in code that follows a `break` or `return` statement.

   Possible Fix: Investigate the unreachable code. For more information, see “Review and Fix Unreachable Code Checks”.
3. Determine if you initialize the variable in code that can be bypassed during execution.

   For instance, you initialize the variable in a loop inside a function. However, for certain function arguments, the loop does not execute.

   Possible Fix:

   - Initialize the variable during declaration.

- Investigate when the code can be bypassed. Determine if you can avoid bypassing of the code.

4 If the variable is an array, determine if you initialize all elements of the array.

5 If the variable is a structured variable, determine if you initialize all fields of the structure.

If you do not initialize a certain field of the structure, see if the field is unused.

Possible Fix: Initialize a field of the structure if you use the field in your code.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see "Add Review Comments to Results".
- To add a justification in your code, see "Add Review Comments to Code".

For instance, you pass a variable to a function by pointer or reference. You intend to initialize the variable in the function body, but you do not provide the function body during verification. Then:

- Polyspace assumes that the function might not initialize the variable.
- If you use the variable following the function call, Polyspace considers that the variable can be non-initialized. It produces an orange **Non-initialized local variable** check on the variable.

For more information, see "Polyspace Software Assumptions".

Note: Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Non-initialized Pointer Checks

Follow one or more of these steps until you determine a fix for the **Non-initialized Pointer** check. There are multiple ways to fix this check. For a description of the check and code examples, see Non-initialized pointer.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

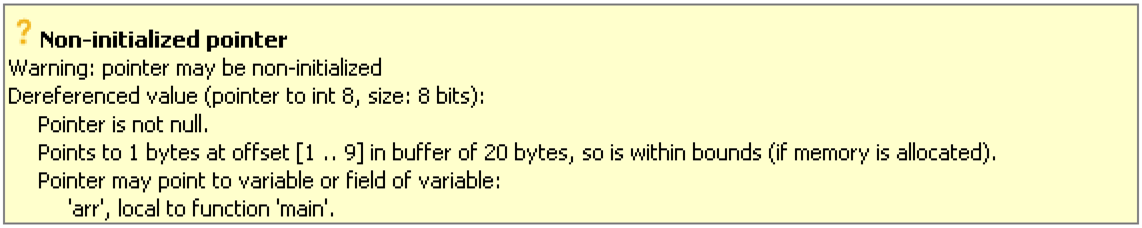
For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

Note: Unless you use the option Ignore default initialization of global variables, Polyspace considers global pointer variables to be initialized to `NULL` according to ANSI C standards.

Step 1: Interpret Check Information

Select the check on the **Results Summary** pane. On the **Check Details** pane, obtain further information about the check.



```
? Non-initialized pointer
Warning: pointer may be non-initialized
Dereferenced value (pointer to int 8, size: 8 bits):
    Pointer is not null.
    Points to 1 bytes at offset [1 .. 9] in buffer of 20 bytes, so is within bounds (if memory is allocated).
    Pointer may point to variable or field of variable:
        'arr', local to function 'main'.
```

Step 2: Determine Root Cause of Check

Right-click the pointer variable and select **Go To Definition**. Initialize the variable when you define it. If you do not want to initialize during definition, identify a suitable point to initialize the variable before you read it.

For orange checks, determine why the pointer is non-initialized on certain execution paths.

1. Find previous instances where write operations are performed on the pointer.
2. For each write operation, determine if the operation occurs:

   - Before the read operation containing the orange **Non-initialized pointer** check.

     Possible Fix: If the write operation occurs after the read operation, see if you intended to perform the operations in reverse order.
   - In an unreachable code block.

     Possible Fix: Investigate why the code block is unreachable. See “Review and Fix Unreachable Code Checks”.
   - In a code block that is not reached on certain execution paths. For example, the operation occurs in an `if` block in a function. The `if` block is not entered for certain function inputs.

     Possible Fix: Perform a write operation on all the execution paths. In the preceding example, perform the write operation in all branches of the `if ... elseif ... else` statement.

Depending on the nature of the variable, use the appropriate method to find previous operations on the variable.

Variable	How to Find Previous Operations on Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted.

Variable	How to Find Previous Operations on Variable
	2 On the **Search** pane, select the previous instances. • Browse the source code. 1 On the **Source** pane, double-click the variable. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. The current instance of the variable is shown on the **Variable Access** pane. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀. Read operations are indicated with ▶.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

Review and Fix Non-initialized Variable Checks

Follow one or more of these steps until you determine a fix for the **Non-initialized Variable** check. There are multiple ways to fix this check. For a description of the check and code examples, see Non-initialized variable.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

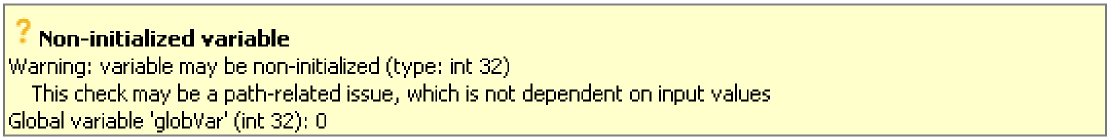
Note: Unless you use the option Ignore default initialization of global variables, Polyspace considers global variables to be initialized according to ANSI C standards. The default values are:

- 0 for `int`
- 0 for `char`
- 0.0 for `float`

In this section...

Step 1: Interpret Check Information

On the **Results Summary** pane, select the check. On the **Check Details** pane, obtain further information about the check.



? **Non-initialized variable**
Warning: variable may be non-initialized (type: int 32)
   This check may be a path-related issue, which is not dependent on input values
Global variable 'globVar' (int 32): 0

Obtain the following information:

- Probable cause of check, if described on the **Check Details** pane.

  In the preceding example, there is an orange **Non-initialized variable** check on the global variable `globVar`.

  The software detects that the check is potentially a path-related issue. Therefore, the global variable can be non-initialized only on certain execution paths. For example, you initialized the global variable in an `if` block, but did not initialize it in the corresponding `else` block.

  Possible Fix: Determine along which paths the global variables can be non-initialized.
- Value of global variable, if initialized.

  In the preceding example, when initialized, the global variable `globVar` has value 0.

Step 2: Determine Root Cause of Check

Right-click the variable and select **Go To Definition**. Initialize the variable when you define it. If you do not want to initialize during definition, identify a suitable point to initialize the variable before you read it.

If the check is orange, determine why the variable is non-initialized on certain execution paths.

1. Right-click the variable. Select **Show In Variable Access View**.
2. On the **Variable Access** pane, select each write operation on the variable.

   Write operations are indicated with ◂ and read operations with ▸.
3. Determine if the write operation occurs:

   - Before the read operation containing the orange **Non-initialized variable** check.

Possible Fix: If the write operation occurs after the read operation, see if you intended to perform the operations in reverse order.

- In an unreachable code block.

  Possible Fix: Investigate why the code block is unreachable. See “Review and Fix Unreachable Code Checks”.
- In a code block that is not reached on certain execution paths. For example, the operation occurs in an `if` block in a function. The `if` block is not entered for certain function inputs.

  Possible Fix: Perform a write operation on all the execution paths. In the preceding example, perform the write operation in all branches of the `if ... elseif ... else` statement.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

Review and Fix Non-Terminating Call Checks

Follow one or more of these steps until you determine a fix for the **Non-Terminating Call** check. There are multiple ways to fix the check. For a description of the check and code examples, see Non-terminating call.

For the general workflow on reviewing checks, see "Review Red Checks".

A red **Non-terminating call** check on a function call indicates one of the following:

- An operation in the function body failed for that particular call. Because there are other calls to the same function that do not cause a failure, the operation failure typically appears as an orange check in the function body.
- The function does not return to its calling context for other reasons. For example, a loop in the function body does not terminate.

In this section...

Step 1: Determine Root Cause of Check

Determine the root cause of the check in the function body.

1 Navigate to the function definition.

   Right-click the function call containing the red check. Select **Go To Definition**, if the option exists.

2 In the function body, determine if there is a loop where the termination condition is never satisfied.

   Possible fix: Change your code or the function arguments so that the termination condition is satisfied.

3 Otherwise, in the function body, identify which orange check caused the red **Non-terminating call** check on the function call.

   If you cannot find the orange check by inspection, rerun verification using the analysis option **Sensitivity context**. Provide the function name as option argument.

The software marks the orange check causing the red **Non-terminating call** check as a dark orange check.

For more information, see “Sensitivity context (C/C++)”.

For a tutorial on using the option, see “Identify Function Call Causing Orange Check”.

Possible fix: Investigate the cause of the orange check. Change your code or the function arguments to avoid the orange check.

Step 2: Look for Common Causes of Check

To trace a **Non-terminating call** check on a function call to an orange check in the function body, try the following:

- If the function call contains arguments, in the function body, search for all instances of the function parameters. See if you can find an orange check related to the parameters. Because other calls to the same function do not cause an operation failure, it is likely that the failure is related to the function parameter values in the red call.

  In the following example, in the body of `func`, search for all instances of `arg1` and `arg2`. Right-click the variable name and select **Search For All References**.

  ```
  void func(int arg1, double arg2) {
    .
    .
  }

  void main() {
    int valInt1,valInt2;
    double valDouble1, valDouble2;
    .
    .
    func(valInt1, valDouble1);
    func(valInt2, valDouble2);
  }
  ```

  Because `valInt1` and `valDouble1` do not cause an operation failure in `func`, the failure might be due to `valInt2` or `valDouble2`. Because `valInt2` and `valDouble2` are copied to `arg1` and `arg2`, the orange check must occur in an operation related to `arg1` or `arg2`.

- If the function call does not contain arguments, identify what is different between various calls to the function. See if you can relate the source of this difference to an orange check in the function body.

  For instance, if the function reads a global variable, different calls to the function can operate on different values of the global variable. Determine if the function body contains an orange check related to the global variable.

Review and Fix Non-Terminating Loop Checks

Follow one or more of these steps until you determine a fix for the **Non-Terminating Loop** check. There are multiple ways to fix the check. For a description of the check and code examples, see Non-terminating loop.

For the general workflow on reviewing checks, see “Review Red Checks”.

In this section...

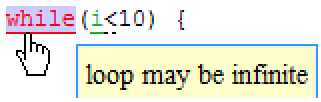
Interpret Check Information

Place your cursor on the loop keyword such as `for` or `while`.

Obtain the following information from the tooltip:

- Whether the loop is infinite or contains a run-time error.

  In the following example, it is likely that the loop is infinite.

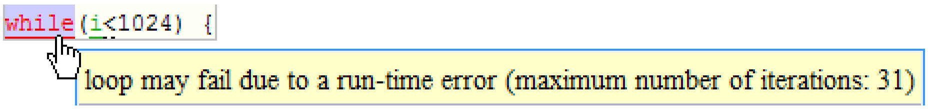


```
while(i<10) {
```

loop may be infinite

- If the loop contains a run-time error, the number of loop iterations. Because Polyspace considers that execution stops when a run-time error occurs, from this number, you can determine which loop iteration contains the error.

  In the following example, it is likely that the loop contains a run-time error. The error is likely to occur on the 31st loop iteration.



```
while(i<1024) {
```

loop may fail due to a run-time error (maximum number of iterations: 31)

Determine Root Cause of Check

- If the loop is infinite, determine why the loop-termination condition is never satisfied.

  If you deliberately have an infinite loop in your code, such as for cyclic applications, you can add a comment and justification in your result or code.

  - To add a justification in your result, see “Add Review Comments to Results”.
  - To add a justification in your code, see “Add Review Comments to Code”.
- If the loop contains a run-time error, identify the error that causes the **Non-terminating loop** check. Fix the error.

  In the loop body, the run-time error typically occurs as an orange check of another type on an operation. The check is orange and not red because the operation typically passes the check in the first few loop iterations and fails only in a later iteration. However, because the failure occurs every time the loop runs, the **Non-terminating loop** check on the loop keyword is red.

Look for Common Causes of Check

- If the loop is infinite:

  - Check your loop-termination condition.
  - Inside the loop body, see if you change at least one of the variables involved in the loop-termination condition.

    For instance, if the loop-termination condition is `while (count1 + count2 < count3)`, see if you are changing at least one of `count1`, `count1`, or `count3` in the loop.
  - If you are changing the variables involved in the loop-termination condition, see if you are changing them in the right direction.

    For instance, if the loop termination condition is `while(i<10)` and you decrement `i` in the loop, the loop does not terminate. You must increment `i`.
- If the loop contains a run-time error:

  - If the loop control variable is an array index, see if you have an orange **Out of bounds array index** error in the loop body.

- If the loop control variable is passed to a function, see if you can relate the red **Non-terminating loop** error to an orange error in the function body.

Review and Fix Out of Bounds Array Index Checks

Follow one or more of these steps until you determine a fix for the **Out of Bounds Array Index** check. There are multiple ways to fix the check. For a description of the check and code examples, see Out of bounds array index.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

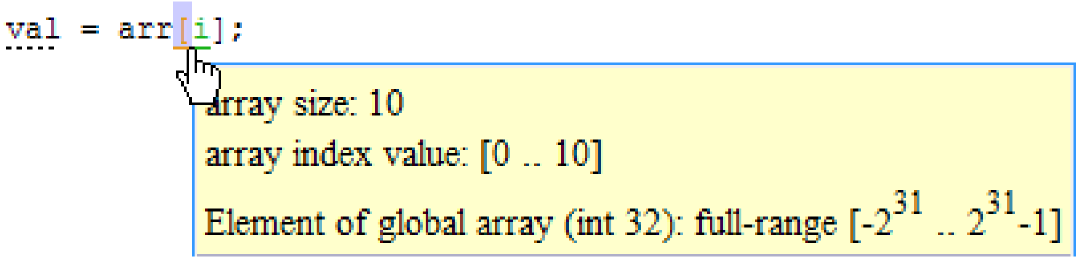
For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

In this section...
"Step 1: Interpret Check Information" on page 10-51
"Step 2: Determine Root Cause of Check" on page 10-52
"Step 3: Look for Common Causes of Check" on page 10-54
"Step 4: Trace Check to Polyspace Assumption" on page 10-54

Step 1: Interpret Check Information

Place your cursor on the `[` symbol.



```
val = arr[i];
```

array size: 10

array index value: [0 .. 10]

Element of global array (int 32): full-range $[-2^{31} .. 2^{31}-1]$

Obtain the following information from the tooltip:

- Array size. The allowed range for array index is 0 to (array size - 1).
- Actual range for array index

In the preceding example, the array size is 10. Therefore, the allowed range for the array index is 0 to 9. However, the actual range is 0 to 10.

Possible fix: To avoid the out of bounds array index, access the array only if the index is between 0 and (array size - 1).

```
#define SIZE 100
int arr[SIZE];
.
.
if(i<SIZE)
 val = arr[i]
else
 /*Error handling */
```

Step 2: Determine Root Cause of Check

If you want to know or change the array size, right-click the array variable and select **Go To Definition**, if the option exists. Otherwise, trace the data flow starting from the array index variable. Identify a point where you can constrain the index variable.

To trace the data flow, select the check, and note the information on the **Check Details** pane.

- If the **Check Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Check Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find previous instances of the array index variable.
  2. Browse through those instances. Find the instance where you constrain the array index variable to (array size - 1).

     Possible fix: If you do not find an instance where you constrain the index variable, specify a constraint for the variable. For example:

     ```
     if(index<SIZE)
      read(array[index]);
     ```
  3. Determine if the constraint applies to the instance where the **Out of bounds array index** error occurs.

For example, you can constrain the index variable in a `for` loop using `for(index=0; index<SIZE; index++)`. However, you access the array outside the loop where the constraint does not apply.

Possible fix: Investigate why the constraint does not apply. See if you have to constrain the index variable again.

4. If the index variable is obtained from another variable, trace the data flow for the second variable. Determine if you have constrained that second variable to (array size - 1).

Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.

Variable	How to Find Previous Instances of Variable

Function argument
`void func(..,int arg,..`
`.`
`.`
`}`
	1 On the **Check Details** pane, select the fx button.
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	2 Select a calling function name. You go to the call to `func` in your source.
	3 Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

Step 3: Look for Common Causes of Check

- See if you are starting the array index variable from 0.
- In the condition that constrains your array index variable, see if you use <= when you intended to use <.
- If a check occurs in or immediately after a `for` or `while` loop, determine if you have to reduce the number of runs of the loop.
- If you use the `sizeof` function to constrain your array, see if you are dividing `sizeof(array)` by `sizeof(array[0])` to obtain the array size.

  `sizeof(array)` returns the array size in bytes.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.

- To add a justification in your code, see “Add Review Comments to Code”.

For instance, you use constrain the array index using a function whose definition you do not provide. Then:

1. Polyspace cannot determine that the array index variable is constrained.
2. When you use this variable as array index, an **Out of bounds array index** error can occur.
3. If you know that the variable is constrained to the array size, add a comment and justification describing why you did not change your code.

For more information, see “Polyspace Software Assumptions”.

Note: Before justifying an orange check, consider carefully whether you can improve your coding design.

For instance, constraining a global variable in one function and using it as array index in a second function causes vulnerabilities in your code. If a new function is called between the previous two functions and modifies your global variable, the constraint no longer applies.

Review and Fix Overflow Checks

Follow one or more of these steps until you determine a fix for the **Overflow** check. There are multiple ways to fix the check. For a description of the check and code examples, see Overflow.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

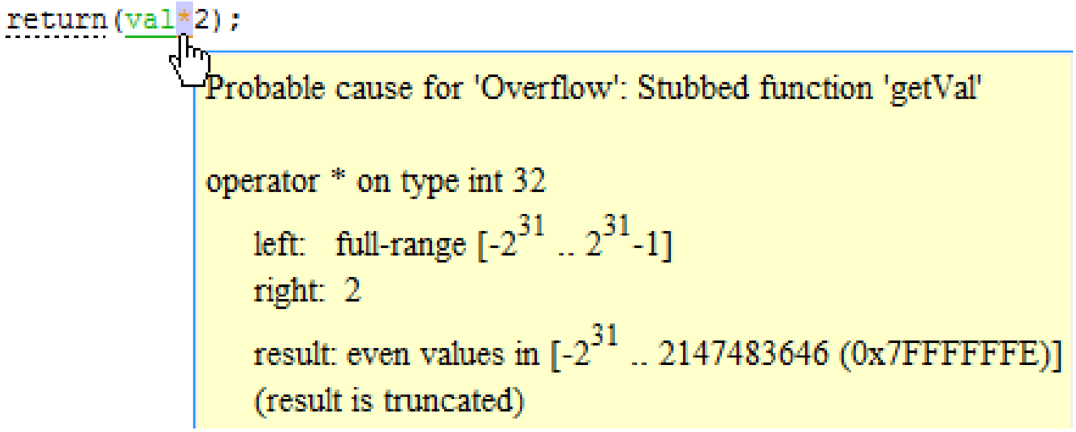
For the general workflow that applies to all checks, see:

- “Review Red Checks”
- “Review Orange Checks”

In this section...

Step 1: Interpret Check Information

Place your cursor on the operation that overflows.



```
return(val*2);
```

Probable cause for 'Overflow': Stubbed function 'getVal'

operator * on type int 32

left: full-range [-2^{31} .. 2^{31}-1]
right: 2

result: even values in [-2^{31} .. 2147483646 (0x7FFFFFFE)]
(result is truncated)

Obtain the following information from the tooltip:

- The operand variable you can constrain to avoid the overflow.

  In the preceding example, the left operand, `val`, has full range of values.

  Possible fix:: To avoid the overflow, perform the multiplication only if `val` is in a certain range.

  ```
  if(val < INT_MAX/2)
      return(val*2);
  else
      /* Alternate action */
  ```

- The probable root cause for overflow, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

  Possible fix: To avoid the overflow, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. To specify a range:

  1. Select the ? icon on the **Check Details** pane.
  2. On the **Orange Sources** pane, find the stubbed function `getVal`. On the **Suggestion** column, click **Add DRS**.
  3. Specify your range on the **Data Range Configuration** pane in the **Init Range** column.

Step 2: Determine Root Cause of Check

Trace the data flow starting from the operand variable that you want to constrain. Identify a suitable point to specify your constraint.

In the following example, trace the data flow starting from `tempVal`.

```
val = func();
.
.
tempVal = val;
.
.
```

```
tempVal++ ;
```

In this example, you might find that:

1. `tempVal` obtains a full-range of values from `val`.

   Possible fix: Assign `val` to `tempVal` only if `val` is less than a certain value.
2. `val` obtains a full-range of values from `func`.

   Possible fix: Constrain the return value of `func`, either in the body of `func` or through the Polyspace Data Range Specification interface, if `func` is stubbed.

   For more information on Data Range Specification, see “Specify Constraints”.

To trace the data flow, select the check and note the information on the **Check Details** pane.

- If the **Check Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Check Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the operand variable.

     Example: The previous write operation on `tempVal` is `tempVal=val`.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

     Example: At `tempVal=val`, you find that `val` has a full-range of values. Therefore, you trace `val`.
  3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

     Example: The previous write operation on `val` is `val=func()`. You can choose to specify your constraint on the return value of `func`.

Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. **1** Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. **2** On the **Search** pane, select the previous instances. • Browse the source code. **1** Double-click the variable on the **Source** pane. All instances of the variable are highlighted. **2** Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	**1** Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. **2** On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.

Function argument
`void func(..,int arg,..`
`.`
`.`
`}`
	1 On the **Check Details** pane, select the fx button.
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	2 Select a calling function name. You go to the call to `func` in your source.
	3 Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

Step 3: Look for Common Causes of Check

If the operation causing the overflow occurs in a loop or in the body of a recursive function:

1. See if you can reduce the number of loop runs or recursions.
2. See if you can place an exit condition in the loop or recursive function before the operation overflows.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see "Add Review Comments to Results".
- To add a justification in your code, see "Add Review Comments to Code".

For instance, you are using a volatile variable in your code. Then:

1. Polyspace assumes that the volatile variable is full-range at every step in the code.
2. An operation using that variable can overflow and is therefore orange.
3. If you know that the variable takes a smaller range of values, add a comment and justification in your code describing why you did not change your code.

For more information, see "Polyspace Software Assumptions".

Note: Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Shift Operations Checks

Follow one or more of these steps until you determine a fix for the **Shift Operations** check. There are multiple ways to fix the check. For a description of the check and code examples, see Shift operations.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- "Review Red Checks"
- "Review Orange Checks"

In this section...

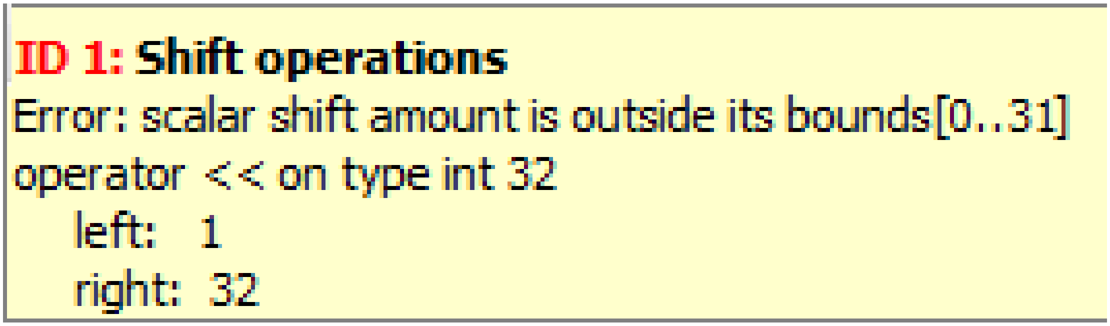
Step 1: Interpret Check Information

Select the red or orange **Shift operations** check. Obtain the following information from the **Check Details** pane:

- The reason for the check being red or orange. Possible reasons:
  - The shift amount can be outside allowed bounds.

    The software also states the allowed range for the shift amount.
  - Left operand of left shift can be negative.

  In the example below, a red error occurs because the shift amount is outside allowed bounds. The allowed range for the shift amount is 0 to 31.



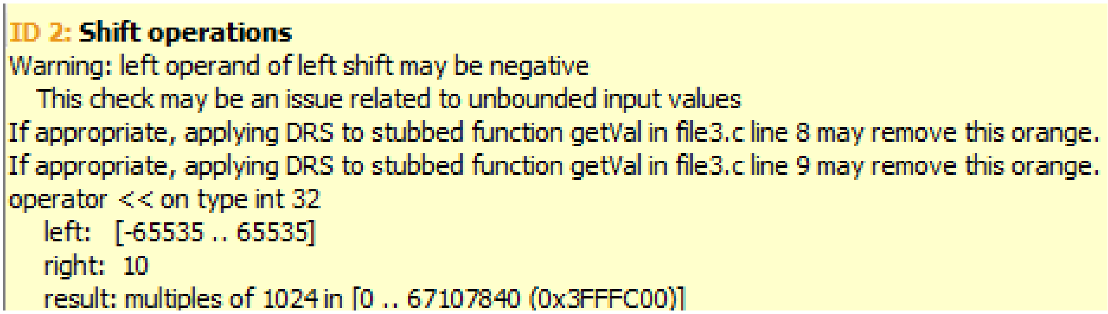
ID 1: **Shift operations**
Error: scalar shift amount is outside its bounds[0..31]
operator << on type int 32
left: 1
right: 32

Possible fix: To avoid the red or orange check, perform the shift operation only if the shift amount is within bounds.

```
if(shiftAmount < (sizeof(int) * 8))
  /* Perform the shift */
else
  /* Error handling */
```

- Probable root cause for the check, if the software provides this information.

  In the example below, the software identifies a stubbed function, `getVal` as probable cause.



ID 2: **Shift operations**
Warning: left operand of left shift may be negative
This check may be an issue related to unbounded input values
If appropriate, applying DRS to stubbed function getVal in file3.c line 8 may remove this orange.
If appropriate, applying DRS to stubbed function getVal in file3.c line 9 may remove this orange.
operator << on type int 32
left: [-65535 .. 65535]
right: 10
result: multiples of 1024 in [0 .. 67107840 (0x3FFFC00)]

Possible fix: To avoid the orange check, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `0..10`. To specify a range:

1 Select the ? icon on the **Check Details** pane.

2. On the **Orange Sources** pane, find the stubbed function `getVal`. On the **Suggestion** column, click **Add DRS**.
3. Specify your range on the **Data Range Configuration** pane in the **Init Range** column.

Step 2: Determine Root Cause of Check

- If the shift amount is outside bounds, trace the data flow for the shift variable. Identify a suitable point where you can constrain the shift variable.

  In the following example, trace the data flow for `shiftAmount`.

  ```
  void func(int val) {
   int shiftAmount = getShiftAmount();
   int res = val >> shiftAmount;
  }
  ```
  You might find that `getShiftAmount` returns full-range of values.

  Possible fix:

  - Perform the shift operation only if `shiftAmount` is between 0 and `(sizeof(int))*8 - 1`.
  - Constrain the return value of `getShiftAmount`, in the body of `getShiftAmount` or through the Polyspace Data Range Specification interface, if you do not have the definition of `getShiftAmount`.

    For more information on Data Range Specification, see "Specify Constraints".
- If the left operand of a left shift operation can be negative, trace the data flow for the left operand variable. Identify a suitable point where you can constrain the left operand variable.

  In the following example, trace the data flow for `shiftAmount`.

  ```
  void func(int shiftAmount) {
   int val = getVal();
   int res = val << shiftAmount;
  }
  ```
  You might find that `getVal` returns full-range of values.

  Possible fix:

- Perform the shift operation only if `val` is positive.
- Constrain the return value of `getVal`, in the body of `getVal` or through the Polyspace Data Range Specification interface, if you do not have the definition of `getVal`.

  For more information on Data Range Specification, see "Specify Constraints".
- If you want Polyspace to allow the operation, use the analysis option **Allow negative operand for left shifts**. See "Allow negative operand for left shifts (C/C ++)".

To trace the data flow, select the check and note the information on the **Check Details** pane.

- If the **Check Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Check Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the variable you want to trace.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.
  3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

  Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**.

Variable	How to Find Previous Instances of Variable
	All instances of the variable appear on the **Search** pane with the current instance highlighted. **2** On the **Search** pane, select the previous instances. • Browse the source code. **1** Double-click the variable on the **Source** pane. All instances of the variable are highlighted. **2** Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	**1** Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. **2** On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.

Function argument
`void func(..,int arg,..`
`.`
`.`
`}`
	1 On the **Check Details** pane, select the *fx* button.
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	2 Select a calling function name. You go to the call to `func` in your source.
	3 Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

Step 3: Look for Common Causes of Check

1. See if you have specified the right target processor type. The target processor type determines the number of bits allowed for a certain variable type.

   To determine the number of bits allowed:

   a. Navigate to the variable definition. Note the variable type.

      Right-click the variable and select **Go To Definition**, if the option exists.
   b. See the number of bits allowed for the type.

      In the configuration used for your results, select the **Target & Compiler** node. Click the **Edit** button next to the **Target processor type** list.
2. For left shifts with a negative operand, see if you intended to perform a right shift instead.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

For instance, you obtain a variable from an undefined function and perform a left shift on it. Then:

1. Polyspace assumes that the function can return a negative value.
2. The left shift operation can occur on a negative value and therefore there is an orange check on the operation.
3. If you know that the function returns a positive value, add a comment and justification describing why you did not change your code.

For more information, see “Polyspace Software Assumptions”.

Review and Fix Unreachable Code Checks

Follow one or more of these steps until you determine a fix for the **Unreachable Code** check. There are multiple ways to fix this check. For a description of the check and code examples, see Unreachable code.

If you determine that the check represents defensive code, add a comment and justification in your result or code explaining why you did not change your code. To:

- Add justification in your result, see “Add Review Comments to Results”.
- Add justification in your code, see “Comment Code for Known Defects”.

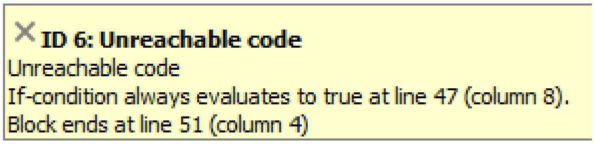
In this section...
“Step 1: Interpret Check Information” on page 10-67
“Step 2: Determine Root Cause of Check” on page 10-68
“Step 3: Look for Common Causes of Check” on page 10-70

Step 1: Interpret Check Information

1 Select the check on the **Results Summary** or **Source** pane.

2 View the message on the **Check Details** pane.

The message explains why the block of code is unreachable.



ID 6: Unreachable code
Unreachable code
If-condition always evaluates to true at line 47 (column 8).
Block ends at line 51 (column 4)

3 A code block is usually unreachable when the condition that determines entry into the block is not satisfied. See why the condition is not satisfied.

a On the **Source** pane, place your cursor on the variables involved in the condition to determine their values.

b Using these values, see why the condition is not satisfied.

Note: Sometimes, a condition itself is redundant. For example, it is always true or coupled:

- Through an || operator to another condition that is always true.
- Through an && operator to another condition that is always false.

For example, in the following code, the condition x%2==0 is redundant because the first condition x>0 is always true.

```
assert(x>0);
if(x>0 || x%2 == 0)
```

If a condition is redundant, instead of a block of code, the condition itself is marked gray.

Step 2: Determine Root Cause of Check

Trace the data flow for each variable involved in the condition.

In the following example, trace the data flow for arg.

```
void foo(void) {
    int x=0;
    .
    .
    bar(x);
    .
    .
}

void bar(int arg) {
    if(arg==0) {
        /*Block 1*/
    }
    else {
        /*Block 2*/
    }
}
```

You might find that bar is called only from foo. Since the only argument of bar has value 0, the else branch of if(arg==0) is unreachable.

Possible fix: If you do not intend to call bar elsewhere and know that the values passed to bar will not change, you can remove the if-else statement in bar and retain only the content of Block 1.

To trace the data flow, select the check and note the information on the **Check Details** pane.

- If the **Check Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Check Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise, for each variable involved in the condition, find previous instances and trace back to the root cause of check. For more information on common root causes, see "Step 3: Look for Common Causes of Check" on page 10-70.

  Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. **1** Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. **2** On the **Search** pane, select the previous instances. • Browse the source code. **1** Double-click the variable on the **Source** pane. All instances of the variable are highlighted. **2** Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	**1** Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. **2** On this pane, select the previous instances of the variable.

Variable	How to Find Previous Instances of Variable
	Write operations on the variable are indicated with ◀ and read operations with ▶.

Function argument
 	
`void func(..,int arg,.`
`.`
`.`
`}`
	1 On the **Check Details** pane, select the fx button.
	
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	2 Select a calling function name. You go to the call to `func` in your source.
	3 Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

Step 3: Look for Common Causes of Check

1. Look for the following in your `if` tests:

   - You are testing the variables that you intend to test.

     For example, you might have a local variable that shadows a global variable. You might be testing the local variable when you intend to test the global one.

   - You are using parentheses to impose the sequence in which you want operations in the `if` test to execute.

     For example, `if((!a && b) || c)` imposes a different sequence of operations from `if(!(a && b) || c)`. Unless you use parentheses, the operations obey operator precedence rules. The rules can cause the operations to execute in a sequence that you did not intend.

   - You are using = and == operators in the right places.

Possible fix: Correct errors if any.

- Use Polyspace Bug Finder to check for common defects such as Invalid use of = (assignment) operator and Variable shadowing.
- To avoid errors due to incorrect operation sequence, check for violations of MISRA C:2012 Rule 12.1.

2. See if you are performing a test that you have performed previously.

   The redundant test typically occurs on the argument of a function. The same test is performed both in the calling and called function.

   ```
   void foo(void) {
       if(x>0)
           bar(x);
       .
       .
   }

   void bar(int arg) {
       if(arg==0) {
       }
   }
   ```

   Possible fix: If you intend to call `bar` later, for example, in yet unwritten code, or reuse `bar` in other programs, retain the test in `bar`. Otherwise, remove the test.

3. See if your code is unreachable because it follows a `break` or `return` statement.

   Possible fix: See if you placed the `break` or `return` statement at an unintended place.

4. See if the section of unreachable code exists because you are following a coding standard. If so, retain the section.

   For example, the `default` block of a `switch-case` statement is present to capture abnormal values of the `switch` variable. If such values do not occur, the block is unreachable. However, you might violate a coding standard if you remove the block.

5. See if the unreachable code is related to an orange check earlier in the code. Following an orange check, Polyspace normally terminates execution paths that contain an error. Because of this termination, code following an orange check can appear gray.

For example, Polyspace places an orange check on the dereference of a pointer `ptr` if you have not vetted `ptr` for `NULL`. However, following the dereference, it considers that `ptr` is not `NULL`. If a test `if(ptr==NULL)` follows the dereference of `ptr`, Polyspace marks the corresponding code block unreachable.

For more examples, see:

- "Gray Check Following Orange Check"

  An exception to this behavior is overflow. If you specify the appropriate **Overflow computation mode**, Polyspace wraps the result of an overflow and does not terminate the execution paths. See "Overflow computation mode (C/C++)".
- "Left operand of left shift may be negative"

Possible fix: Investigate the orange check. In the above example, investigate why the test `if(ptr==NULL)` occurs after the dereference and not before.

Review and Fix User Assertion Checks

Follow one or more of these steps until you determine a fix for the **User Assertion** check. There are multiple ways to fix this check. For a description of the check and code examples, see User assertion.

Sometimes, especially for an orange check, you might determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- “Review Red Checks”
- “Review Orange Checks”

How to use this check: Typically you use `assert` statements during debugging to check if a condition is satisfied at the current point in your code. For instance, if you expect a variable `var` to have values in a range `[1,10]` at a certain point in your code, you place the following statement at that point:

```
assert(var >=1 && var <= 10);
```

Polyspace statically determines whether the condition is satisfied.

Therefore, you can judiciously insert `assert` statements that test for requirements from your code.

- A red result for the **User assertion** check indicates that the requirement is definitely not met.
- An orange result for the **User assertion** check indicates that the requirement is possibly not met.

In this section...

Step 1: Determine Root Cause of Check

Trace the data flow for each variable involved in the `assert` statement.

In the following example, trace the data flow for myArray.

```
int* getArray(int numberOfElements) {
    .
    .
    arrayPtr = (int*) malloc(numberOfElements);
    .
    .
    return arrayPtr;
}
void func() {
    int* myArray = getArray(10);
    assert(myArray!=NULL);
    .
    .
}
```

In this example, it is possible that in getArray, the return value of malloc is not checked for NULL.

Possible fix: If you expect a certain requirement, see if you have tests that enforce the requirement. In this example, if you expect getArray to return a non-NULL value, in getArray, test the return value of malloc for NULL.

To trace the data flow, select the check and note the information on the **Check Details** pane.

- If the **Check Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Check Details** pane shows the line number of probable cause for the check, right-click in the **Source** pane. Select **Go To Line**. Enter the line number.
- Otherwise, for each variable involved in the condition, find previous instances and trace back to the root cause of the check. For more information on common root causes, see "Step 3: Look for Common Causes of Check" on page 10-70.

  Depending on the variable, use the following navigation shortcuts to find previous instances.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable.

Variable	How to Find Previous Instances of Variable
	1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.

Function argument
`void func(..,int arg,..`
`.`
`.`
`}`
	1 On the **Check Details** pane, select the *fx* button.
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	2 Select a calling function name. You go to the call to `func` in your source.
	3 Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

Step 2: Look for Common Causes of Check

1. If the check is orange and occurs in a function, see if the function is called multiple times. Determine if the assertion fails only on certain calls. For those calls, navigate to the caller body and see if you have tests that enforce your assertion requirement.

   - To see the callers of a function, select the function name on the **Source** pane. All callers appear on the **Call Hierarchy** pane. Select a caller name to navigate to it in your source.
   - To determine if a subset of calls cause the orange check, use the option "Sensitivity context (C/C++)". For a tutorial, see "Identify Function Call Causing Orange Check".

2. If you have tests that enforce the assertion requirement, see if:

   - The assertion statement is within the scope of the tests.
   - You modify the test variables between the test and the assertion.

   For instance, the test `if(index < SIZE)` enforces that `index` is less than `SIZE`. However, the assertion `assert(index < SIZE)` can fail if:

   - It occurs outside the `if` block.
   - Before the assertion, you modify `index` in the `if` block.

   Possible fix: Consider carefully whether you must meet your assertion requirements. If you must meet those requirements, place tests that enforce your requirement. After the tests, avoid modifying the test variables.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code.

- To add a justification in your result, see “Add Review Comments to Results”.
- To add a justification in your code, see “Add Review Comments to Code”.

For instance, after a function call, you assert a relation between two variables. Then:

1. Depending on the depth of the function call and the complexity of your code, Polyspace can sometimes approximate the variable ranges. Because of the approximation, the software cannot establish if the relation holds and produces an orange **User assertion** check.
2. Run dynamic tests on the orange check to determine if the assertion fails.

   For a tutorial, see “Test Orange Checks for Run-Time Errors”.
3. Try to reduce your code complexity and rerun the verification. Otherwise, add a comment and a justification in your result or code describing why you did not change your code.

For more information, see “Polyspace Software Assumptions”.

Note: Before justifying an orange check, consider carefully whether you can improve your coding design.

Managing Orange Checks

Sources of Orange Checks

In this section...

When Orange Checks Occur

An orange check indicates that Polyspace detects a possible run-time error but cannot prove it. A check on an operation appears orange if both conditions are true:

First condition	Second condition	Example
The operation occurs multiple times on an execution path or on multiple execution paths.	During static verification, the operation fails only a fraction of times or only on a fraction of paths.	The operation occurs in: • A loop with more than one iterations. • A function that is called more than once.
The operation involves a variable that can take multiple values.	During static verification, the operation fails only for a fraction of values.	The operation involves a `volatile` variable.

During static verification, Polyspace can consider more execution paths than the execution paths that occur during run time. If an operation fails on a subset of paths, Polyspace cannot determine if that subset actually occurs during run time. Therefore, instead of a red check, it produces an orange check on the operation.

Why Review Orange Checks

Considering a superset of actual execution paths is a sound approximation because Polyspace does not lose information. If an operation contains a run-time error, Polyspace does not produce a green check on the operation. If Polyspace cannot prove the run-time error because of approximations, it produces an orange check. Therefore, you must review orange checks.

Examples of Polyspace approximations include:

- Approximating the range of a variable at a certain point in the execution path. For instance, Polyspace can approximate the range `{-1} U [0,10]` of a `float` variable by `[-1,10]`.
- Approximating the interleaving of instructions in multitasking code. For instance, even if certain instructions in a pair of tasks cannot interrupt each other, Polyspace verification might not take that into account.

How to Review Orange Checks

To ensure that an operation does not fail during run time:

1. Determine if the execution paths on which the operation fails can actually occur.

   For more information, see “Review Orange Checks”.
2. If any of the execution paths can occur, fix the cause of the failure.
3. If the execution paths cannot occur, enter a comment in your Polyspace result or source code, describing why they cannot occur. For more information on:

   - Entering comments in results, see “Add Review Comments to Results”.
   - Entering comments in code, see “Add Review Comments to Code”.

   In a later verification, you can import these comments into your results. Then, if the orange check persists in the later verification, you do not have to review it again.

How to Reduce Orange Checks

Polyspace performs approximations because of one of the following.

- Your code does not contain complete information about run-time execution. For example, your code is partially developed or contains variables whose values are known only at run time.

  If you want fewer orange checks, provide the information that Polyspace requires. For more information, see “Provide Context for Verification”.
- Your code is very complex. For example, there can be multiple type conversions or multiple `goto` statements.

  If you want fewer orange checks, reduce the complexity of your code and follow recommended coding practices. For more information, see “Follow Coding Rules”.

- Polyspace must complete the verification in reasonable time and use reasonable computing resources.

  If you want fewer orange checks, improve the verification precision. But higher precision also increases verification time. For more information, see “Improve Verification Precision”.

Managing Orange Checks

Polyspace checks every operation in your code for certain run-time errors. Therefore, you can have several orange checks in your verification results. To avoid spending unreasonable time on an orange check review, you must develop an efficient review process.

Depending on your stage of software development and quality goals, you can choose to:

- Review all red checks and critical orange checks.
- Review all red checks and all orange checks.

To see only red and critical orange checks, on the **Results Summary** pane, select **Show** > **Critical checks**.

In this section...
"Software Development Stage" on page 11-5
"Quality Goals" on page 11-7

Software Development Stage

Development Stage	Situation	Review Process
Initial stage or unit development stage	In initial stages of development, you can have partially developed code or want to verify each source file independently. In that case, it is possible that: • You have not defined all your functions and class methods. • You do not have a `main` function Because of insufficient information in the code, Polyspace makes assumptions that result	In the initial stages of development, review all red checks. For orange checks, depending on your requirements, do one of the following: • You want your partially developed code to be free of errors independent of the remaining code. For instance, you want your functions to not cause run-time errors for any input.

Development Stage	Situation	Review Process
	in many orange checks. For instance, if you use the default configuration, Polyspace assumes full range for inputs of functions that are not called in the code.	If so, review orange checks at this stage. • You might want your partially developed code to be free of errors only in the context of the remaining code. If so, do one of the following: • Ignore orange checks at this stage. • Provide the context and then review orange checks. For instance, you can provide stubs for undefined functions to emulate them more accurately. For more information, see "Provide Context for Verification".
Later stage or integration stage	In later stages of development, you have provided all your source files. However, it is possible that your code does not contain all information required for verification. For example, you have variables whose values are known only at run time.	Depending on the time you want to spend, do one of the following: • Review red checks only. • Review red and critical orange checks.

Development Stage	Situation	Review Process
Final stage	• You have provided all your source files. • You have emulated run-time environment accurately through the verification options.	Depending on the time you want to spend, do one of the following: • Review red checks and critical orange checks. • Review red checks and all orange checks. For each orange check: • If the check indicates a run-time error, fix the cause of the error. • If the check indicates a Polyspace approximation, enter a comment in your results or source code. As part of your final release process, you can have one of these criteria: • All red and critical orange checks must be reviewed and justified. • All red and orange checks must be reviewed and justified. To justify a check, assign the **Status** of **No action planned** or **Justify with annotations** to the check.

Quality Goals

For critical applications, you must review all red and orange checks.

- If an orange check indicates a run-time error, fix the cause of the error.
- If an orange check indicates a Polyspace approximation, enter a comment in your results or source code.

As part of your final release process, review and justify all red and orange checks. To justify a check, assign the **Status** of **No action planned** or **Justify with annotations** to the check.

For noncritical applications, you can choose whether or not to review the noncritical orange checks.

Related Examples

- "Prioritize Check Review"

More About

- "Sources of Orange Checks"

Limit Display of Orange Checks

This example shows how to control the number and type of orange checks displayed on the **Results Summary** pane. To reduce your review effort, you can do one of the following:

- Display only the critical orange checks.

  Use the option **Show** > **Critical checks** on the **Results Summary** pane. For more information, see “Critical Orange Checks”.
- Limit the number of orange checks to display, or suppress orange checks for check types that you do not want to review.

  Use predefined limits or assign your own limits through **Tools** > **Preferences**. If you assign your own limits, you can share the limits file to help developers in your organization review at least a certain number or percentage of orange checks.

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

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

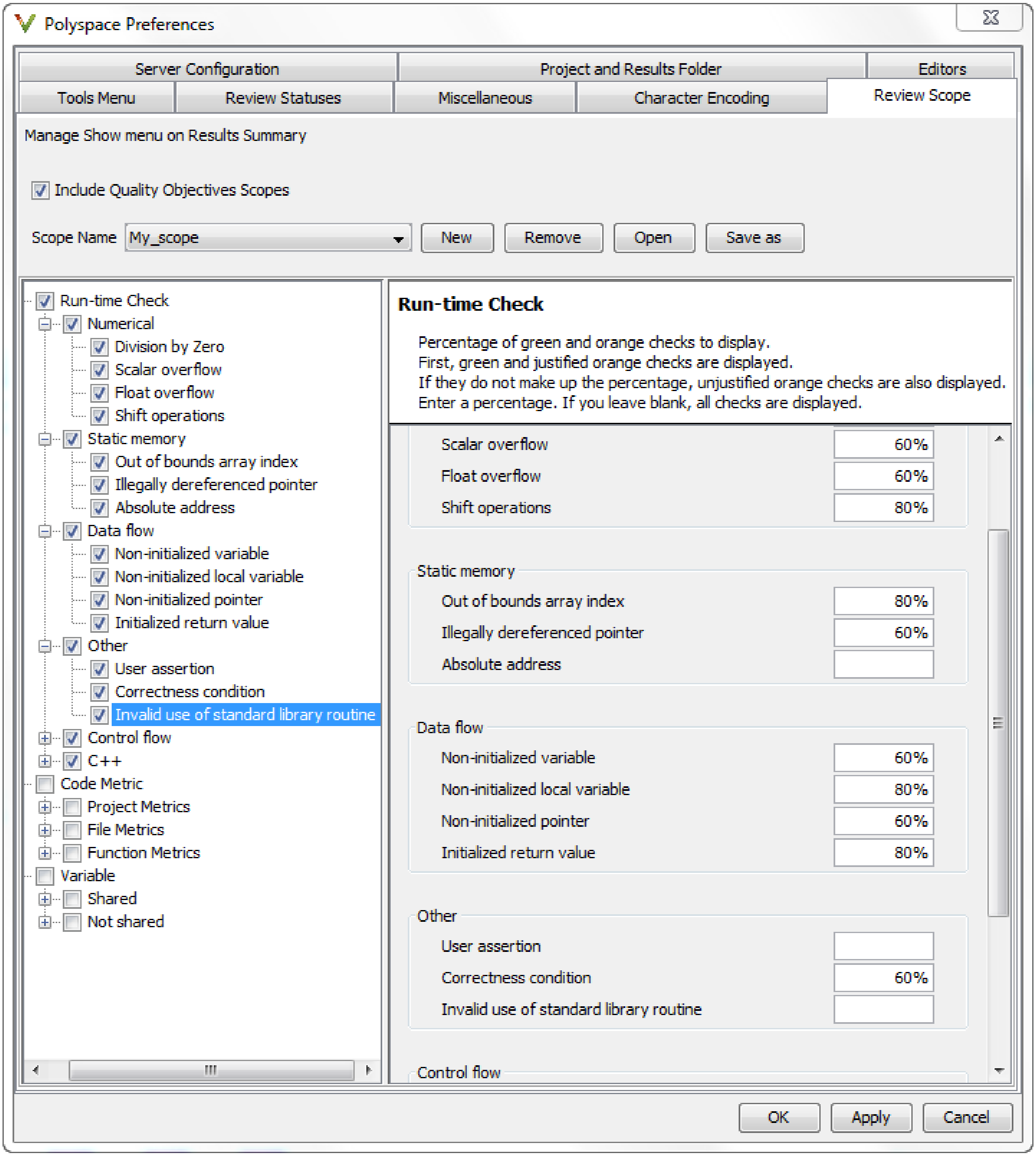
     The **Scope Name** list shows additional options, `HIS`, `SQO-4`, `SQO-5` and `SQO-6`. Select an option to see the limit values.

     In addition to orange checks, the options impose limits on the display of code metrics. The limits on code metrics are the same for all options. The option `HIS` displays code metrics only. For more information, see “Review Code Metrics”.

     For a detailed explanation of the predefined levels, see “Predefined Software Quality Objectives”.
   - To define your own limits, select **New**. Save your limits file.

     If you want a check to be suppressed from **Results Summary**, on the left pane, under **Run-time Check**, clear the box for the check. Otherwise, on the right pane, specify a percentage of orange checks to display. The default is 100%.

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



Manage Show menu on Results Summary

☑ Include Quality Objectives Scopes

Scope Name: My_scope

- ☑ Run-time Check
  - ☑ Numerical
    - ☑ Division by Zero
    - ☑ Scalar overflow
    - ☑ Float overflow
    - ☑ Shift operations
  - ☑ Static memory
    - ☑ Out of bounds array index
    - ☑ Illegally dereferenced pointer
    - ☑ Absolute address
  - ☑ Data flow
    - ☑ Non-initialized variable
    - ☑ Non-initialized local variable
    - ☑ Non-initialized pointer
    - ☑ Initialized return value
  - ☑ Other
    - ☑ User assertion
    - ☑ Correctness condition
    - ☑ Invalid use of standard library routine
  - ☑ Control flow
  - ☑ C++
- □ Code Metric
  - □ Project Metrics
  - □ File Metrics
  - □ Function Metrics
- □ Variable
  - □ Shared
  - □ Not shared

Run-time Check

Percentage of green and orange checks to display.
First, green and justified orange checks are displayed.
If they do not make up the percentage, unjustified orange checks are also displayed.
Enter a percentage. If you leave blank, all checks are displayed.

Scalar overflow	60%
Float overflow	60%
Shift operations	80%
Static memory	
Out of bounds array index	80%
Illegally dereferenced pointer	60%
Absolute address	
Data flow	
Non-initialized variable	60%
Non-initialized local variable	80%
Non-initialized pointer	60%
Initialized return value	80%
Other	
User assertion	
Correctness condition	60%
Invalid use of standard library routine	
Control flow	

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

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

   - If you use predefined limits, the options `HIS`, `SQO-4`, `SQO-5` and `SQO-6` appear.
   - If you define your own limits, the option corresponding to your limits file name appears.

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

   - If you specify an absolute number, Polyspace displays that number of orange checks.
   - If you specify a percentage, Polyspace calculates that percentage of the total green and orange checks. The software then considers whether green checks alone make up the percentage. If they do not make up the percentage, the software then displays sufficient orange checks to make up the percentage. For example, if you specify 60, the software checks if 60% of your green and orange checks comprise of green checks only. Otherwise, it displays sufficient orange checks to make up the 60%.

     You can use a review scope with percentage specifications to ensure that at least 60% of (green + orange) checks are either green or justified orange. To justify a check, you must assign a **Status** of either `No action planned` or `Justify with annotations`. For more information, see "Add Review Comments to Results".

Related Examples

- "Prioritize Check Review"
- "Filter and Group Results"
- "Reduce Orange Checks"

Critical Orange Checks

The software identifies a subset of orange checks that are most likely run-time errors. On the **Results Summary** pane, if you select **Show** > **Critical checks**, you can view this subset.

These orange checks are related to path and bounded input values. Here, input values refer to values that are external to the application. Examples include:

- Inputs to functions called by generated main. For more information on functions called by generated main, see “Functions to call (C)”.
- Global and volatile variables.
- Data returned by a stubbed function. The data can be the value returned by the function or a function parameter modified through a pointer.

In this section...

Path

The following example shows a path-related orange check that might be identified as a potential run-time error.

Consider the following code.

```
void path(int x) {
   int result;
   result = 1 / (x - 10);
   // Orange Division by Zero
 }

void main() {
    path(1);
    path(10);
 }
```

The software identifies the orange ZDV check as a potential error. The **Check Details** pane indicates the potential error:

```
...
Warning: scalar division by zero may occur
...
```

This **Division by Zero** check on `result=1/(x-10)` is orange because:

- `path(1)` does not cause a division by zero error.
- `path(10)` causes a division by zero error.

Polyspace indicates the definite division by zero error through a **Non-terminating call** error on `path(10)`. If you select the red check on `path(10)`, the **Check Details** pane provides the following information:

```
NTC .... Reason for the NTC: {path.x=10)
```

Bounded Input Values

Most input values can be bounded by data range specifications (DRS). The following example shows an orange check related to bounded input values that might be identified as a potential run-time error.

```
int tab[10];
extern int val;
// You specify that val is in [5..10]

void assignElement(int index) {
   int result;
   result = tab[index];
   // Orange Out of bounds array index
 }
void main(void) {
   assignElement(val);
}
```

If you specify a **PERMANENT** data range of 5 to 10 for the variable `val`, verification generates an orange **Out of bounds array index** check on `tab[index]`. The **Check Details** pane provides information about the potential error:

```
Warning: array index may be outside bounds: [0..9]
This check may be an issue related to bounded input values
Verifying DRS on extern variable val may remove this orange.
  array size: 10
```

```
 array index value: [5 .. 10]
```

Unbounded Input Values

The following example shows an orange check related to unbounded input values that might be identified as a potential run-time error:

```
int tab[10];
extern int val;

void assignElement(int index) {
   int result;
   result = tab[index];
   // Orange Out of bounds array index
 }
void main(void) {
   assignElement(val);
}
```

The verification generates an orange **Out of bounds array index** check on `tab[index]`. The **Check Details** pane provides information about the potential error:

```
Warning: array index may be outside bounds: [0..9]
This check may be an issue related to unbounded input values
If appropriate, applying DRS to extern variable val may remove this orange.
 array size: 10
 array index value: [-2^31 .. 2^31-1]
```

Reduce Orange Checks

An orange check indicates that Polyspace detects a possible run-time error but cannot prove it. To help Polyspace produce more proven results, you can:

- Follow appropriate coding practices.
- Specify appropriate verification options.

You can also limit the number and family of orange checks displayed on **Results Summary**. For more information, see “Limit Display of Orange Checks”.

Provide Context for Verification

This example shows how to provide additional information about run-time execution of your code. Sometimes, the code you provide does not contain this information. For instance:

- You do not have a `main` function
- You have a function that is declared but not defined.
- You have function arguments whose values are available only at run-time.
- You have concurrently running functions that are intended for execution in a specific sequence.

Without sufficient information, Polyspace Code Prover cannot verify the presence or absence of run-time errors.

To provide more context for verification and reduce orange checks, use the following methods.

Method	Example
Define how the `main` generated by Polyspace initializes variables and calls functions	• C code: “Code Prover Verification” • C++ code: “Code Prover Verification”

Method	Example
Define a stub for functions whose definitions are not yet written.	“Constrain Data with Stubbing”
Define ranges for global variables and function arguments.	“Create Constraint Template After Verification”
Define execution sequence for multitasking code.	“Model Execution Sequence in Tasks”

Improve Verification Precision

This example shows how to improve the precision of your verification. Increased precision reduces orange checks, but increases verification time.

Use the following options. In the Polyspace user interface, the options appear on the **Configuration** pane under the **Precision** node.

Option	Purpose
“Precision level (C/C++)”	Specify the verification algorithm.
• “Verification level (C)” • “Verification level (C++)”	Specify the number of times the Polyspace verification process runs on your source code.
“Improve precision of interprocedural analysis (C/C++)”	Propagate greater information about function arguments into the called function.
“Sensitivity context (C/C++)”	If a function contains a red and green check on the same instruction from two different calls, display both checks instead of an orange check.

Follow Coding Rules

This example shows how to follow coding rules that help Polyspace Code Prover prove the presence or absence of run-time errors. If your code follows certain subsets of MISRA coding rules, Polyspace can verify the presence or absence of run-time errors more easily.

1 Check whether your code follows the relevant subset of coding rules.

a In the Polyspace user interface, on the **Configuration** pane, depending on the code, select one of the options under the **Coding Rules** node.

Type of Code	Option	Rule Description
Handwritten C code	**Check MISRA C:2004**	“Software Quality Objective Subsets (C:2004)”
Generated C code	**Check MISRA AC AGC**	“Software Quality Objective Subsets (AC AGC)”
Handwritten C++ code	**Check MISRA C++ rules**	“Software Quality Objective Subsets (C++)”

b From the option drop-down list, select `SQO-subset1` or `SQO-subset2`.

2 Run verification and review your results.

3 Fix the coding rule violations.

More About

- “Sources of Orange Checks”
- “Managing Orange Checks”

Identify Function Call Causing Orange Check

This tutorial shows how to find the function call that causes an orange check in the function body. If a function is called multiple times, an orange check can appear on an operation in the function body because different calls cause different colors on the operation. For example:

- Some calls cause a red check on the operation and the other calls cause a green check.
- Some calls cause an orange check on the operation and the other calls cause a green check.

In the first case, a red **Non-terminating call** check appears on function calls that cause a red check. Using this red check on a function call, you can sometimes identify whether that call is related to a particular orange check in the function body. In the second case, a green check appears on all the function calls. Use the option **Sensitivity context** to identify exactly which function calls cause a particular orange check in the function body.

For this tutorial, store the following code in `file.c`:

```
double getRatio(int num, int den) {
  return(num/den);
}

int input(void);

void main() {
  int i=1, j=1;
  double ratio;

  /* First division */
  ratio = getRatio(i,j);

  /* Second division */
  j=input();
  if(j>=0)
    ratio = getRatio(i,j);
}
```

1 Create a Polyspace project. Add `file.c` to the project.

2 Specify that you want to store call context information for the function `getRatio`.

   a On the **Configuration** pane, select **Precision**.

- **b** Under **Specific Construct Settings**, for **Sensitivity context**, select `custom`.
- **c** Click ✚ to add a new field. Enter `getRatio`.

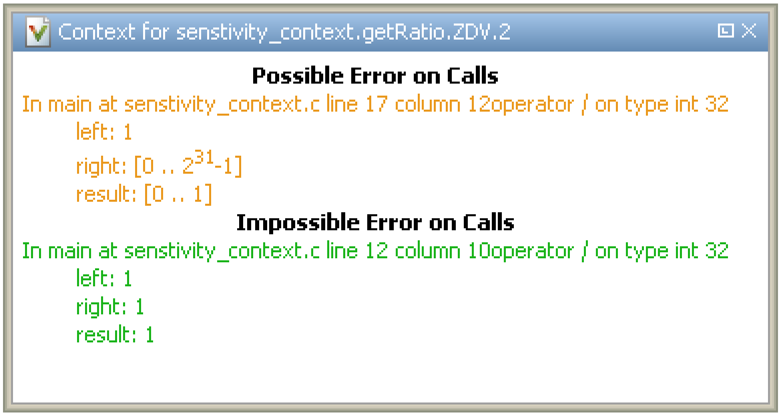
3 Run a verification and open the results.

An orange **Division by zero** check appears in the body of `getRatio`.

4 To identify which of the two calls to `getRatio` produces the orange check:

- **a** Select **Window** > **Show/Hide View** > **Sensitivity Context**.
- **b** On the **Results Summary** pane, select the orange check.

A window opens displaying the line numbers of the two calls. You can see that one call produces a green check on the division operation and the other call produces an orange check.



Context for senstivity_context.getRatio.ZDV.2

Possible Error on Calls

In main at senstivity_context.c line 17 column 12operator / on type int 32

left: 1

right: [0 .. 2^{31}-1]

result: [0 .. 1]

Impossible Error on Calls

In main at senstivity_context.c line 12 column 10operator / on type int 32

left: 1

right: 1

result: 1

See Also

"Sensitivity context (C/C++)"

Related Examples

- "Test Orange Checks for Run-Time Errors"

More About

- "Sources of Orange Checks"

Test Orange Checks for Run-Time Errors

This example shows how to run dynamic tests on orange checks. An orange check means that Polyspace static verification detects a possible error but cannot prove it. Orange checks can occur because of:

- Run-time errors.
- Approximations that Polyspace made during static verification.

By running tests, you can determine which orange checks represent run-time errors. Provided that you have emulated the run-time environment accurately, if a dynamic test fails, the orange check represents a run-time error. For this example, save the following code in a file `test_orange.c`:

```
volatile int r;
#include <stdio.h>

int input() {
 int k;
 k = r%21 - 10;
 // k  has value in [-10,10]
 return k;
}


void main() {
int x=input();
printf("%.2f",1.0/x);
}
```

Run Tests for Full Range of Input

Note: The Automatic Orange Tester is not supported on Mac.

1. Create a Polyspace project. Add `test_orange.c` to your project.
2. On the **Configuration** pane, under **Advanced Settings**, select **Automatic Orange Tester**.

After verification, Polyspace generates additional source code that tests each orange check for run-time errors. The software compiles this instrumented code. When you run the automatic orange tester later, the software tests the resulting binary code.

3. Run a verification and open the results.

   An orange **Division by Zero** error appears on the operation `1.0/x`.
4. Select **Tools** > **Automatic Orange Tester**.
5. In the Automatic Orange Tester window, click **Start**.

   The Automatic Orange Tester runs tests on your code. If the tests take too long, use the **Stop All** button to stop the tests. Reduce **Number of tests** before running again.
6. After the tests are completed, under **AOT Results**, view the number of **Tests that detected run-time errors**.

   The orange **Division by Zero** check represents a run-time error, so you see test case failures.
7. On the **Results** tab, click the row describing the check.

   A Test Case Detail window shows:

   - The operation on which the tests were run.
   - The test cases that failed with the reason for the failure.

Run Tests for Specified Range of Input

The Automatic Orange Tester window lists variables that cause orange checks. Because Polyspace does not have sufficient information about these variables, it makes assumptions about their range. If you know the variable range, you can specify the variables before running dynamic tests on orange checks. For pointer variables, you can specify the amount of memory written through the pointer. For instance, if the pointer points to an array, you can specify whether the first element of the array or the entire array is written through the pointer.

1. On the row describing `r`, click **Advanced**.
2. In the Edit Values window, under **Variable Values**, select **Range of values**.
3. Specify a minimum value of 1 and maximum of 9 for `r`. The Automatic Orange Tester saves the range as a `.tgf` file in the `Polyspace-Instrumented` folder in your results folder.

4 Click **Start** to restart tests on the orange **Division by Zero** check for `r` in `[1,9]`.

   A division by zero cannot occur for `r` in `[1,9]`, so there are no test failures. Although a test failure indicates a run-time error for specified inputs, because of the finite number of tests performed, the absence of test failures does not mean absence of a run-time error.

5 To prove that your range converts the orange check into a green check, you must provide the range during another static verification.

   a Select **File** > **Export DRS**.

   b Save your ranges as a text file.

   c Before running the next verification, on the **Configuration** pane, under **Inputs & Stubbing**, provide the text file for **Variable/function range setup**.

   d Run a verification and open your results.

      Instead of orange, there is a green **Division by Zero** check on the operation `1.0/x`.

Related Examples

- "Prioritize Check Review"
- "Identify Function Call Causing Orange Check"

More About

- "Limitations of Automatic Orange Tester"
- "Sources of Orange Checks"
- "Managing Orange Checks"

Limitations of Automatic Orange Tester

The Automatic Orange Tester has the following limitations:

In this section...

Unsupported Platforms

The Automatic Orange Tester is not supported on Mac.

Unsupported Polyspace Options

The software does not support the following options with `-automatic-orange-tester`.

- `-div-round-down`
- `-char-is-16its`
- `-short-is-8bits`

In addition, the software does not support global asserts in the code of the form `Pst_Global_Assert(A,B)` .

Options with Restrictions

Do not specify the following with `-automatic-orange-tester`:

- `-target [c18 | tms320c3c | x86_64 | sharc21x61]`
- `-data-range-specification` (in global assert mode)

You must use the `-target mcpu` option together with `-pointer-is-32bits`.

Unsupported C Routines

The software does not support verification of C code that contains calls to the following routines:

- `va_start`
- `va_arg`
- `va_end`
- `va_copy`
- `setjmp`
- `sigsetjmp`
- `longjmp`
- `siglongjmp`
- `signal`
- `sigset`
- `sighold`
- `sigrelse`
- `sigpause`
- `sigignore`
- `sigaction`
- `sigpending`
- `sigsuspend`
- `sigvec`
- `sigblock`
- `sigsetmask`
- `sigprocmask`
- `siginterrupt`
- `srand`
- `srandom`
- `initstate`
- `setstate`

Coding Rule Sets and Concepts

Rule Checking

Polyspace Coding Rule Checker

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

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

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

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

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

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

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

Differences Between Bug Finder and Code Prover

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

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

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

Polyspace MISRA C 2004 and MISRA AC AGC Checkers

The Polyspace MISRA C:2004 checker helps you comply with the MISRA C 2004 coding standard.[3]

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

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

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

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

- "Software Quality Objective Subsets (C:2004)" on page 12-5
- "Software Quality Objective Subsets (AC AGC)" on page 12-10

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

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

Software Quality Objective Subsets (C:2004)

In this section...

Rules in `SQO-Subset1`

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

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
14.4	The *goto* statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.

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

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

Rules in `SQO-Subset2`

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function

Rule number	Description
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression
10.5	Bitwise operations shall not be performed on signed integer types
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.1	Limited dependence should be placed on C's operator precedence rules in expressions
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits
12.5	The operands of a logical && or || shall be primary-expressions
12.6	Operands of logical operators (&&, || and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, || or !)
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used

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

Rule number	Description
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

Note: Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \
return -1 else return 0; }
```

Software Quality Objective Subsets (AC AGC)

Rules in `SQO-Subset1`

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

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation.*

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits

Rule number	Description
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.

Note: Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \
return -1 else return 0; }
```

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

MISRA C:2004 Coding Rules

In this section...

Supported MISRA C:2004 Rules

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

Note: The Polyspace coding rules checker:

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

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

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

Environment

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1	All code shall conform to ISO 9899:1990 “Programming languages - C”, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	The text *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996* precedes each of the following messages: • ANSI C does not allow ‘#include_next' • ANSI C does not allow macros with variable arguments list	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• ANSI C does not allow ‘#assert’ • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1 (cont.)		The text *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/ COR2:1996* precedes each of the following messages: • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int. • Too many nesting levels of #includes: N_1. The limit is N_0. • Too many macro definitions: N_1. The limit is N_0. • Too many nesting levels for control flow: N_1. The limit is N_0.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
		- Too many enumeration constants: N_1. The limit is N_0.	

Language Extensions

N.	MISRA Definition	Messages in report file	Polyspace Specification
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	No warnings if code is encapsulated in `asm` functions or in `asm pragma` (only warning is given on asm statements even if it is encapsulated by a MACRO).
2.2	Source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule **Note**: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. **Note**: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Specification
3.4	All uses of the *#pragma* directive shall be documented and explained.	All uses of the #pragma directive shall be documented and explained.	To check this rule, the option `-allowed-pragmas` must be set to the list of pragmas that are allowed in source files. Warning if a pragma that does not belong to the list is found.

Character Sets

N.	MISRA Definition	Messages in report file	Polyspace Specification
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	\<character> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in report file	Polyspace Specification
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters	Identifier 'XX' should not rely on the significance of more than 31 characters.	All identifiers (global, static and local) are checked.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	• Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier.	Assumes that rule 8.1 is not violated.
5.3	A typedef name shall be a unique identifier	{typedef name}'%s' should not be reused. (already used as {typedef name} at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name}'%s' should not be reused. (already used as {tag name} at %s:%d)	Warning when a tag name is reused as another identifier name
5.5	No object or function identifier with a static storage duration should be reused.	{static identifier/parameter name}'%s' should not be reused. (already used as {static identifier/parameter name} with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name Bug Finder and Code Prover check this coding rule

N.	MISRA Definition	Messages in report file	Polyspace Specification
			differently. The analyses can produce different results.
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name}'%s' should not be reused. (already used as {member name} at %s:%d)	Warning when an `idf` in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	No violation reported when: • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Polyspace Specification
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators, explicit casts to integral types and '?' (for the 2nd and 3rd operands)	Warning when a plain char is used with an operator other than =, ==, !=, explicit casts to integral types, or as the second or third operands of the ? operator.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	• Value of type plain char is implicitly converted to signed char. • Value of type plain char is implicitly converted to unsigned char.	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• Value of type signed char is implicitly converted to plain char. • Value of type unsigned char is implicitly converted to plain char.	
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types	typedefs that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type *unsigned int* or *signed int*.	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type *signed int* shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule **6.4** is violated).

Constants

N.	MISRA Definition	Messages in report file	Polyspace Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	• Octal constants other than zero and octal escape sequences shall not be used. • Octal constants (other than zero) should not be used. • Octal escape sequences should not be used.	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	• Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition.	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off
8.4	If objects or functions are declared more than once their types shall be compatible.	• If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition.	Violations of this rule might be generated during the link phase. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.5	There shall be no definitions of objects or functions in a header file	• Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file.	Tentative of definitions are considered as definitions.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• Fragment of function should not be defined in a header file.	
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiples files.	Restricted to explicit extern declarations (tentative of definitions are ignored). Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative of definition for object XX • Global variable has multiples tentative of definitions • Undefined global variable XX	Tentative of definitions are considered as definitions, no warning on predefined symbols. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	Assumes that 8.1 is not violated. No warning if 0 uses. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Size of array 'XX' should be explicitly stated.	

Initialization

N.	MISRA Definition	Messages in report file	Polyspace Specification
9.1	All automatic variables shall have been assigned a value before being used.		Checked during code analysis. Violations displayed as Non-initialized variable results. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression	• Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type.	ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types. An expression of bool or enum types has int as underlying type. Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting). The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed | unsigned int are used for bitfield (Rule 6.4).
10.1 (cont)		• Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying	No violation reported when: • The implicit conversion is a type widening, without change of signedness if integer • The expression is an argument expression or a return expression

N.	MISRA Definition	Messages in report file	Polyspace Specification
		type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying type XX as argument of function whose corresponding parameter type is XX.	No violation reported when the following are all true: • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness if integer • The conversion does not change the representation of the constant value or the result of the operation • The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • it is not a conversion to a wider floating type, or • the expression is complex, or • the expression is a function argument, or • the expression is a return expression	• Implicit conversion of the expression from XX to XX that is not a wider floating type. • Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. • Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from XX to XX, but it is a complex expression. • Implicit conversion of complex floating expression from XX to XX. • Implicit conversion of floating expression of XX type in function return whose expected type is XX. • Implicit conversion of floating expression of XX type as argument of function whose corresponding parameter type is XX.	ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. No violation reported when: • The implicit conversion is a type widening • The expression is an argument expression or a return expression.

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

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

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from `NULL` or `(void*)0` do not give any warning.

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

Expressions

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	Limited dependence should be placed on C's operator precedence rules in expressions	
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	• The value of 'sym' depends on the order of evaluation. • The value of volatile 'sym' depends on the order of evaluation because of multiple accesses.	The expression is a simple expression of symbols (Unlike i = i++; no detection on tab[2] = tab[2]++;). Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1) and the comma operator is not used (rule 12.10).

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

N.	MISRA Definition	Messages in report file	Polyspace Specification

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

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

12.12	The underlying bit representations of floating-point values shall not be used.	The underlying bit representations of floating-point values shall not be used.	Warning when:
			• A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to `void` does not generate a warning.
			• A float is packed with another data type. For example:
			`union {`
			` float f;`
			` int i;`
			`} …`

12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.1	Assignment operators shall not be used in expressions that yield Boolean values.	Assignment operators shall not be used in expressions that yield Boolean values.	
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option `-boolean-types` may

N.	MISRA Definition	Messages in report file	Polyspace Specification
			increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type	The controlling expression of a for statement shall not contain any objects of floating type	If *for* index is a variable symbol, checked that it is not a float.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.5	The three expressions of a *for* statement shall be concerned only with loop control	• 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of for loops are allowed: (a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; (c) all three expressions shall be empty for a deliberate infinite loop.	Checked if the for loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.
13.6	Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the for loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.7	Boolean operations whose results are invariant shall not be permitted	• Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted.	During compilation, check comparisons with at least one constant operand.

Control Flow

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.1	There shall be no unreachable code.	There shall be no unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14.2	All non-null statements shall either have at lest one side effect however executed, or cause control flow to change	• All non-null statements shall either: • have at lest one side effect however executed, or • cause control flow to change	
14.3	All non-null statements shall either • have at lest one side effect however executed, or • cause control flow to change	A null statement shall appear on a line by itself	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when:

N.	MISRA Definition	Messages in report file	Polyspace Specification
			• there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The *goto* statement shall not be used.	The goto statement shall not be used.	
14.5	The *continue* statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one *break* statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement	• The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.9	An *if (expression)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement	• An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement	
14.10	All *if else if* constructs should contain a final *else* clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Specification

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional *break* statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.
15.3	The final clause of a *switch* statement shall be the *default* clause	The final clause of a switch statement shall be the default clause	
15.4	A *switch* expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option `-boolean-types` may increase the number of warnings generated.
15.5	Every *switch* statement shall have at least one *case* clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	Done by Polyspace software (Use the call graph in Polyspace Code Prover). Polyspace also partially checks this rule during the compilation phase.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule **8.6** is not violated.

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.10	If a function returns error information, then that error information shall be tested.	If a function returns error information, then that error information shall be tested.	Warning if a non-`void` function is called and the returned value is ignored. No warning if the result of the call is cast to `void`. No check performed for calls of `memcpy`, `memmove`, `memset`, `strcpy`, `strncpy`, `strcat`, or `strncat`.

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Polyspace Specification
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointer arithmetic shall only be applied to pointers that address an array or array element.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Array indexing shall be the only allowed form of pointer arithmetic.	Warning on operations on pointers. (`p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer).
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	
17.6	The address of an object with automatic storage shall not	Pointer to a parameter is an illegal return value. Pointer	Warning when assigning address to a global variable,

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

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Specification
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	• An object shall not be assigned to an overlapping object. • Destination and source of XX overlap, the behavior is undefined.	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.1	#include statements in a file shall only be preceded by other preprocessors directives or comments	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include directive is preceded by other things than preprocessor directives, comments, spaces or “new lines”.
19.2	Nonstandard characters should not occur in header file names in #include directives	• A message is displayed on characters ', \, " or / * between < and > in #include <filename> • A message is displayed on characters ', \or / * between " and " in #include "filename"	

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.
19.8	A function-like macro shall not be invoked without all of its arguments	• arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args.	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	If `x` is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: `#x`, `##x`, and `x##`. Otherwise, parentheses are required around `x`. The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if `x` appears as `(x)` or `(x,` or `,x)` or `,x,`.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	

19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as,
			`#ifndef <control macro>`
			`#define <control macro>`
			`<contents> #endif`
			or,
			`#ifndef <control macro>`
			`#error ...`
			`#else`
			`#define <control macro>`
			`<contents> #endif`
			it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	• '#elif' not within a conditional. • '#else' not within a conditional. • '#elif' not within a conditional. • '#endif' not within a conditional. • unbalanced '#endif'. • unterminated '#if' conditional. • unterminated '#ifdef' conditional. • unterminated '#ifndef' conditional.	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	• The macro '<name> shall not be redefined. • The macro '<name> shall not be undefined.	
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is **20.1**. Tentative of definitions are considered as definitions.

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	Warning for argument in library function call if the following are all true: • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`.
20.4	Dynamic heap memory allocation shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule **20.2** is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule **20.2** is not violated
20.6	The macro *offsetof*, in library <stddef.h>, shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.7	The *setjmp* macro and the *longjmp* function shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.10	The library functions atof, atoi and toll from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.12	The time handling functions of library <time.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Polyspace Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults.		Done by Polyspace. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Unsupported MISRA C:2004 Rules

The Polyspace coding rules checker does not check the following MISRA C:2004 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. They may concern documentation, dynamic aspects, or functional aspects of MISRA rules. The "**Polyspace Specification**" column describes the reason each rule is not checked.

Environment

Rule	Description	Polyspace Specification
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/ assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character	The documentation of compiler must be checked.

Rule	Description	Polyspace Specification
	significance and case sensitivity are supported for external identifiers.	
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	The documentation of compiler must be checked as this implementation is done by the compiler

Language Extensions

Rule	Description	Polyspace Specification
2.4 (Advisory)	Sections of code should not be "commented out"	It might be some pseudo code or code that does not compile inside a comment.

Documentation

Rule	Description	Polyspace Specification
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	The documentation of compiler must be checked. Error detection is based on undefined behavior, according to choices made for implementation- defined constructions. Documentation can not be checked.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	The documentation of compiler must be checked.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	The documentation of compiler must be checked.
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	The documentation of compiler must be checked.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	The documentation of compiler must be checked.

Structures and Unions

Rule	Description	Polyspace Specification
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C:2012 Checker

The Polyspace MISRA C:2012 checker helps you to comply with the MISRA C 2012 coding standard.[4]

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

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

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

There are additional subsets of MISRA C:2012 guidelines defined by Polyspace called Software Quality Objectives (SQO) that can have a direct or indirect impact on the precision of your results. When you set up checking, you can select these subsets. These subsets are defined in “Software Quality Objective Subsets (C:2012)” on page 12-54.

See Also

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

Related Examples

- “Set Up Coding Rules Checking”

More About

- “MISRA C:2012 Directives and Rules”
- “Software Quality Objective Subsets (C:2012)” on page 12-54

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

Software Quality Objective Subsets (C:2012)

In this section...

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

Guidelines in `SQO-Subset1`

The following set of MISRA C:2012 coding guidelines typically reduces the number of unproven results.

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

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

Guidelines in `SQO-Subset2`

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible

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

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

See Also

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

Related Examples

- “Set Up Coding Rules Checking”

More About

- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

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

Number	Category	AGC Category	Definition
Directive 1.1	Required	Required	Any implementation-defined behavior on which the output of the program depends shall be documented and understood
Directive 2.1	Required	Required	All source files shall compile without any compilation errors
Directive 3.1	Required	Required	All code shall be traceable to documented requirements
Directive 4.2	Advisory	Advisory	All usage of assembly language should be documented
Directive 4.4	Advisory	Advisory	Sections of code should not be “commented out”
Directive 4.5	Advisory	Readability	Identifiers in the same name space with overlapping visibility should be typographically unambiguous
Directive 4.7	Required	Required	If a function returns error information, then that error information shall be tested
Directive 4.8	Advisory	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden
Directive 4.12	Required	Required	Dynamic memory allocation shall not be used
Directive 4.13	Advisory	Advisory	Functions which are designed to provide operations on a resource should be called in an appropriate sequence
Rule 2.6	Advisory	Readability	A function should not contain unused label declarations
Rule 2.7	Advisory	Readability	There should be no unused parameters in functions

Number	Category	AGC Category	Definition
Rule 17.5	Advisory	Readability	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements
Rule 17.8	Advisory	Readability	A function parameter should not be modified
Rule 21.12	Advisory	Advisory	The exception handling features of <fenv.h> should not be used.
Rule 22.1	Required	Required	All resources obtained dynamically by means of Standard Library functions shall be explicitly released
Rule 22.2	Mandatory	Mandatory	A block of memory shall only be freed if it was allocated by means of a Standard Library function
Rule 22.3	Required	Required	The same file shall not be open for read and write access at the same time on different streams
Rule 22.4	Mandatory	Mandatory	There shall be no attempt to write to a stream which has been opened as read only
Rule 22.5	Mandatory	Mandatory	A pointer to a FILE object shall not be dereferenced
Rule 22.6	Mandatory	Mandatory	The value of a pointer to a FILE shall not be used after the associated stream has been closed

Polyspace MISRA C++ Checker

The Polyspace MISRA C++ checker helps you comply with theMISRA C++:2008 coding standard.[5]

When MISRA C++ rules are violated, the Polyspace MISRA C++ checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis. The MISRA C++ checker can check 185 of the 228 MISRA C++ coding rules.

There are subsets of MISRA C++ coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in “Software Quality Objective Subsets (C++)” on page 12-62.

Note: The Polyspace MISRA C++ checker is based on MISRA C++:2008 – “Guidelines for the use of the C++ language in critical systems." For more information on these coding standards, see http://www.misra-cpp.com.

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

Software Quality Objective Subsets (C++)

In this section...

SQO Subset 1 – Direct Impact on Selectivity

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

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

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

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

SQO Subset 2 – Indirect Impact on Selectivity

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

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

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

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

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

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

MISRA C++ Coding Rules

In this section...

Supported MISRA C++ Coding Rules

Language Independent Issues

N.	MISRA Definition	Polyspace Specification
0-1-1	A project shall not contain unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-2	A project shall not contain infeasible paths.	
0-1-7	The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-10	Every defined function shall be called at least once.	Detects if static functions are not called in their translation unit. Other cases are detected by the software.

General

N.	MISRA Definition	Polyspace Specification
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Lexical Conventions

N.	MISRA Definition	Polyspace Specification
2-3-1	Trigraphs shall not be used.	
2-5-1	Digraphs should not be used.	
2-7-1	The character sequence /* shall not be used within a C-style comment.	This rule cannot be annotated in the source code.
2-10-1	Different identifiers shall be typographically unambiguous.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	No detection for logical scopes: fields or member functions hiding outer scopes identifiers or hiding ancestors members.

N.	MISRA Definition	Polyspace Specification
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-5	The identifier name of a non-member object or function with static storage duration should not be reused.	For functions the detection is only on the definition where there is a declaration. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	If the identifier is a function and the function is both declared and defined then the violation is reported only once. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	
2-13-4	Literal suffixes shall be upper case.	

N.	MISRA Definition	Polyspace Specification
2-13-5	Narrow and wide string literals shall not be concatenated.	

Basic Concepts

N.	MISRA Definition	Polyspace Specification
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	
3-1-2	Functions shall not be declared at block scope.	
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	
3-2-1	All declarations of an object or function shall have compatible types.	
3-2-2	The One Definition Rule shall not be violated.	Report type, template, and inline function defined in source file
3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	
3-2-4	An identifier with external linkage shall have exactly one definition.	
3-3-1	Objects or functions with external linkage shall be declared in a header file.	
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	
3-9-1	The types used for an object, a function return type, or a function parameter	Comparison is done between current declaration and last seen declaration.

N.	MISRA Definition	Polyspace Specification
	shall be token-for-token identical in all declarations and re-declarations.	
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.	No detection in non-instantiated templates.
3-9-3	The underlying bit representations of floating-point values shall not be used.	

Standard Conversions

N.	MISRA Definition	Polyspace Specification
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.	
4-5-2	Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	
4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N	

Expressions

N.	MISRA Definition	Polyspace Specification
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	

N.	MISRA Definition	Polyspace Specification
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.	
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.	Assumes that `ptrdiff_t` is signed integer
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that `ptrdiff_t` is signed integer If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-5	There shall be no implicit floating-integral conversions.	This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.
5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.	
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	
5-0-11	The plain char type shall only be used for the storage and use of character values.	For numeric data, use a type which has explicit signedness.
5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-14	The first operand of a conditional-operator shall have type bool.	

N.	MISRA Definition	Polyspace Specification
5-0-15	Array indexing shall be the only form of pointer arithmetic.	Warning on operations on pointers. (p+I, I+p and p-I, where p is a pointer and I an integer, p[i] accepted).
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	Report when relational operator are used on pointers types (casts ignored).
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.	
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.	
5-2-1	Each operand of a logical && or || shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associativity (a && b && c), (a || b || c).
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.	
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	No violation if pointer types of operand and target are identical.

N.	MISRA Definition	Polyspace Specification
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.	"Extended to all pointer conversions including between pointer to struct object and pointer to type of the first member of the struct type. Indirect conversions through non-pointer type (e.g. int) are not detected."
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	Exception on zero constants. Objects with pointer type include objects with pointer to function type.
5-2-9	A cast should not convert a pointer type to an integral type.	
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.	
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.	
5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	
5-3-3	The unary & operator shall not be overloaded.	
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.	No warning on volatile accesses and function calls
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	

N.	MISRA Definition	Polyspace Specification
5-14-1	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses and function calls.
5-18-1	The comma operator shall not be used.	
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

Statements

N.	MISRA Definition	Polyspace Specification
6-2-1	Assignment operators shall not be used in sub-expressions.	
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.	
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	
6-4-2	All if ... else if constructs shall be terminated with an else clause.	Also detects cases where the last `if` is in the block of the last `else` (same behavior as JSF, stricter than MISRA C). Example: "`if` … `else` { `if` …{}}" raises the rule

N.	MISRA Definition	Polyspace Specification
6-4-3	A switch statement shall be a well-formed switch statement.	Return statements are considered as jump statements.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.	
6-4-6	The final clause of a switch statement shall be the default-clause.	
6-4-7	The condition of a switch statement shall not have bool type.	
6-4-8	Every switch statement shall have at least one case-clause.	
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.	
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	
6-5-3	The loop-counter shall not be modified within condition or statement.	Detect only direct assignments if for_index is known (see 6-5-1).
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.	
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	

N.	MISRA Definition	Polyspace Specification
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	
6-6-2	The goto statement shall jump to a label declared later in the same function body.	
6-6-3	The continue statement shall only be used within a well-formed for loop.	Assumes 6.5.1 to 6.5.6: so it is implemented only for supported 6_5_x rules.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	
6-6-5	A function shall have a single point of exit at the end of the function.	At most one return not necessarily as last statement for void functions.

Declarations

N.	MISRA Definition	Polyspace Specification
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	
7-3-2	The identifier main shall not be used for a function other than the global function main.	
7-3-3	There shall be no unnamed namespaces in header files.	
7-3-4	using-directives shall not be used.	
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	

N.	MISRA Definition	Polyspace Specification
7-4-2	Assembler instructions shall only be introduced using the asm declaration.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
7-4-3	Assembly language shall be encapsulated and isolated.	
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	
7-5-3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	
7-5-4	Functions should not call themselves, either directly or indirectly.	

Declarators

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

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

Classes

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

Derived Classes

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

Member Access Control

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

Special Member Functions

N.	MISRA Definition	Polyspace Specification
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.	
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	

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

Templates

N.	MISRA Definition	Polyspace Specification
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	
14-8-1	Overloaded function templates shall not be explicitly specialized.	All specializations of overloaded templates are rejected even if overloading occurs after the call.

N.	MISRA Definition	Polyspace Specification
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	

Exception Handling

N.	MISRA Definition	Polyspace Specification
15-0-2	An exception object should not have pointer type.	NULL not detected (see 15-1-2).
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	
15-1-2	NULL shall not be thrown explicitly.	
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.	
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions.	Detect that there is no try/catch in the main and that the catch does not handle all exceptions. Not detected if no "main". Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	
15-3-5	A class type exception shall always be caught by reference.	
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all	

N.	MISRA Definition	Polyspace Specification
	of its bases, the handlers shall be ordered most-derived to base class.	
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.	
15-5-1	A class destructor shall not exit with an exception.	Limit detection to throw and catch that are internals to the destructor; rethrows are partially processed; no detections in nested handlers.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	Limit detection to throw that are internals to the function; rethrows are partially processed; no detections in nested handlers.

Preprocessing Directives

N.	MISRA Definition	Polyspace Specification
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.	
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.	
16-0-3	#undef shall not be used.	
16-0-4	Function-like macros shall not be defined.	
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be	

N.	MISRA Definition	Polyspace Specification
	enclosed in parentheses, unless it is used as the operand of # or ##.	
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	
16-2-1	The preprocessor shall only be used for file inclusion and include guards.	The rule is raised for #ifdef/#define if the file is not an include file.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Include guards shall be provided.	
16-2-4	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	The \ character should not occur in a header file name.	
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.	
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	
16-3-2	The # and ## operators should not be used.	

Library Introduction

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

Language Support Library

N.	MISRA Definition	Polyspace Specification
18-0-1	The C library shall not be used.	
18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.	
18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	The option `-dialect iso` must be used to detect violations, for example, `exit`.
18-0-4	The time handling functions of library <ctime> shall not be used.	
18-0-5	The unbounded functions of library <cstring> shall not be used.	
18-2-1	The macro offsetof shall not be used.	
18-4-1	Dynamic heap memory allocation shall not be used.	
18-7-1	The signal handling facilities of <csignal> shall not be used.	

Diagnostic Library

N.	MISRA Definition	Polyspace Specification
19-3-1	The error indicator errno shall not be used.	

Input/output Library

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

Unsupported MISRA C++ Rules

Language Independent Issues

N.	MISRA Definition	Polyspace Specification
0–1–3	A project shall not contain unused variables.	
0-1-4	A project shall not contain non-volatile POD variables having only one use.	
0-1-5	A project shall not contain unused type declarations.	
0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	

N.	MISRA Definition	Polyspace Specification
0-1-8	All functions with void return type shall have external side effects.	
0-1-9	There shall be no dead code.	Not checked by the coding rules checker. Can be enforced through detection of dead code during analysis.
0-1-11	There shall be no unused parameters (named or unnamed) in nonvirtual functions.	
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	
0-2-1	An object shall not be assigned to an overlapping object.	
0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	
0-3-2	If a function generates error information, then that error information shall be tested.	
0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.	
0-4-2	Use of floating-point arithmetic shall be documented.	
0-4-3	Floating-point implementations shall comply with a defined floating-point standard.	

General

N.	MISRA Definition	Polyspace Specification
1-0-2	Multiple compilers shall only be used if they have a common, defined interface.	

N.	MISRA Definition	Polyspace Specification
1-0-3	The implementation of integer division in the chosen compiler shall be determined and documented.	

Lexical Conventions

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

Standard Conversions

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

Expressions

N.	MISRA Definition	Polyspace Specification
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool.	
5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	

N.	MISRA Definition	Polyspace Specification
5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Declarations

N.	MISRA Definition	Polyspace Specification
7-1-1	A variable which is not modified shall be const qualified.	
7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	
7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	
7-4-1	All usage of assembler shall be documented.	

Classes

N.	MISRA Definition	Polyspace Specification
9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	
9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	

Templates

N.	MISRA Definition	Polyspace Specification
14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	

N.	MISRA Definition	Polyspace Specification
14-7-1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	

Exception Handling

N.	MISRA Definition	Polyspace Specification
15-0-1	Exceptions shall only be used for error handling.	
15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	
15-5-3	The terminate() function shall not be called implicitly.	

Preprocessing Directives

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

Library Introduction

N.	MISRA Definition	Polyspace Specification
17-0-3	The names of standard library functions shall not be overridden.	
17-0-4	All library code shall conform to MISRA C++.	

Polyspace JSF C++ Checker

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

6

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

Note: The Polyspace JSF C++ checker is based on JSF++:2005. For more information on these coding standards, see Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program.

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

JSF C++ Coding Rules

In this section...

Supported JSF C++ Coding Rules

- “Fault Handling” on page 12-117
- “Portable Code” on page 12-118

Code Size and Complexity

N.	JSF++ Definition	Polyspace Specification
1	Any one function (or method) **will** contain no more than 200 logical source lines of code (L-SLOCs).	Message in report file: `<function name>` has `<num>` logical source lines of code.
3	All functions **shall** have a cyclomatic complexity number of 20 or less.	Message in report file: `<function name>` has cyclomatic complexity number equal to `<num>`.

Environment

N.	JSF++ Definition	Polyspace Specification
8	All code **shall** conform to ISO/IEC 14882:2002(E) standard C++.	Reports the compilation error message
9	Only those characters specified in the C++ basic source character set **will** be used.	
11	Trigraphs **will not** be used.	
12	The following digraphs **will not** be used: `<%`, `%>`, `<:`, `:>`, `%:`, `%:%:`.	Message in report file: The following digraph will not be used: `<digraph>`. Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in `-dialect iso`.
13	Multi-byte characters and wide string literals **will not** be used.	Report `L'c'`, `L"string"`, and use of `wchar_t`.
14	Literal suffixes **shall** use uppercase rather than lowercase letters.	
15	Provision **shall** be made for run-time checking (defensive programming).	Done with checks in the software.

Libraries

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

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Specification
26	Only the following preprocessor directives **shall** be used: `#ifndef`, `#define`, `#endif`, `#include`.	

N.	JSF++ Definition	Polyspace Specification
27	`#ifndef`, `#define` and `#endif` **will** be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files **will not** be used.	Detects the patterns `#if !defined`, `#pragma once`, `#ifdef`, and missing `#define`.
28	The `#ifndef` and `#endif` preprocessor directives **will** only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only `#ifndef`.
29	The `#define` preprocessor directive **shall not** be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: • 29.1 : The `#define` preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.
30	The `#define` preprocessor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` preprocessor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` preprocessor directive **will** only be used to include header (*.h) files.	

Header Files

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

Style

N.	JSF++ Definition	Polyspace Specification
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
41	Source lines **will** be kept to a length of 120 characters or less.	
42	Each expression-statement **will** be on a separate line.	Reports when two consecutive expression statements are on the same line.
43	Tabs **should** be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following if/else, do/while, for, and while statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) **will not** rely on significance of more than 64 characters.	

N.	JSF++ Definition	Polyspace Specification
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h'	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by the presence/absence of the underscore character. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by a mixture of case. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by letter `O`, with the number `0`.
50	The first word of the name of a class, structure, namespace, enumeration, or type created with `typedef` **will** begin with an uppercase letter. All others letters **will** be lowercase.	Messages in report file: • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
51	All letters contained in function and variables names **will** be composed entirely of lowercase letters.	Messages in report file: • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values **shall** be lowercase.	Messages in report file: • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files **will** always have file name extension of "`.h`".	`.H` is allowed if you set the option `-dos`.
53.1	The following character sequences **shall** not appear in header file names: `'`, `\`, `/*`, `//`, or `"`.	
54	Implementation files **will** always have a file name extension of ".cpp".	Not case sensitive if you set the option `-dos`.
57	The public, protected, and private sections of a class **will** be declared in that order.	
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument **will** be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.

N.	JSF++ Definition	Polyspace Specification
59	The statements forming the body of an if, else if, else, while, do ... while or for statement **shall** always be enclosed in braces, even if the braces form an empty block.	Messages in report file: • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block **will** be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block **will** have nothing else on the line except comments.	
62	The dereference operator ‘*’ and the address-of operator ‘&’ will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.

N.	JSF++ Definition	Polyspace Specification
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	Reports when the following characters are not directly connected to a white space: - . - -> - ! - ~ - - - ++ - — **Note:** A violation will be reported for "." used in float/double definition.

Classes

N.	JSF++ Definition	Polyspace Specification
67	Public and protected data **should** only be used in structs - not classes.	
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	Reports when default constructor, assignment operator, copy constructor or destructor is not declared.
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.
74	Initialization of nonstatic class members **will** be performed through the member initialization list rather than through assignment in the body of a constructor.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in `ctor` body. Message in report file: Initialization of nonstatic class members "`<field>`" will be performed through the member initialization list.

N.	JSF++ Definition	Polyspace Specification
75	Members of the initialization list **shall** be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator **shall** be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in report file: • `no copy constructor and no copy assign` • `no copy constructor` • `no copy assign` Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
77.1	The definition of a member function **shall not** contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function **shall** define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. **Note:** A violation is raised even if "new" is done in a "if/else".

N.	JSF++ Definition	Polyspace Specification
81	The assignment operator shall handle self-assignment correctly	Reports when copy assignment body does not begin with "`if (this != arg)`" A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement. A violation is raised when the `if` statement is followed by a statement other than the return statement.

82	An assignment operator **shall** return a reference to `*this`.	The following operators should return `*this` on method, and `*first_arg` on plain function.
		`operator=`
		`operator+=`
		`operator-=`
		`operator*=`
		`operator >>=`
		`operator <<=`
		`operator /=`
		`operator %=`
		`operator |=`
		`operator &=`
		`operator ^=`
		`prefix operator++`
		`prefix operator--`
		Does not report when no return exists.
		No special message if type does not match.
		Messages in report file:
		• An assignment operator shall return a reference to `*this`.
		• An assignment operator shall return a reference to its first arg.

N.	JSF++ Definition	Polyspace Specification
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.
88	Multiple inheritance **shall** only be allowed in the following restricted form: `n` interfaces plus `m` private implementations, plus at most one protected implementation.	Messages in report file: • Multiple inheritance on public implementation shall not be allowed: *`<public_base_class>`* is not an interface. • Multiple inheritance on protected implementation shall not be allowed : *`<protected_base_class_1>`*. • *`<protected_base_class_2>`* are not interfaces.
88.1	A stateful virtual base **shall** be explicitly declared in each derived class that accesses it.	
89	A base class **shall not** be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function **shall not** be redefined in a derived class.	Does not report for destructor. Message in report file: Inherited nonvirtual function `%s` shall not be redefined in a derived class.
95	An inherited default parameter **shall never** be redefined.	
96	Arrays **shall not** be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
97	Arrays **shall not** be used in interface.	Only to prevent array-to-pointer-decay. Not checked on private methods

N.	JSF++ Definition	Polyspace Specification
97.1	Neither operand of an equality operator (== or !=) **shall** be a pointer to a virtual member function.	Reports == and != on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Polyspace Specification
98	Every nonlocal name, except `main()`, **should** be placed in some namespace.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
99	Namespaces **will not** be nested more than two levels deep.	

Templates

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

Functions

N.	JSF++ Definition	Polyspace Specification
107	Functions **shall** always be declared at file scope.	
108	Functions with variable numbers of arguments **shall not** be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments **will not** be used.	
111	A function **shall not** return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.

N.	JSF++ Definition	Polyspace Specification
113	Functions **will** have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions **shall** be through return statements.	
116	Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through the software. Message in report file: Function <*F*> shall not call directly itself.
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

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

Declarations and Definitions

N.	JSF++ Definition	Polyspace Specification
135	Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
136	Declarations should be at the smallest feasible scope.	Reports when: • A global variable is used in only one function. • A local variable is not used in a statement (`expr`, `return`, `init` ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. **Note:** • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
137	All declarations at file scope should be static where possible.	
138	Identifiers **shall not** simultaneously have both internal and external linkage in the same translation unit.	
139	External objects will not be declared in more than one file.	Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in all translation units.
140	The register storage class specifier **shall not** be used.	
141	A class, structure, or enumeration **will not** be declared in the definition of its type.	

Initialization

N.	JSF++ Definition	Polyspace Specification
142	All variables **shall** be initialized before use.	Done with Non-initialized variable checks in the software.

N.	JSF++ Definition	Polyspace Specification
144	Braces **shall** be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct **shall not** be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Specification
147	The underlying bit representations of floating point numbers **shall not** be used in any way by the programmer.	Reports on casts with float pointers (except with `void*`).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace Specification
149	Octal constants (other than zero) **shall not** be used.	
150	Hexadecimal constants **will** be represented using all uppercase letters.	
151	Numeric values in code **will not** be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value `1, 0`) in expressions, non `-const` initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
151.1	A string literal shall not be modified.	Report when a `char*`, `char[]`, or `string` type is used not as `const`.

N.	JSF++ Definition	Polyspace Specification
		A violation is raised if a string literal (for example, “ “) is cast as a non `const`.

Variables

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

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Specification
153	Unions **shall not** be used.	
154	Bit-fields **shall** have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) **shall** be named and shall only be accessed via their names.	Reports unnamed bit-fields (unnamed fields are not allowed).

Operators

N.	JSF++ Definition	Polyspace Specification
157	The right hand operand of a && or || operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: • The right hand operand of a && operator shall not contain side effects. • The right hand operand of a || operator shall not contain side effects.
158	The operands of a logical && or || **shall** be parenthesized if the operands contain binary operators.	Messages in report file: • The operands of a logical && shall be parenthesized if the operands contain binary operators. • The operands of a logical || shall be parenthesized if the operands contain binary operators.

N.	JSF++ Definition	Polyspace Specification
		Exception for: `X || Y || Z , Z&&Y &&Z`
159	Operators ||, &&, and unary & **shall not** be overloaded.	Messages in report file: • Unary operator & shall not be overloaded. • Operator || shall not be overloaded. • Operator && shall not be overloaded.
160	An assignment expression **shall** be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values **shall not** be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic **shall not** be used.	
164	The right hand operand of a shift operator **shall** lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator **shall not** have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator **shall not** be applied to an unsigned expression.	
166	The `sizeof` operator **will not** be used on expressions that contain side effects.	
168	The comma operator **shall not** be used.	

Pointers and References

N.	JSF++ Definition	Polyspace Specification
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection **shall not** be used.	Only reports on variables/parameters.

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

Type Conversions

N.	JSF++ Definition	Polyspace Specification
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".

N.	JSF++ Definition	Polyspace Specification
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases).	Reports explicit down casting, dynamic_cast included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.	Reports this specific down cast. Allows dynamic_cast.

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float`
		`float => integer`
		Does not report for cast to `bool` reports for implicit cast on constant done with the options `-scalar-overflows-checks signed-and-unsigned` or `-ignore-constant-overflows`
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` are also reported.
182	Type casting from any type to or from pointers **shall not** be used.	Does not report when Rule 181 applies.

N.	JSF++ Definition	Polyspace Specification
184	Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Specification
186	There **shall** be no unreachable code.	Done with gray checks in the software. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	

N.	JSF++ Definition	Polyspace Specification
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` loop **will** perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Specification
202	Floating point variables **shall not** be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions **shall not** lead to overflow/underflow.	Done with overflow checks in the software.
204	A single operation with side-effects shall only be used in the following contexts: • by itself	Reports when: • A side effect is found in a return statement

N.	JSF++ Definition	Polyspace Specification
	• the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation	• A side effect exists on a single value, and only one operand of the function call has a side effect.
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once **Note:** Read-write operations such as ++, are only considered as a write.
205	The volatile keyword **shall not** be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

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

Fault Handling

N.	JSF++ Definition	Polyspace Specification
208	C++ exceptions **shall not** be used.	Reports `try`, `catch`, `throw spec`, and `throw`.

Portable Code

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

215	Pointer arithmetic **will not** be used.	Reports:
		`p + I`
		`p - I`
		`p++`
		`p--`
		`p+=`
		`p-=`
		Allows `p[i]`.

Unsupported JSF++ Rules

- "Code Size and Complexity" on page 12-119
- "Rules" on page 12-119
- "Environment" on page 12-120
- "Libraries" on page 12-120
- "Header Files" on page 12-120
- "Style" on page 12-120
- "Classes" on page 12-121
- "Namespaces" on page 12-122
- "Templates" on page 12-122
- "Functions" on page 12-123

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

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

Style

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

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	The invariant for a class should be: • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.

N.	JSF++ Definition
91	Public inheritance will be used to implement “is-a” relationships.
92	A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: **1** with respect to the template in isolation considering assumptions or requirements placed on its arguments. **2** with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.
103	Constraint checks should be applied to template arguments.
105	A template definition’s dependence on its instantiation contexts should be minimized.

N.	JSF++ Definition
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
117	Arguments should be passed by reference if NULL values are not possible: • **117.1** – An object should be passed as `const T&` if the function should not change the value of the object. • **117.2** – An object should be passed as `T&` if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
122	Trivial accessor and mutator functions should be inlined.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
127	Code that is not used (commented out) shall be deleted. **Note**: This rule cannot be annotated in the source code.
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.

N.	JSF++ Definition
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

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

Types

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

Unions and Bit Fields

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

Operators

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

Type Conversions

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

Expressions

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

Memory Allocation

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

Portable Code

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

N.	JSF++ Definition
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

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

Miscellaneous

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

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Checking Coding Rules

Set Up Coding Rules Checking

You can use Polyspace to look for coding rule violations. You can look for violation of:

- MISRA or JSF coding rules.

  For more information, see “Coding Rules”.
- Naming conventions for identifiers that you define.

  For more information, see “Custom Coding Rules”.

With the exception of certain rules, Polyspace checks for coding rule violations during the compilation phase. If you want to check for coding rules only, you can run Polyspace on your source code upto this phase. For information on the analysis option that controls upto which phase you can run, see “Verification level (C)” or “Verification level (C++)”.

In this section...
“Select Predefined Coding Rule Sets” on page 13-2
“Select Specific MISRA or JSF Coding Rules” on page 13-3
“Create Coding Rules” on page 13-3

Select Predefined Coding Rule Sets

You can check for violations of predefined subsets of MISRA or JSF coding rules.

User Interface	Command Line
Select your project configuration. On the **Configuration** pane, select **Coding Rules**. Select an appropriate option. For more information on the options, see the following: • C code: “Coding Rules” • C++ code: “Coding Rules”	Use the appropriate option with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. For more information on the options, see the section **Command-Line Information** in the following: • C code: “Coding Rules” • C++ code: “Coding Rules”

Select Specific MISRA or JSF Coding Rules

You can check for violations of specific MISRA or JSF coding rules.

User Interface	Command Line

1 Select your project configuration. On the **Configuration** pane, select **Coding Rules**.
2 Select the option corresponding to the coding rules whose violations you want to check.
3 From the dropdown list for the option, select `custom`.
The software displays a new field for your custom file.
4 To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.
Select **On** for the rules you want to check.
5 Click **OK** to save the rules and close the window.
The full path to the rules file appears. To reuse this rules file for other projects, type this path name or use the 📁 icon in the New File window.
	1 Create a coding rules file in one of the two ways:
	- Create the file from the user interface.
	- Enter the rules directly in a text file.
	On each line, specify a coding rule in the format:
	`Rule_number on|off #Comments`
	For example:
	`10.5 off # rule 10.5: essential type model`
	`17.2 on # rule 17.2: functions`
	2 Use the appropriate option with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function.
	Provide the full path to the file you created as option argument. For example:
	`polyspace-code-prover-nodesktop -misra3 C:\m`

Create Coding Rules

You can create your own coding rules to enforce naming conventions for identifiers in your source code. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

User Interface	Command Line

1 Select your project configuration. On the **Configuration** pane, select **Coding Rules**.
2 Select the **Check custom rules** box.
3 Click Edit.
The New File window opens, displaying a table of rule groups.
4 From the drop-down list **Set the following state to all Custom C**, select `Off`. Click **Apply**.
5 For each rule you want to turn on, specify the naming convention and associated error message if the convention is violated. For more information, see "Check custom rules (C/C++)".
For a tutorial with a specific code, see "Create Custom Coding Rules".
	1 Create a coding rules file in one of the two ways:
	• Create the file from the user interface.
	• Enter the rules directly in a text file.
	On each line, specify a coding rule in the format:
	`Rule_number` `on|off` *`#Comments`*
	For example:
	`10.5 off # rule 10.5:`
	`          essential type model`
	`17.2 on # rule 17.2: functions`
	2 Use the option `-custom-rules` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function.
	Provide the full path to the file you created as option argument. For example:
	`polyspace-code-prover-nodesktop -custom-rule`
	`                           C:\myRulesFile.txt`

Related Examples

- "Exclude Files from Rules Checking"
- "Configure Additional Options for Certain Rules"
- "Review Coding Rule Violations"

Create Custom Coding Rules

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

For the high-level workflow, see “Create Coding Rules”.

The tutorial uses the following code stored in a file `printInitialValue.c`:

```
#include <stdio.h>

typedef struct {
int a;
int b;
} collection;

void main()
{
 collection myCollection={0,0};
 printf("Initial values in the collection are %d
                    and %d.",myCollection.a,myCollection.b);
}
```

1 Create Coding Rules File

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

   The New File window opens, displaying a table of rule groups.
4. From the drop-down list **Set the following state to all Custom C**, select `Off`. Click **Apply**.
5. Expand the **Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
On	Select ●.
Convention	Enter `All struct fields must begin with s_ and have capital letters`
Pattern	Enter `s_[A-Z0-9_]`
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

2. **Review Coding Rule Violations**

   1. Save the file and run the verification. On the **Results Summary** pane, you see two violations of rule 4.3. Select the first violation.

      a. On the **Source** pane, the line `int a;` is marked.

      b. On the **Check Details** pane, you see the error message you had entered, `All struct fields must begin with s_ and have capital letters`

   2. Right-click on the **Source** pane and select **Open Editor**. The file `printInitialValue.c` opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.

   3. In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the verification.

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

Related Examples

- “Exclude Files from Rules Checking”

More About

- “Rule Checking”
- “Format of Custom Coding Rules File”

Format of Custom Coding Rules File

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

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

- *`N.n`* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `on` — The software checks for violation of the rule. After verification, it displays the coding rule violation on the **Results Summary** pane.
- *`violation_message`* — Software displays this text in an XML file within the *`Results`*/`Polyspace-Doc` folder.
- *`regular_expression`* — Software compares this text pattern against a source code identifier that is specific to the rule. See “Custom Coding Rules”.

  The keywords `convention=` and `pattern=` are optional. If present, they apply to the rule whose number immediately precedes these keywords. If `convention=` is not given for a rule, then a standard message is used. If `pattern=` is not given for a rule, then the default regular expression is used, that is, `.*`.

  Use the symbol # to start a comment. Comments are not allowed on lines with the keywords `convention=` and `pattern=`.

  The following example contains three custom rules: 1.1, 8.1, and 9.1.

```
# Custom rules configuration file
1.1  off           # Disable custom rule number 1.1
8.1  on        # Violation of custom rule 8.1 produces a warning
convention=Global constants must begin by G_ and must be in capital letters.
pattern=G_[A-Z0-9_]*
9.1  on    # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

Related Examples

- “Create Custom Coding Rules”

Exclude Files from Rules Checking

You can exclude certain files from coding rules checking. The excluded files are still included during Code Prover verification.

Exclude Files

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Inputs & Stubbing**. 2 Select the **Files and folders to ignore** check box. 3 Select an option from the drop-down list. For instance, to exclude all include files, select `all-headers`. For more information, see: • "Files and folders to ignore (C)" • "Files and folders to ignore (C++)"	Use the option `-includes-to-ignore` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. For information on the option argument, see the section **Command-Line Information** in: • "Files and folders to ignore (C)" • "Files and folders to ignore (C++)"

Exclude Certain Header Files

If you choose the option value `all-headers` for excluding files from coding rules checking, Polyspace excludes all header files.

- In the user interface, Polyspace excludes all header files in folders that you specify under **Include**.
- At the command line, Polyspace excludes all header files in folders that you specify with the `-I` option.

To exclude only certain header files, do the following:

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Inputs & Stubbing**. 2 Select the **Files and folders to ignore** check box. 3 Do one of the following: • Select the option `all-headers` from the drop-down list. In your project, add to **Source** node the header files, where you want coding rules checking. For more information, see “Add Sources and Includes”. Use this approach if you have few header files that you want to include. • Select the option `custom`. On a new line, add the name of each header file or folder containing header files where you do not want coding rules checking. All other header files will be checked for coding rule violations. Use this approach if you have few header files that you want to exclude.	Do one of the following: • Use the option `-includes-to-ignore all-headers` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. To specify the header files where you want coding rules checking, add them using the `-sources` option. Use this approach if you have few header files that you want to include. • Use the option `-includes-to-ignore` with the `polyspace-code-prover-nodesktop` command. As the option argument, specify the names of header files or folders containing header files where you do not want coding rules checking. All other header files will be checked for coding rule violations. For information on the syntax, see the section **Command-Line Information** in: • “Files and folders to ignore (C)” • “Files and folders to ignore (C++)” Use this approach if you have few header files that you want to exclude.

Related Examples

- “Set Up Coding Rules Checking”

More About

- “Rule Checking”

Configure Additional Options for Certain Rules

To check certain MISRA C rules, you must provide additional information outside your code.

In this section...
"Specify Allowed Custom Pragma Directives" on page 13-10
"Specify Effective Boolean Types" on page 13-10

Specify Allowed Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C:2004 rule 3.4 requires checking that all pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Coding Rules**. **2** Select **Check MISRA C:2004** or **Check MISRA AC AGC**. The **Allowed pragmas** option appears. **3** Enter the allowed pragma names. For more information, see "Allowed pragmas (C)".	Use the option `-allowed-pragmas` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. For more information, see the section **Command-Line Information** in "Allowed pragmas (C)".

Specify Effective Boolean Types

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

- MISRA C:2004 — 12.6, 13.2, 15.4.

For more information, see “MISRA C:2004 Rules”.

- MISRA C:2012
  - MISRA C:2012 Rule 14.4
  - MISRA C:2012 Rule 16.7

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Coding Rules**. **2** Select one of the predefined coding rule options. The **Effective boolean types** option appears. **3** Enter the type names. For more information, see “Effective boolean types (C)”.	Use the option `-boolean-types` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. For more information, see the section **Command-Line Information** in “Effective boolean types (C)”.

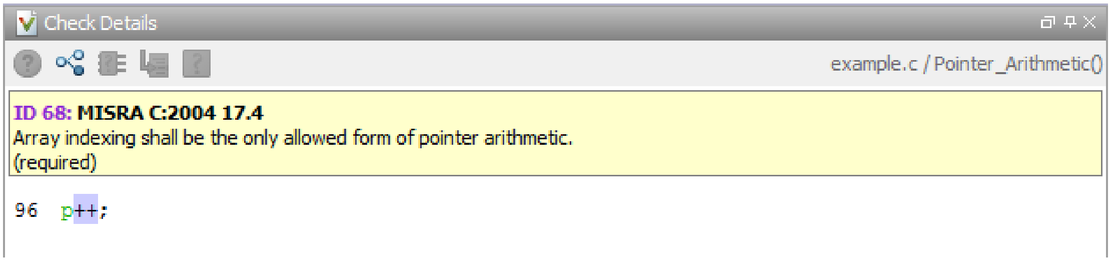
Review Coding Rule Violations

This example shows how to review coding rule violations in the Polyspace user interface once code analysis is complete. After analysis, the **Results Summary** pane displays the rule violations.

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

   - The predefined rules such as MISRA or JSF are indicated by ▽ .
   - The custom rules are indicated by ▼ .

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



ID 68: MISRA C:2004 17.4
Array indexing shall be the only allowed form of pointer arithmetic.
(required)

```
p++;
```

3 For MISRA C: 2012 rules, on the **Check Details** pane, click the ? icon to see the rationale for the rule. In some cases, you can also see code examples illustrating the violation.

4 Review the violation in your code. Determine whether you must fix the code to avoid the violation.

   - If you decide to fix the code, to open the source file that contains the coding rule violation, on the **Source** pane, right-click the code with the purple check. Select **Open Editor**. The file opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
   - Otherwise, add a comment and justification in your result or code explaining why you did not change the code. For more information, see “Add Review Comments to Results” or “Add Review Comments to Code”.

Related Examples

- "Set Up Coding Rules Checking"
- "Filter and Group Coding Rule Violations"

Filter and Group Coding Rule Violations

This example shows how to use filters in the **Results Summary** pane to focus on specific kinds of coding rule violations. By default, the software displays all coding rule violations and run-time checks.

Group Violations

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

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

Filter Violations

1. On the **Results Summary** pane, place your cursor on the **Check** column header. Click the filter icon that appears.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

Related Examples

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Filter and Group Results"

Generate Coding Rules Report

This example shows how to generate and view a coding rules report after verification.

Generate Report

1. With the results open, select **Reporting** > **Run Report**.
2. In the Run Report dialog box, from the **Select Reports** menu, select `CodingRules`.
3. Specify **Output folder** and **Output format**.
4. Select **Run Report**.

Open Existing Report

1. With the results open, select **Reporting** > **Open Report**.
2. In the Open Report dialog box, navigate to the folder that contains the coding rules report.

   The default location is in *`ResultFolder`*`\Polyspace-Doc`
3. Select the report and click **OK**.

View Report

In the coding rules report, you can view the following information:

- **Summary for all Files** — Lists number of violations in each file.
- **Summary for Enabled Rules** — For each rule, lists the:
  - Rule number.
  - Rule description.
  - Number of times the rule is broken.
- **Violations** — For each file that Polyspace checked for coding rule violations, lists each violation along with the:
  - Rule description.
  - Unique ID for the violation. Use this ID to find the violation on the **Results Summary** pane.
  - Function where the rule violation is found.
  - Line and column number.

  - Review information you enter such as **Class**, **Status** and **Comment**.
- **Configuration Settings** — Lists analysis options used for the verification, along with coding rules that Polyspace checked.

Related Examples

- “Set Up Coding Rules Checking”
- “Customize Report Templates”

Rules Not Checked in Compilation Phase

The following coding rules are not checked entirely during compilation phase:

- MISRA C:2004 — 9.1, 13.7 and 21.1

  For more information, see “MISRA C:2004 Rules”.
- MISRA C:2012
  - MISRA C:2012 Rule 2.2
  - MISRA C:2012 Rule 9.1
  - MISRA C:2012 Rule 14.3
  - MISRA C:2012 Rule 18.1

To check for violations of these rules, you must run your analysis past the compilation phase.

For all other rules, if you want to detect rule violations alone, you can run only the compilation phase of the analysis. For more information, see “Set Up Coding Rules Checking” on page 13-2.

Software Quality with Polyspace Metrics

Code Quality Metrics

Polyspace Metrics is a web dashboard that generates code quality metrics from your verification 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.

For each project or run, you can view the code quality metrics spread over four tabs, at project, file, and function level.

- The **Summary** tab provides a high-level overview of the verification results.
- The **Code Metrics** tab provides the details of the code complexity metrics in your results.

  See "Code Metrics".
- The **Coding rules** tab provides the details of the coding rule violations in your results.

  See "Coding Rules".
- The **Run-Time Checks** tab provides details of run-time checks in your results.

  See "Run-Time Checks".

If you turn on Software Quality Objectives, each tab also specifies how your project or run compares against those objectives. See "Compare Metrics Against Software Quality Objectives".

In this section...

Summary Tab

The **Summary** tab summarizes the verification results for a project or run.

To see the results embedded in your source code, download the results from Polyspace Metrics to the user interface. For more information, see “Review Metrics for Particular Project or Run”.

Column Name		Description
Verification Status		Verification level completed. See “Verification level (C)” or “Verification level (C++)”.
Code Metrics	**Files**	Number of files in project.
	Lines of code	Number of lines of code, broken down by file.
Coding Rules	**Confirmed Defects**	Number of coding rule violations to which you assign a **Classification** of `High`, `Medium` or `Low` in the Polyspace user interface. See “Add Review Comments to Results”.
	Violations	Total number of coding rule violations.
Run-Time Errors	**Confirmed Defects**	Number of run-time checks to which you assign a **Classification** of `High`, `Medium` or `Low` in the Polyspace user interface. See “Add Review Comments to Results”.
	Run-Time Reliability	A measure of your code quality, expressed as a percentage.

Column Name		**Description**
		The percentage is calculated as number of green and other justified checks divided by the total number of checks. To justify a check, in the Polyspace user interface, you must assign an appropriate **Status**. See “Add Review Comments to Results”.
Software Quality Objectives	**Overall Status**	A status of **PASS** or **FAIL** based on whether your code satisfies the software quality objectives you specified. For more information, see “Compare Metrics Against Software Quality Objectives”.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives. See: • “Predefined Software Quality Objectives” • “Customize Software Quality Objectives”

Column Name		Description
	Review Progress	A measure of your review progress, expressed as a percentage. The percentage is calculated as number of reviewed non-green checks and coding rule violations divided by the total number of non-green checks and rule violations. To review a check, in the Polyspace user interface, you must assign a **Status**. See "Add Review Comments to Results".
	Code Metrics over Threshold	Number of code metrics that exceed the limits that you specify.
	Justified Coding Rules	Percentage of coding rule violations that you have justified. To justify a rule violation, in the Polyspace user interface, you must assign an appropriate **Status**. See "Add Review Comments to Results".

Column Name		Description
	Justified Run-Time Errors	Percentage of run-time checks that you have justified. To justify a check, in the Polyspace user interface, you must assign an appropriate **Status**. See “Add Review Comments to Results”.

Code Metrics Tab

The **Code Metrics** tab lists the code complexity metrics for your project or run.

Some metrics are calculated at the project level, while others are calculated at file or function level. For metrics calculated at the function level, the entry displayed for a file is either an aggregate or a maximum over the functions in the file.

For more information, see “Code Metrics”.

Coding Rules Tab

The **Coding Rules** tab lists the coding rule violations in your project or run. For more information on the coding rules, see “Coding Rules”.

You can group the information in the columns by **Files** or **Coding Rules**.

Column Name		Description
Coding Rules	**Confirmed Defects**	Number of coding rule violations to which you assign a **Classification** of `High`, `Medium`, or `Low` in the Polyspace user interface. See “Add Review Comments to Results”.

Column Name		Description
	Justified	Number of coding rule violations that you have justified. To justify a rule violation, in the Polyspace user interface, assign an appropriate **Status**. See“Add Review Comments to Results”.
	Violations	Total number of coding rule violations.
Software Quality Objectives	**Quality Status**	A status of **PASS** or **FAIL** based on whether your code satisfies the software quality objectives you specified. See “Compare Metrics Against Software Quality Objectives”.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives, or specify your own objectives. See: • “Predefined Software Quality Objectives” • “Customize Software Quality Objectives”

Column Name		Description
	Review Progress	A measure of your review progress, expressed as a percentage. The percentage is calculated as the number of reviewed coding rule violations divided by the total number of violations. To mark a check as reviewed, in the Polyspace user interface, assign a **Status** to the check. See “Add Review Comments to Results”.

Run-Time Checks Tab

The **Run-Time Checks** tab lists the run-time checks in your project or run. For more information on the checks, see “Run-Time Checks”.

You can group the information in the columns by **Files** or **Run-Time Categories**.

Column Name		Description
Confirmed Defects		Number of run-time checks to which you assign a **Classification** of `High`, `Medium`, or `Low` in the Polyspace user interface. See “Add Review Comments to Results”.
Run-Time Reliability		A measure of your code quality, expressed as a percentage. The percentage is calculated as the number of green

Column Name		Description
		and other justified checks divided by the total number of checks. To justify a check, in the Polyspace user interface, assign an appropriate **Status**. See "Add Review Comments to Results".
Green Code	**Checks**	Number of green checks. See "Result and Source Code Colors".
Systematic Run-Time Errors (Red Checks)	**Justified**	Percentage of red checks that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate **Status**. See "Add Review Comments to Results".
	Checks	Number of red checks. See "Result and Source Code Colors".
Unreachable Branches (Gray Checks)	**Justified**	Percentage of gray checks that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate **Status**. See "Add Review Comments to Results".
	Checks	Number of gray checks. See "Result and Source Code Colors".

Column Name		Description
Other Run-Time Errors (Orange Checks)	**Justified**	Percentage of orange checks that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate **Status**. See "Add Review Comments to Results".
	Checks	Number of orange checks. See "Result and Source Code Colors".
	Path-Related Issues	Number of orange checks that indicate a run-time error only on certain execution paths. See "Critical Orange Checks".
	Bounded-Input Issues	Number of orange checks that indicate a run-time error only for certain inputs. You have specified external constraints on the inputs. See "Critical Orange Checks".
	Unbounded-Input Issues	Number of orange checks that indicate a run-time error only for certain inputs. You have not specified any external constraints on the inputs. See "Critical Orange Checks".

Column Name		Description
Non-terminating constructs	**Justified**	Percentage of non-terminating constructs that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate **Status**. See“Add Review Comments to Results”.
	Checks	Number of non-terminating constructs such as Non-terminating call and Non-terminating loop.
Software Quality Objectives	**Quality Status**	A status of **PASS** or **FAIL** based on whether your code satisfies the software quality objectives you specified. See “Compare Metrics Against Software Quality Objectives”.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives. See: • “Predefined Software Quality Objectives” • “Customize Software Quality Objectives”

Column Name		Description
	Review Progress	A measure of your review progress, expressed as a percentage. The percentage is calculated as the number of reviewed checks divided by the total number of checks. To mark a check as reviewed, in the Polyspace user interface, assign a **Status** to the check. See “Add Review Comments to Results”.

Related Examples

- “Generate Code Quality Metrics”
- “View Code Quality Metrics”
- “Compare Metrics Against Software Quality Objectives”
- “View Trends in Code Quality Metrics”

Generate Code Quality Metrics

After verification, you can upload the results to the Polyspace Metrics web interface. The web interface displays a summary of your verification results. You can share this summary with others even if they do not have Polyspace installed locally. You can also compare the results against previous verifications on the same project or measure them against predefined software quality objectives.

For more information, see “Code Quality Metrics”.

Before you generate code quality metrics, set up Polyspace Metrics. See “Set Up Polyspace Metrics”.

In this section...

Upload Results to Polyspace Metrics

If you perform a remote verification, you can specify that the results must be uploaded automatically to the web interface after verification. Otherwise, upload the results after verification manually.

- If you perform a remote verification:

  1. On the **Configuration** pane, select **Distributed Computing**.
  2. Along with **Batch**, select **Add to results repository**.

     After verification, the results are automatically uploaded to the web interface.
  3. If you do not select **Add to results repository**, after verification, the results are downloaded to your computer.

     To upload them later to the Polyspace Metrics web interface, select **Metrics > Upload to Metrics**.
  4. When you upload results to metrics, you are prompted to enter a password. Leave the field blank if you do not want to specify one.

     If you specify a password, you have to enter it every time you open your project in Polyspace Metrics. The session lasts for 30 minutes even if you close and reopen your web browser. After 30 minutes, enter your password again.

You can also specify a password later. On the Polyspace Metrics web interface, right-click your project and select **Change/Set Password**.

Note: The password for a Polyspace Metrics project is encrypted. The web data transfer is not encrypted. The password feature minimizes unintentional data corruption, but it does not provide data security. However, data transfers between a Polyspace Code Prover local host and the remote verification MJS host are always encrypted. To use a secure web data transfer with HTTPS, see "Configure Web Server for HTTPS".

- If you perform a local verification, to upload your results to the Polyspace Metrics web interface, select **Metrics** > **Upload to Metrics**.

For more information, see "View Code Quality Metrics".

Specify Automatic Uploading of Results

Note: This functionality will be removed in a future release.

You can:

- Configure verifications to start automatically and periodically, for example, at a specific time every night.
- Specify that Polyspace must upload your results automatically to the Polyspace Metrics web interface.
- Specify that Polyspace will send you an email after uploading the results. This email contains:
  - Links to results
  - If the verification produces compilation errors, an attached log file
  - A summary of new findings, for example, new coding rule violations, and new potential and actual run-time errors

To configure automatic verification and uploading of results:

1. Save the following content in an XML file. Name the file `Projects.psproj`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Polyspace Metrics Automatic Verification Project File -->
<Configuration>
<Project name="Demo_C" language="C" verificationKind="INTEGRATION"
product="CODE-PROVER">
  <Options>
    -O2
    -to pass2
    -target sparc
    -temporal-exclusions-file sources/temporal_exclusions.txt
    -entry-points tregulate,proc1,proc2,server1,server2
    -critical-section-begin Begin_CS:CS1
    -critical-section-end End_CS:CS1
    -misra2 all-rules
    -includes-to-ignore sources/math.h
    -D NEW_DEFECT
  </Options>
  <LaunchingPeriod hour="12" minute="20" month="*" weeDay="1">
  </LaunchingPeriod>
  <Commands>
    <GetSource>
      /bin/cp -vr /yourcompany/home/auser/tempfolder/Demo_C_Studio/sources/ .
    </GetSource>
    <GetVersion>
    </GetVersion>
  </Commands>
  <Users>
    <User>
      <FirstName>Polyspace</FirstName>
      <LastName>User</LastName>
      <Mail resultsMail="ALWAYS"
    compilationFailureMail="yes">userid@yourcompany.com</Mail>
    </User>
  </Users>
</Project>
<SmtpConfiguration server="smtp.yourcompany.com" port="25">
</SmtpConfiguration>
</Configuration>
```

2 Save this file in the results repository on the Polyspace Metrics server. For example:

```
/var/Polyspace/results-repository
```

3 Modify the contents of this file appropriately.

Replace	Replace with
`Demo_C`	Project name.
`C`	`C` or `CPP`: Source code language.
`INTEGRATION`	`INTEGRATION` or `UNIT-BY-UNIT`: Verification mode. For

Replace	Replace with
	more information on UNIT-BY-UNIT, see “Run File-by-File Remote Verification”.
CODE-PROVER	BUG-FINDER or CODE-PROVER: Product name.
<Options> -O2 -to pass2 -target sparc -temporal-exclusions-file sources/tempo -entry-points tregulate,proc1,proc2,ser -critical-section-begin Begin_CS:CS1 -critical-section-end End_CS:CS1 -misra2 all-rules -includes-to-ignore sources/math.h -D NEW_DEFECT </Options>	Your own set of options in <Options>..</Options>. For more information, see “Analysis Options”.
12	Integer in 0–23: Starting hour of verification.
20	Integer in 0–59: Starting minute of verification.
*	One of the following: • Integer in 1–12: Starting month of verification. • * to specify once every month. • *n1*-*n2* to specify a range. For example, 1-5.
1	One of the following: • Integer in 1–7: Starting weekday of verification. • * to specify once every day. • *n1*-*n2* to specify a range. For example, 1-5.

Replace	Replace with
`/bin/cp -vr /yourcompany/home/auser/tempfolder/Demo_C_Studio/sources/`	Optional command to retrieve source files from the configuration management system, or the file system of the user. The command is executed in a temporary folder on the client computer, which is also used to store results from the compilation phase of the verification. This temporary folder is removed after the verification is moved to the Polyspace server.
`Polyspace`	Your first name.
`User`	Your last name.
`ALWAYS`	One of the following: • `ALWAYS`: An email is always sent at the end of each automatic verification • `NEW-CERTAIN-FINDINGS`: An email is sent only if there are new red or gray checks. • `NEW-POTENTIAL-FINDINGS`: An email is sent only if there are new orange checks. • `NEW-CODING-RULES-FINDINGS`: An email is sent only if there are new coding rule violations. • `ALL-NEW-FINDINGS`: An email is sent if there are new checks or coding-rule violations.

Replace	Replace with
`yes`	`yes` or `no`: An email is sent if verification fails because of compilation failure.
`user_id@yourcompany.com`	Your email address.
`smtp.yourcompany.com`	Your Simple Mail Transport Protocol (SMTP) server.
`25`	Your SMTP server port.

Related Examples

- “View Code Quality Metrics”
- “Compare Metrics Against Software Quality Objectives”
- “View Trends in Code Quality Metrics”

View Code Quality Metrics

Before you can view software quality metrics, upload your results to the Polyspace Metrics repository. You can upload the results of a local verification or remote verification. For more information, see “Generate Code Quality Metrics”.

In this section...
“Open Metrics Interface” on page 14-19
“View All Projects and Runs” on page 14-19
“Review Metrics for Particular Project or Run” on page 14-21

Open Metrics Interface

You can open the metrics interface in one of the following ways:

- If you have a local installation of Polyspace, select **Metrics** > **Open Metrics**.
- If you do not have a local installation, enter the following URL in a web browser:

  ```
  protocol:// ServerName: PortNumber
  ```

  - `protocol` is either `http` (default) or `https`.

    To use HTTPS, set up the configuration file and the **Metrics configuration** preferences. For more information, see “Configure Web Server for HTTPS”.

  - `ServerName` is the name or IP address of your Polyspace Metrics server.
  - `PortNumber` is the web server port number (default 8080)

View All Projects and Runs

On the Polyspace Metrics interface, you can view either all projects or all runs.

- On the **Projects** tab, view all projects.

  On this tab, you can do the following:

Goal	Action
See number of project runs.	Hover your cursor over the project name.

Goal	Action
Group projects together.	Right-click a project. Select **Create Project Category**. Drag projects to your new category.
Filter projects from display.	In the field below the **Project** column header, enter the name of the project you want.
Delete project from the Metrics repository.	Right-click the project. Select **Delete Project from Repository**.
Assign password to project.	Right-click the project. Select **Change/ Set Password**.
See code quality metrics for all runs of project.	Click the project name. For more information, see “Review Metrics for Particular Project or Run”.

Tip If a new verification has been carried out for a project since your last visit, then on the **Projects** tab, the icon appears before the project name.

- If a project has multiple runs, on the **Runs** tab, view the individual runs. To identify different runs of the same project, use the **Project** and **Version** column.

  On this tab, you can do the following:

Goal	Action
Delete run from repository.	Right-click the run. Select **Delete Run from Repository**.
Assign password to run.	Right-click the run. Select **Change/Set Password**.
See runs between two specific dates.	Select the starting date in the **From** field and the end date in the **To** field.
See only the last *n* runs.	In the field **Maximum number of runs**, enter *n*.
See code quality metrics for a run.	Right-click the run. Select **Go to Metrics Page**. For more information, see “Review Metrics for Particular Project or Run”.

Goal	Action
Download results of run to Polyspace user interface.	Click the run name.

Review Metrics for Particular Project or Run

If you select a project on the **Projects** tab or **Go to Metrics Page** for a run on the **Runs** tab, you can view the code quality metrics for the project or run. A summary of the metrics appears on the **Summary** tab.

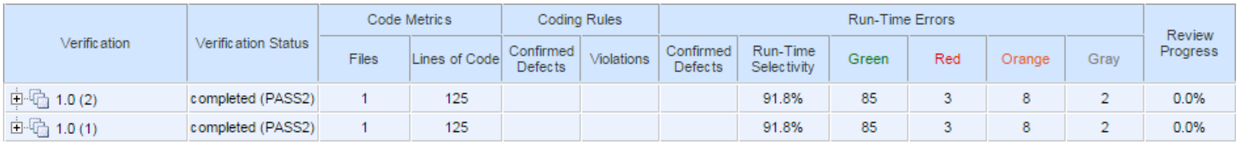
If you want to compare the code quality metrics against standards you have previously defined, before reviewing your results, you can turn on quality objectives. For more information, see "Compare Metrics Against Software Quality Objectives".

Otherwise, review the absolute values of code quality metrics on the **Summary** tab.

1. Select an entry on the **Summary** tab to open another tab with further details.

   - If you select an entry under the group **Code Metrics**, you can see your code complexity metrics on the **Code Metrics** tab.
   - If you select an entry under the group **Coding Rules**, you can see your coding rule violations on the **Coding Rules** tab.
   - If you select an entry under the group **Run-Time Errors**, you can see your run-time checks on the **Run-Time Checks** tab.

   For example, in the following metrics, there are three red checks. Select the entry in the **Red** column to view the checks on the **Run-time Checks** tab.



Verification	Verification Status	Code Metrics		Coding Rules		Run-Time Errors						Review Progress
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Run-Time Selectivity	Green	Red	Orange	Gray	
1.0 (2)	completed (PASS2)	1	125				91.8%	85	3	8	2	0.0%
1.0 (1)	completed (PASS2)	1	125				91.8%	85	3	8	2	0.0%

2. On the **Code Metrics**, **Coding Rules** or **Run-Time Errors** tabs, select an entry to download the result to the Polyspace user interface.
3. In the Polyspace user interface, review the particular result, investigate the root cause in your source code, and assign review comments and justifications.
4. To upload your comments and justifications to the Polyspace Metrics repository, select **Metrics > Save comments to Metrics**.

Tip To upload automatically your comments and justifications to the Polyspace Metrics repository when you save them:

a Select **Tools** > **Preferences**.

b On the **Server Configuration** tab, select **Save justifications in the Polyspace Metrics repository**.

5 After your review is over, in the Polyspace Metrics interface, click Refresh to view updated metrics.

Related Examples

- "View Trends in Code Quality Metrics"

Compare Metrics Against Software Quality Objectives

After generating metrics from your verification results and viewing them, you can review the results in further details. You can download each result into the Polyspace user interface, investigate it in your source code and add review comments to them. For more information, see “View Code Quality Metrics”.

However, before diving into the result details, to focus your review, you can:

1. Define quality objectives that you or developers in your organization must meet.
2. Apply the quality objectives to your verification results.
3. Review only those results that fail to meet those objectives.

In this section...
“Apply Predefined Objectives to Metrics” on page 14-23
“Customize Software Quality Objectives” on page 14-25

Apply Predefined Objectives to Metrics

By default, the software quality objectives are turned off. To apply quality objectives:

1. Open the Polyspace Metrics interface. View the metrics for a project or run on the **Summary** tab.

   For more information, see “View Code Quality Metrics”.
2. From the **Quality Objectives** list in the upper left, select `ON`.

   - A new group of **Software Quality Objectives** columns appears.
   - In the **Overall Status** column, the software uses the last used quality objective level to generate a status of **PASS** or **FAIL** for your results.
   - In the **Level** column, you can see the quality objective level used.

     To change your quality objective level, on this column, select a cell. From the drop-down list, select a quality level. For more information, see “Predefined Software Quality Objectives”.
3. For files with an **Overall Status** of **FAIL**, to see what causes the failure, view the entries in the other **Software Quality Objectives** columns. The entries which cause the failure are marked red.

If the ⚠ icon appears next to the status, it means that Polyspace does not have sufficient information to compute the status. For instance, if you specify the level `SQO-1`, but do not check for coding rule violations in your project, Polyspace cannot determine whether your project satisfies all the objectives specified in `SQO-1`.

4. View further details for the entries which are marked red on the **Summary** tab. For example, if an entry on the **Code Metrics over Threshold** column is marked red, select it. You can see values of the code complexity metrics on the **Code Metrics** tab.

5. Review each code complexity metric, coding rule violation, or run-time error because of which your project fails quality objectives. Fix your code or justify the errors or violations.

Tab	Action
Code Metrics	Note which entries are red. Investigate in your source code why those code complexity metrics exceed threshold. Fix your code to avoid exceeding the threshold. For more information, see “Code Metrics”.
Coding Rules	In the **Justified** column, note which entries are red. Select each entry to download the coding rule violations to the Polyspace user interface. Review the violations and fix or justify them. If you justify a violation, you can upload your justifications to Polyspace Metrics.
Run-Time Checks	In the **Justified** columns, note which entries are red. Select each entry to download the checks to the Polyspace user interface. Review the checks and fix or justify them. If you justify a check, you can upload your justifications to Polyspace Metrics.

For more information on the review process, see “Review Metrics for Particular Project or Run”.

6 After your review is over, in the Polyspace Metrics interface, click Refresh to view the updated metrics. See if your project has an **Overall Status** of **PASS** because of your justifications.

In addition to justifying some results, if you change your code, to update the metrics, rerun your verification and upload the results to the Polyspace Metrics repository. If you have justifications in your previous results, import them to the new results before uploading them to the repository. See “Import Review Comments from Previous Verifications”.

Tip You can apply a quality objective to all files in a project or run. If you want to turn off quality objectives or apply different objectives for some files in your project, you can place them in a separate module. To create a new module:

1. **Ctrl**-select the rows containing the files that you want to group.
2. Right-click the selection. Select **Add to Module**.
   All the files in the selection appear in your new module.
3. In the **Level** column for this module, select your quality objective from the drop-down list.
   The software applies this objective to all files in the module and determines an **Overall Status** of **PASS** or **FAIL** to the module.

Customize Software Quality Objectives

Instead of using a predefined objective, you can also define your own quality objectives and apply them to your project or module.

1. Save the following content in an XML file. Name the file `Custom-SQO-Definitions.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<MetricsDefinitions>
    <!-- Copyright 2010-2014 The MathWorks, Inc. -->

    <SQO ID="Custom_SQO_Level" ApplicableProduct="Code Prover">
        <comf>20</comf>
        <path>80</path>
        <goto>0</goto>
        <vg>10</vg>
        <calling>5</calling>
        <calls>7</calls>
        <param>5</param>
        <stmt>50</stmt>
        <level>4</level>
        <return>1</return>
        <vocf>4</vocf>
        <ap_cg_cycle>0</ap_cg_cycle>
        <ap_cg_direct_cycle>0</ap_cg_direct_cycle>
        <Num_Unjustified_Violations>Custom_MISRA_Rules_Set
</Num_Unjustified_Violations>
        <Num_Unjustified_Red>0</Num_Unjustified_Red>
        <Num_Unjustified_NT_Constructs>0
</Num_Unjustified_NT_Constructs>
        <Num_Unjustified_Gray>0</Num_Unjustified_Gray>
        <Percentage_Proven_Or_Justified>
Custom_Runtime_Checks_Set</Percentage_Proven_Or_Justified>
    </SQO>

    <CodingRulesSet ID="Custom_MISRA_Rules_Set">
        <Rule Name="MISRA_C_5_2">0</Rule>
        <Rule Name="MISRA_C_17_6">0</Rule>
    </CodingRulesSet>

    <RuntimeChecksSet ID="Custom_Runtime_Checks_Set">
        <Check Name="OBAI">80</Check>
        <Check Name="IDP">60</Check>
    </RuntimeChecksSet>

</MetricsDefinitions>
```

2 Save this XML file in the folder where remote analysis data is stored, for example, `C:\Users\JohnDoe\AppData\Roaming\Polyspace_RLDatas`.

   If you want to change the folder location, select **Metrics** > **Metrics and Remote Server Settings**.

3 Modify the content of this file to specify your own quality thresholds. For more information, see “Elements in Custom Software Quality Objectives File”.

a For specifying coding rules, begin the rule name with the appropriate string followed by the rule number. Use _ instead of a decimal point in the rule number.

Rule	String	Rule numbers
MISRA C	`MISRA_C_`	"MISRA C:2004 Coding Rules"
MISRA C++	`MISRA_Cpp_`	"MISRA C++ Coding Rules"
JSF C++	`JSF_Cpp_`	"JSF C++ Coding Rules"
Custom coding rules	`Custom_`	"Custom Coding Rules"

b For specifying checks, use the appropriate check acronym. For more information, see "Check and Code Metric Acronyms".

4. After you have made your modifications, in the Polyspace Metrics interface, open the metrics for your project. From the **Quality Objectives** list in the upper left, select `ON`.
5. On the **Summary** tab, select an entry in the **Level** column. Your new quality objective appears in the drop-down list.
6. Select this objective.

   The software compares the thresholds you had specified against your results and updates the **Overall Status** column with **PASS** or **FAIL**.

Related Examples

- "View Trends in Code Quality Metrics"

View Trends in Code Quality Metrics

Using the Polyspace Metrics interface, you can track improvements or regression in code quality metrics over various runs on the same source code.

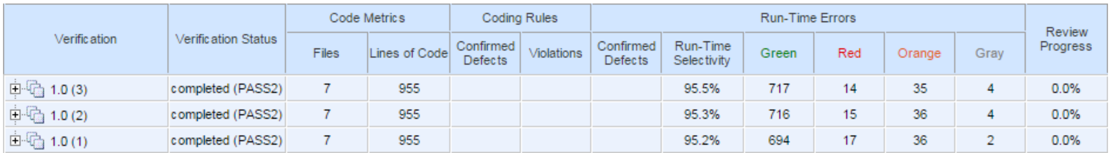
To view trends in metrics, upload the various versions of your results to the Polyspace Metrics repository.

1 Open the Polyspace Metrics interface.

   For more information, see "Open Metrics Interface".

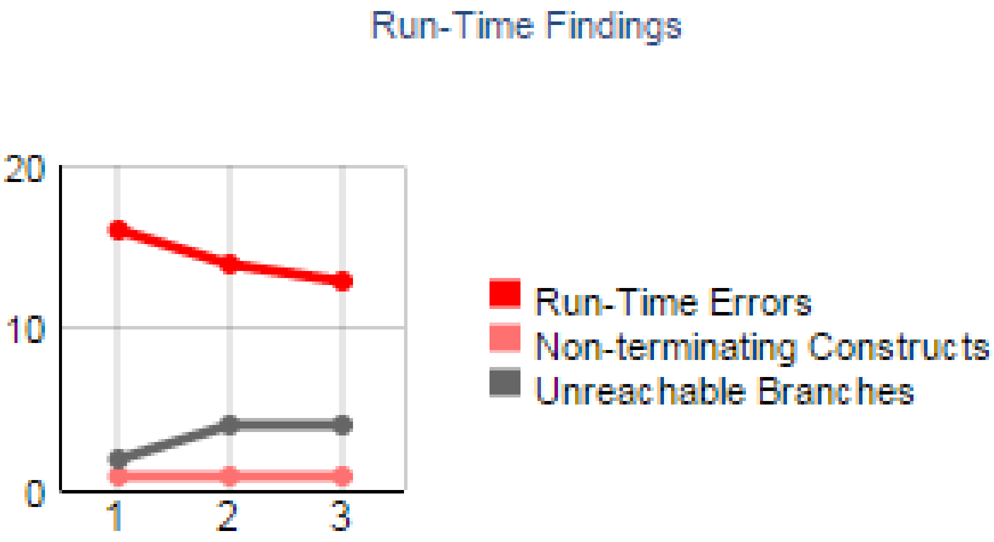
2 On the **Projects** tab, select the project for which you want to view trends.

   The code quality metrics for all versions of the project appear on the **Summary**, **Code Metrics**, **Coding Rules**, and **Run-Time Checks** tabs. For example, the figure shows the **Summary** tab displaying three versions of a project.



Verification	Verification Status	Code Metrics		Coding Rules		Run-Time Errors						Review Progress
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Run-Time Selectivity	Green	Red	Orange	Gray	
⊞ 1.0 (3)	completed (PASS2)	7	955				95.5%	717	14	35	4	0.0%
⊞ 1.0 (2)	completed (PASS2)	7	955				95.3%	716	15	36	4	0.0%
⊞ 1.0 (1)	completed (PASS2)	7	955				95.2%	694	17	36	2	0.0%

   In addition, you can see a graphical view of the trends on each tab. For example, the figure shows the trend in **Run-Time Findings** over three versions of a project.



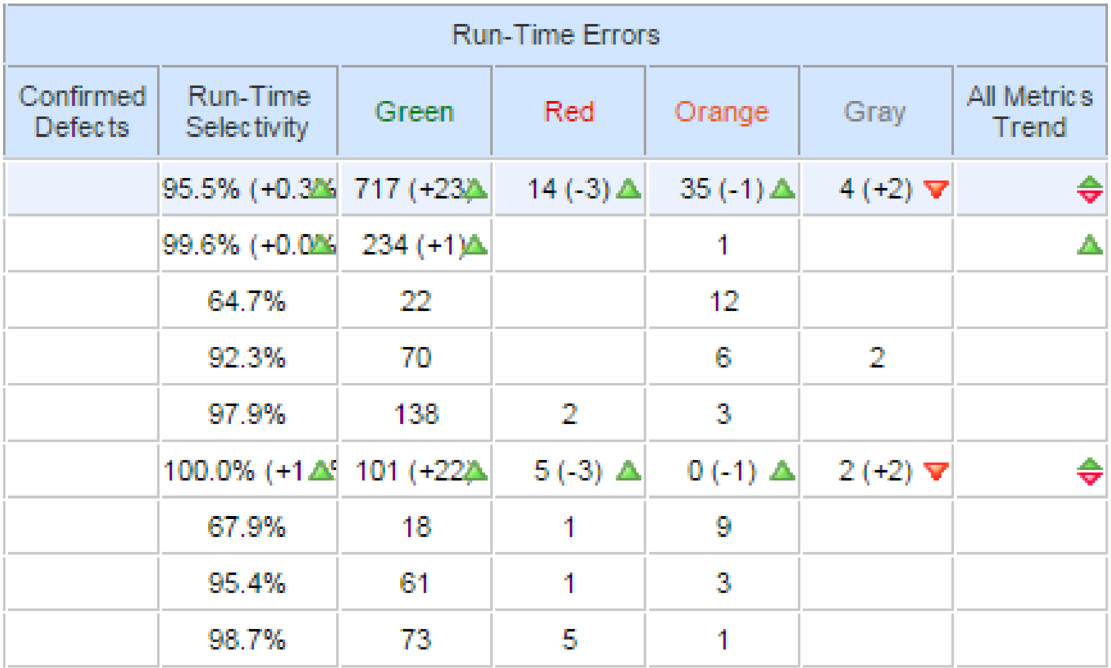
3 To compare two versions of the same project:

a In the **From** and **To** lists on the upper left of the web dashboard, select the two versions that you want to compare.

b Select the **Compare** box.

On each tab, new columns appear and existing columns display improvement or regression in a metric. For example, in the figure below, you see a new **All Metrics Trend** column that appears on the **Summary** tab. This column describes how the metrics in the **Run-Time Errors** group compare over two versions of a project. The number of red checks decreased by 3 and the number of gray checks increased by 2. Because the decrease in red checks is an improvement and the increase in gray checks is a regression, you see:

- A ▲ in the **Red** column
- A ▼ in the **Gray** column
- A mixed ▲▼ in the **All Metrics Trend** column.



Run-Time Errors						
Confirmed Defects	Run-Time Selectivity	Green	Red	Orange	Gray	All Metrics Trend
	95.5% (+0.3▲	717 (+23▲	14 (-3) ▲	35 (-1) ▲	4 (+2) ▼	▲▼
	99.6% (+0.0▲	234 (+1)▲		1		▲
	64.7%	22		12		
	92.3%	70		6	2	
	97.9%	138	2	3		
	100.0% (+1▲	101 (+22▲	5 (-3) ▲	0 (-1) ▲	2 (+2) ▼	▲▼
	67.9%	18	1	9		
	95.4%	61	1	3		
	98.7%	73	5	1		

4 To see only the new findings in a version compared to a previous version:

a In the **From** and **To** lists on the upper left of the web dashboard, select the two versions that you want to compare.

b Select the **New Findings Only** box.

The existing columns display only the new findings. In addition, you also see two new columns:

- The **Newly Confirmed** column shows those new findings to which you assign a **Classification** of `High`, `Medium`, or `Low` in the Polyspace user interface.
- The **Newly Fixed** column shows those findings to which you had assigned a **Classification** of `High`, `Medium` or `Low` in the previous run. However, the assignment does not exist in the current run, either because a red or orange check turned green, or because you changed the **Classification** to `Not a defect`.

Related Examples

- “Code Quality Metrics”

Web Browser Requirements for Viewing Metrics

Polyspace Metrics supports the following web browsers:

- Internet Explorer® version 7.0, or later
- Firefox® version 3.6, or later
- Google® Chrome version 12.0, or later
- Safari for Mac version 6.1.4 and 7.0.4

To use Polyspace Metrics, install Java, version 1.4 or later on your computer.

For the Firefox web browser, manually install the required Java plug-in. For example, if your computer uses the Linux operating system:

1. Create a Firefox folder for plug-ins:

   ```
   mkdir ~/.mozilla/plugins
   ```

2. Go to this folder:

   ```
   cd ~/.mozilla/plugins
   ```

3. Create a symbolic link to the Java plug-in, which is available in the Java Runtime Environment folder of your MATLAB installation:

   ```
   ln -s MATLAB_Install/sys/java/jre/glnxa64/jre/lib/amd64/libnpjp2.so
   ```

Predefined Software Quality Objectives

The Software Quality Objectives or SQOs are a set of thresholds that generate a **Quality Status** of **PASS** or **FAIL** for your verification results. You can use a predefined SQO level or define your own SQOs. Following are the quality thresholds specified by each predefined SQO.

SQO Level 1

Metric	Threshold Value
Comment density of a file	20
Number of paths through a function	80
Number of `goto` statements	0
Cyclomatic complexity	10
Number of calling functions	5
Number of calls	7
Number of parameters per function	5
Number of instructions per function	50
Number of call levels in a function	4
Number of `return` statements in a function	1
Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: `(N1+N2) / (n1+n2)` • *n1* — Number of different operators • *N1* — Total number of operators • *n2* — Number of different operands • *N2* — Total number of operands	4
Number of recursions	0
Number of direct recursions	0
Number of unjustified violations of the following MISRA C:2004 rules:	0

Metric	Threshold Value
• 5.2 • 8.11, 8.12 • 11.2, 11.3 • 12.12 • 13.3, 13.4, 13.5 • 14.4, 14.7 • 16.1, 16.2, 16.7 • 17.3, 17.4, 17.5, 17.6 • 18.4 • 20.4	
Number of unjustified violations of the following MISRA C:2012 rules: • 8.8, 8.11, and 8.13 • 11.1, 11.2, 11.4, 11.5, 11.6, and 11.7 • 14.1 and 14.2 • 15.1, 15.2, 15.3, and 15.5 • 17.1 and 17.2 • 18.3, 18.4, 18.5, and 18.6 • 19.2 • 21.3	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C++ rules: • 2-10-2 • 3-1-3, 3-3-2, 3-9-3 • 5-0-15, 5-0-18, 5-0-19, 5-2-8, 5-2-9 • 6-2-2, 6-5-1, 6-5-2, 6-5-3, 6-5-4, 6-6-1, 6-6-2, 6-6-4, 6-6-5 • 7-5-1, 7-5-2, 7-5-4 • 8-4-1 • 9-5-1 • 10-1-2, 10-1-3, 10-3-1, 10-3-2, 10-3-3 • 15-0-3, 15-1-3, 15-3-3, 15-3-5, 15-3-6, 15-3-7, 15-4-1, 15-5-1, 15-5-2 • 18-4-1	0

SQO Level 2

In addition to all the requirements of SQO Level 1, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified red checks	0
Number of unjustified Non-terminating call and Non-terminating loop checks	0

SQO Level 3

In addition to all the requirements of SQO Level 2, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified gray Unreachable code checks	0

SQO Level 4

In addition to all the requirements of SQO Level 3, this level includes the following thresholds:

Metric	Threshold Value
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	C++ specific checks: 50
	Correctness condition: 60
	Division by zero: 80
	Exception handling: 50
	Function returns a value: 80
	Illegally dereferenced pointer: 60
	Initialized return value: 80
	Non-initialized local variable: 80
	Non-initialized pointer: 60
	Non-initialized variable: 60
	Non-null this-pointer in method: 50
	Object oriented programming: 50
	Out of bounds array index: 80
	Overflow: 60
	Shift operations: 80
	User assertion: 60

SQO Level 5

In addition to all the requirements of SQO Level 4, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: • 6.3 • 8.7	0

Metric	Threshold Value
• 9.2, 9.3 • 10.3, 10.5 • 11.1, 11.5 • 12.1, 12.2, 12.5, 12.6, 12.9, 12.10 • 13.1, 13.2, 13.6 • 14.8, 14.10 • 15.3 • 16.3, 16.8, 16.9 • 19.4, 19.9, 19.10, 19.11, 19.12 • 20.3	
Number of unjustified violations of the following MISRA C:2012 rules: • 11.8 • 12.1 and 12.3 • 13.2 and 13.4 • 14.4 • 15.6 and 15.7 • 16.4 and 16.5 • 17.4 • 20.4, 20.6, 20.7, 20.9, and 20.11	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C++ rules: • 3-4-1, 3-9-2 • 4-5-1 • 5-0-1, 5-0-2, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-0-13, 5-2-1, 5-2-2, 5-2-7, 5-2-11, 5-3-3, 5-2-5, 5-2-6, 5-3-2, 5-18-1 • 6-2-1, 6-3-1, 6-4-2, 6-4-6, 6-5-3 • 8-4-3, 8-4-4, 8-5-2, 8-5-3 • 11-0-1 • 12-1-1, 12-8-2 • 16-0-5, 16-0-6, 16-0-7, 16-2-2, 16-3-1	0
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	C++ specific checks: 70
	Correctness condition: 80
	Division by zero: 90
	Exception handling: 70
	Function returns a value: 90
	Illegally dereferenced pointer: 70
	Initialized return value: 90
	Non-initialized local variable: 90
	Non-initialized pointer: 70
	Non-initialized variable: 70
	Non-null this-pointer in method: 70
	Object oriented programming: 70
	Out of bounds array index: 90
	Overflow: 80
	Shift operations: 90
	User assertion: 80

SQO Level 6

In addition to all the requirements of SQO Level 5, this level includes the following thresholds:

Metric	Threshold Value
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	C++ specific checks: 90
	Correctness condition: 100
	Division by zero: 100
	Exception handling: 90
	Function returns a value: 100
	Illegally dereferenced pointer: 80
	Initialized return value: 100
	Non-initialized local variable: 100
	Non-initialized pointer: 80
	Non-initialized variable: 80
	Non-null this-pointer in method: 90
	Object oriented programming: 90
	Out of bounds array index: 100
	Overflow: 100
	Shift operations: 100
	User assertion: 100

SQO Exhaustive

In addition to all the requirements of SQO Level 1, this level includes the following thresholds. The thresholds for coding rule violations apply only if you check for coding rule violations.

Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified red checks	0

Metric	Threshold Value
Number of unjustified Non-terminating call and Non-terminating loop checks	0
Number of unjustified gray Unreachable code checks	0
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	100

For information on the rationales behind these levels, see Software Quality Objectives for Source Code.

Related Examples

- "Compare Metrics Against Software Quality Objectives"

Elements in Custom Software Quality Objectives File

The following tables list the XML elements that can be added to the custom SQO file. The content of each element specifies a threshold against which the software compares verification results. For each element, the table lists the metric to which the threshold applies. Here, HIS refers to the Hersteller Initiative Software.

For information on custom SQOs, see "Customize Software Quality Objectives".

In this section...

HIS Metrics

Element	Metric
`comf`	Comment density of a file
`path`	Number of paths through a function
`goto`	Number of `goto` statements
`vg`	Cyclomatic complexity
`calling`	Number of calling functions
`calls`	Number of calls
`param`	Number of parameters per function
`stmt`	Number of instructions per function
`level`	Number of call levels in a function
`return`	Number of `return` statements in a function
`vocf`	Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: `(N1+N2) / (n1+n2)` • *n1* — Number of different operators • *N1* — Total number of operators

Element	Metric

	• *n2* — Number of different operands
	• *N2* — Total number of operands
	The computation is based on the preprocessed source code. Consider the following code.
	`int f(int i)`
	`  {`
	`    if (i == 1)`
	`      return i;`
	`    else`
	`      return i * g(i-1);`
	`  }`
	The code contains the following:
	• Distinct operators — `int`, `(`, `)`, `{`, `if`, `==`, `return`, `else`, `*`, `-`, `;`, and `}`
	• Number of operators —12
	• Number of operator occurrences —19
	• Distinct operands — `f`, `i`, `1`, and `g`
	• Number of operands — 4
	• Number of operand occurrences — 9
	Therefore, the language scope for the code is `VOCF = (19 + 9) / (12 + 4)`, that is, `1.8`.

`ap_cg_cycle`	Number of recursions
`ap_cg_direct_cycle`	Number of direct recursions
`Num_Unjustified_Violations`	Number of unjustified violations of MISRA C rules specified by entries under the element `CodingRulesSet`
`Num_Unjustified_Red`	Number of unjustified red checks
`Num_Unjustified_NT_Constructs`	Number of unjustified Non-terminating call and Non-terminating loop checks
`Num_Unjustified_Gray`	Number of unjustified gray Unreachable code checks

Element	Metric
`Percentage_Proven_Or_Justified`	Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.

Non-HIS Metrics

Element	Description of metric
`fco`	Estimated function coupling, which is calculated as follows: `SOC - DFF + 1` • `SOC` — Sum (over all file functions) of calls within body of each function • `DFF` — Number of defined file functions Does not take into account member functions of a template class or template functions. Computed metric reflects coupling of non-template functions only.
`flin`	Number of lines within function body
`fxln`	Number of execution lines within function body A variable declaration with initialization is treated as a statement, but not as an execution line of function body.
`ncalls`	Number of calls within function body Includes explicit and implicit calls to constructors.
`pshv`	Number of protected shared variables
`unpshv`	Number of unprotected shared variables

Configure Model for Code Analysis

Configure Simulink Model

Before analyzing your generated code, there are certain settings that you should apply to your model. Use the following workflow to prepare your model for code analysis.

- If you know of results ahead of time, annotate your blocks with Polyspace annotations.
- Set the recommended configuration parameters.
- Double-check your model settings.
- Generate code.
- Set up your Polyspace options.

Recommended Model Settings for Code Analysis

For Polyspace analyses, set the following parameter configurations before generating code.

Grouping	Parameter	Recommended value	Name and Location in Configuration	If you do not use recommendation
Code Generation	`SystemTargetF`	An Embedded Coder Target Language Compiler (TLC) file. For example `ert.tlc` or `autosar.tlc`	Location: **Code Generation** Name: **System target file** Value: Embedded Coder target file	Error
	`MatFileLoggin`	`'off'`	Location: **Code Generation** > **Interface** Name: **MAT-file logging** Value: ☐ Not selected	Warning
	`GenerateRepor`	`'on'`	Location: **Code Generation** > **Report** Name: **Create code-generation report** Value: ☑ Selected	Error
	`IncludeHyperl`	`'on'`	Location: **Code Generation** > **Report** Name: **Code-to-model** Value: ☑ Selected	Error
	`GenerateSampl`	`'off'`	Location: **Code Generation** > **Templates** Name: **Generate an example main program**	Warning

Grouping	Parameter	Recommended value	Name and Location in Configuration	If you do not use recommendation
			Value: □ Not selected	
	`GenerateComme`	`'on'`	Location: **Code Generation > Comments** Name: **Include comments** Value: ☑ Selected	Warning
Optimization	`InlineParams`	`'on'`	Location: **Optimization > Signals and Parameters** Name: **Inline parameters** Value: ☑ Selected	Warning
	`InitFltsAndDb`	`'on'`	Location: **Optimization** Name: **Use memset to initialize floats and doubles to 0.0** Value: □ Not selected	Warning
	`ZeroExternalM`	`'on'` when **Configuratio Parameters > Polyspace > Data Range Management > Output** is `Global assert`	Location: **Optimization** Name: **Remove root level I/O zero initialization** Value: □ Not selected	Warning
Solver	`SolverType`	`'Fixed-Step'`	Location: **Solver** Name: **Type** Value: `Fixed-step`	Warning

Grouping	Parameter	Recommended value	Name and Location in Configuration	If you do not use recommendation
	`Solver`	`'FixedStepD`	Location: **Solver** Name: **Solver** Value: `discrete (no continuous states)`	Warning

Check Simulink Model Settings

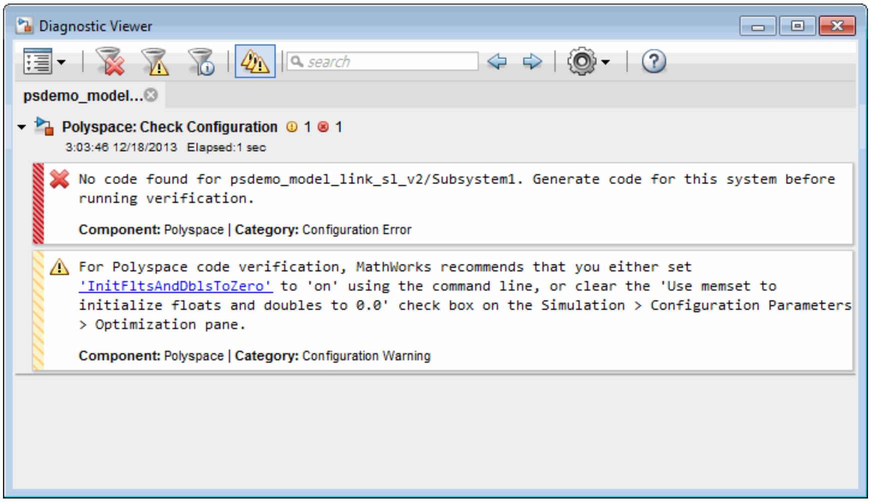
With the Polyspace plug-in, you can check your model settings before generating code or before starting an analysis. If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, warnings appear when you run the analysis.

Check Simulink Model Settings Using the Code Generation Advisor

Before generating code, you can check your model settings against the “Recommended Model Settings for Code Analysis” on page 15-3. If you do not use the recommended model settings, the back-to-model linking will not work correctly.

1. From the Simulink model window, select **Code** > **C/C++ Code** > **Code Generation Options**. The Configuration Parameters dialog box opens, displaying the **Code Generation** pane.
2. Select **Set Objectives**.
3. From the **Set Objective – Code Generation Advisor** window, add the `Polyspace` objective and any others that you want to check.
4. In the **Check model before generating code** drop-down list, select either:
   - `On (stop for warnings)`, the process stops for either errors or warnings without generating code.
   - `On (proceed with warnings)`, the process stops for errors, but continues generating code if the configuration only has warnings.
5. Select **Check Model**.

   The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

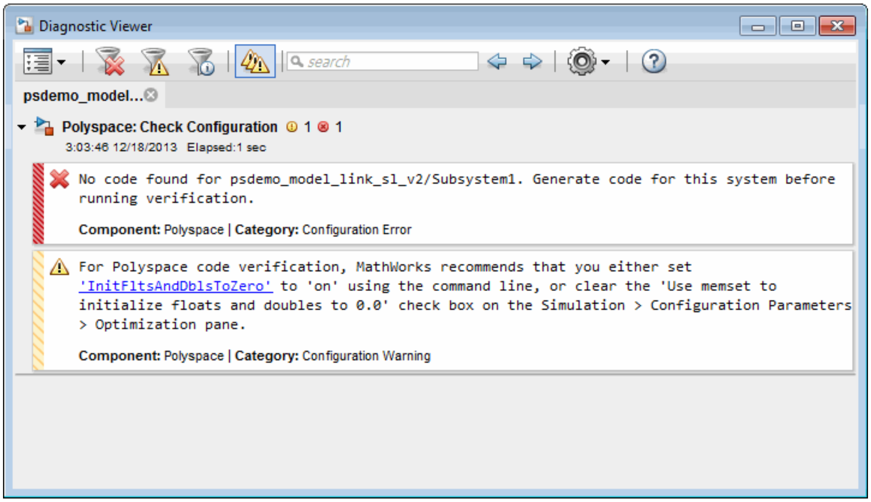
```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

Check Simulink Model Settings Before Analysis

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.



Polyspace: Check Configuration 1 1

3:03:46 12/18/2013 Elapsed:1 sec

No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before running verification.

Component: Polyspace | **Category:** Configuration Error

For Polyspace code verification, MathWorks recommends that you either set 'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters > Optimization pane.

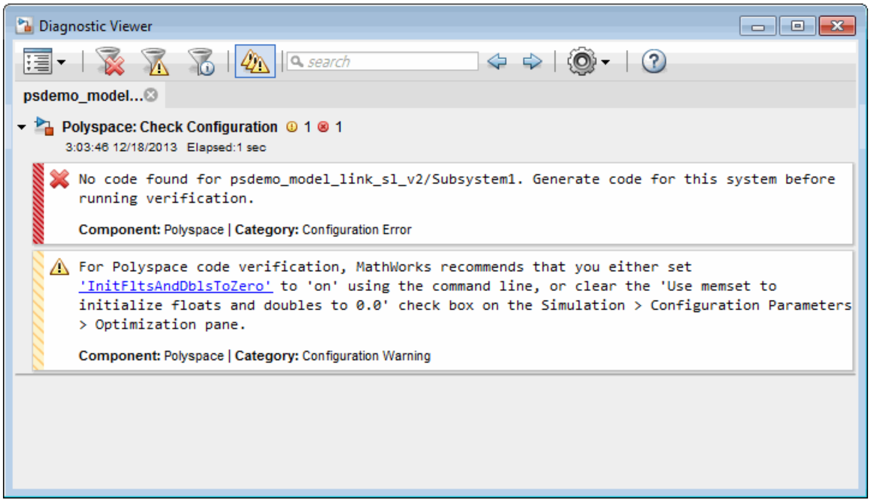
Component: Polyspace | **Category:** Configuration Warning

3 From the **Check configuration before verification** menu, select either:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)`, the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

4 Select **Run verification**.

The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration ⚠ 1 ⊗ 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

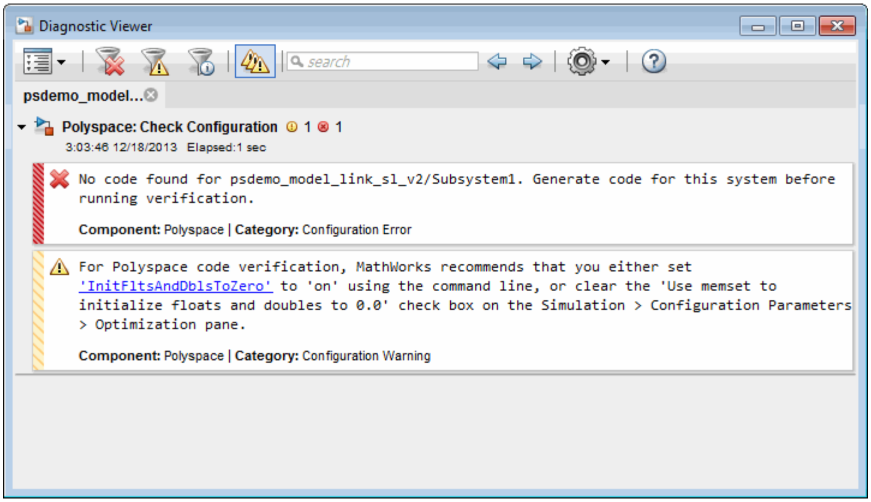
Component: Polyspace | **Category:** Configuration Warning

If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

Check Simulink Model Settings Automatically

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.



Polyspace: Check Configuration ⚠ 1 ⊗ 1
3:03:46 12/18/2013 Elapsed:1 sec

✖ No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before running verification.

Component: Polyspace | **Category:** Configuration Error

⚠ For Polyspace code verification, MathWorks recommends that you either set 'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters > Optimization pane.

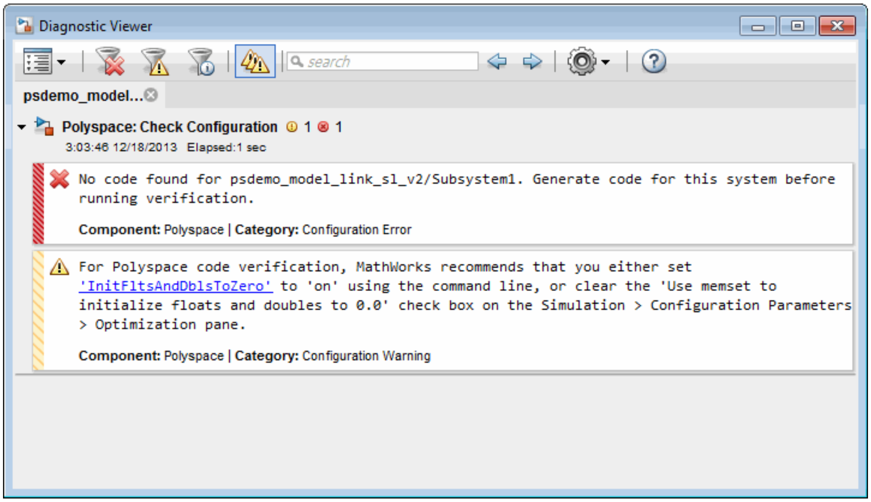
Component: Polyspace | **Category:** Configuration Warning

3 From the **Check configuration before verification** menu, select either:

- `On (stop for warnings)` — will
- `On (proceed with warnings)`

4 Select **Run verification**.

The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

If you select:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)` — the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

More About

- “Recommended Model Settings for Code Analysis” on page 15-3

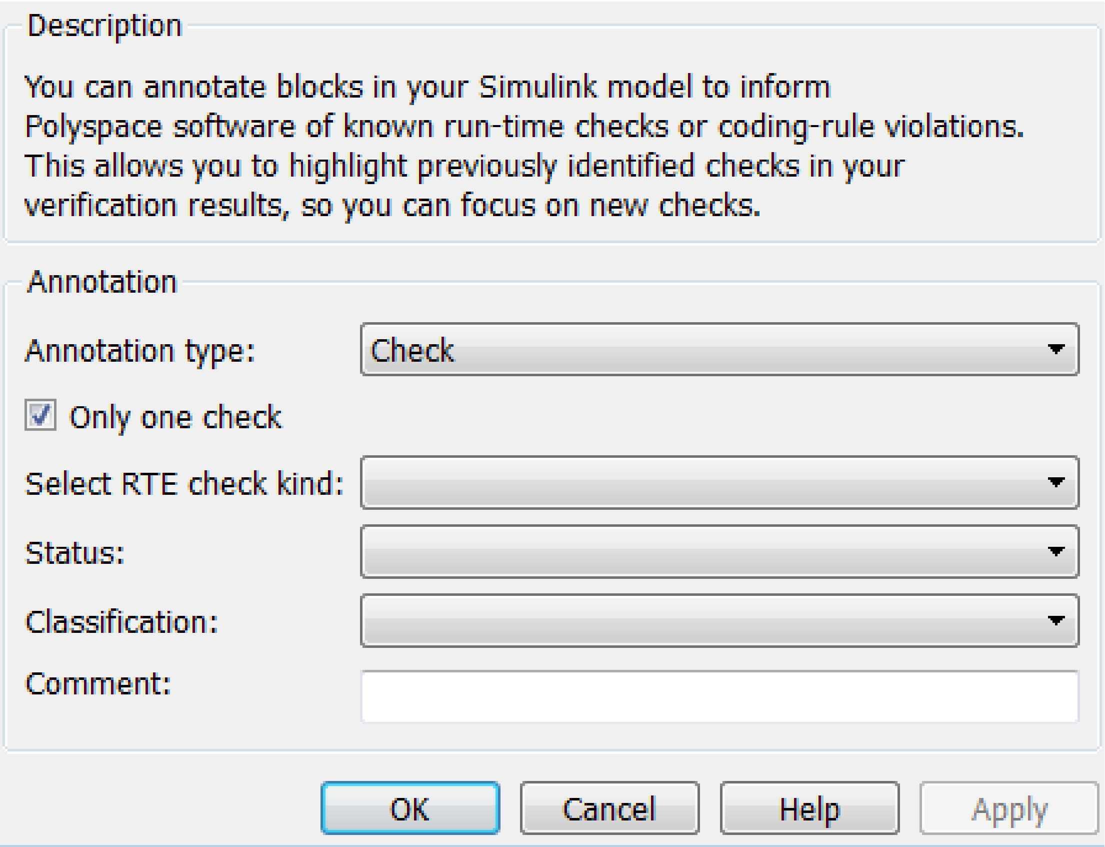
Annotate Blocks for Known Results

You can annotate individual blocks in your Simulink model to inform Polyspace software of known defects, run-time checks, or coding-rule violations. This allows you to highlight and categorize previously identified results, so you can focus on reviewing new results.

Your Polyspace results displays the information that you provide with block annotations. To annotate blocks:

1 In the Simulink model window, right-click the block you want to annotate.

2 From the context menu, select **Polyspace** > **Annotate Selected Block** > **Edit**. The Polyspace Annotation dialog box opens.



Description

You can annotate blocks in your Simulink model to inform Polyspace software of known run-time checks or coding-rule violations. This allows you to highlight previously identified checks in your verification results, so you can focus on new checks.

Annotation

Annotation type: Check

☑ Only one check

Select RTE check kind:

Status:

Classification:

Comment:

3. From the **Annotation type** drop-down list, select one of the following:

   - `Check` — To indicate a Code Prover run-time error
   - `Defect` — To indicate a Bug Finder defect
   - `MISRA-C` — To indicate a MISRA C coding rule violation
   - `MISRA-C++` — To indicate a MISRA C++ coding rule violation
   - `JSF` — To indicate a JSF C++ coding rule violation

4. If you want to highlight only one kind of result, select **Only 1 check** and the relevant error or coding rule from the **Select RTE check kind** (or **Select defect kind**, **Select MISRA rule**, **Select MISRA C++ rule**, or **Select JSF rule**) drop-down list.

   If you want to highlight a list of checks, clear **Only 1 check**. In the **Enter a list of checks** (or **Enter a list of defects**, or **Enter a list of rule numbers**) field, specify the errors or rules that you want to highlight.

5. Select a **Status** to describe how you intend to address the issue:

   - `Fix`
   - `Improve`
   - `Investigate`
   - `Justify with annotations`

     (This status also marks the result as justified.)
   - `No action planned`

     (This status also marks the result as justified.)
   - `Other`
   - `Restart with different options`
   - `Undecided`

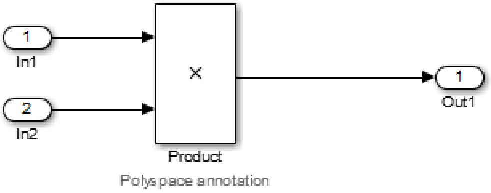
6. Select a **Classification** to describe the severity of the issue:

   - `High`
   - `Medium`
   - `Low`

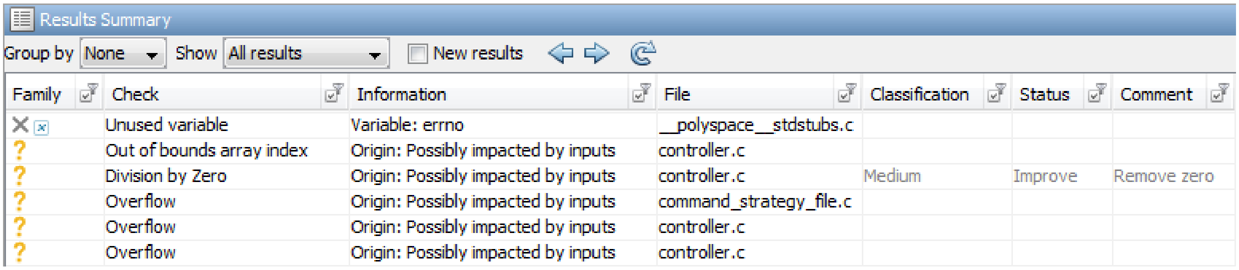
- `Not a defect`

7 In the **Comment** field, enter additional information about the check.

8 Click **OK**. The software adds the Polyspace annotation is to the block.



When you run an analysis, the **Results Summary** pane pre-populates the results with your annotation.



Group by None Show All results □ New results

Family	Check	Information	File	Classification	Status	Comment
✕ ☒	Unused variable	Variable: errno	__polyspace__stdstubs.c			
?	Out of bounds array index	Origin: Possibly impacted by inputs	controller.c			
?	Division by Zero	Origin: Possibly impacted by inputs	controller.c	Medium	Improve	Remove zero
?	Overflow	Origin: Possibly impacted by inputs	command_strategy_file.c			
?	Overflow	Origin: Possibly impacted by inputs	controller.c			
?	Overflow	Origin: Possibly impacted by inputs	controller.c			

See Also

`pslinkfun`

16

Model Link for Polyspace Code Prover

Install Polyspace Plug-In for Simulink

By default, when you install Polyspace R2013b or later, the Simulink plug-in is installed and connected to MATLAB.

If you model on a previous version of Simulink and MATLAB, you can also connect the Polyspace plug-in on this previous version. That way you use the latest verification software with your preferred version of Embedded Coder or TargetLink®. The Simulink plug-in supports the four previous releases of MATLAB. For example, the R2015a version of the Polyspace plug-in supports MATLAB R2013a, R2013b, R2014a,R2014b, and R2015a.

However, if you use a cross-version of Polyspace and MATLAB, local batch analyses can only be submitted from the Polyspace environment. or using the `pslinkrun` command.

Note: To install a newer version of Polyspace on MATLAB R2013b or later, you must install MATLAB without the corresponding version of Polyspace.

---

1. Using an account with read/write privileges, open the older version of MATLAB.
2. If you have a previous version of Polyspace connected, execute the `pslinksetup('uninstall')` command to disconnect it. This command does not work with MATLAB R2013b or later (see preceding Note).
3. Restart MATLAB.
4. Change your **Current Folder** to `*matlabroot*\polyspace\toolbox\pslink\pslink`. *`matlabroot`* is the Simulink plug-in that you want to connect, for example, `C:\Program Files\MATLAB\R2015a`.
5. Execute the `pslinksetup('install')` command to connect the new version of Polyspace.

Related Examples

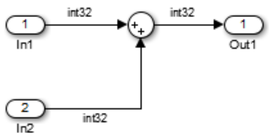
- "Verify Code from a Simple Simulink Model"

More About

- "Troubleshoot Back to Model"

Annotate Code to Justify Polyspace Checks

A verification of Embedded Coder generated code might highlight overflows for certain operations that are legitimate because of the way Embedded Coder implements these operations. Consider the following model and the corresponding generated code.



```
32 /* Sum: '<Root>/Sum' incorporates:
33   *  Inport: '<Root>/In1'
34   *  Inport: '<Root>/In2'
35   */
36  qY_0 = sat_add_U.In1 + sat_add_U.In2;
37  if ((sat_add_U.In1 < 0) && ((sat_add_U.In2 < 0) && (qY_0 >= 0))) {
38    qY_0 = MIN_int32_T;
39  } else {
40   if ((sat_add_U.In1 > 0) && ((sat_add_U.In2 > 0) && (qY_0 <= 0))) {
41      qY_0 = MAX_int32_T;
42    }
43  }
```

Embedded Coder software recognizes that the largest built-in data type is 32-bit. It is not possible to saturate the results of the additions and subtractions using `MIN_INT32` and `MAX_INT32`, and a bigger single-word integer data type. Instead the software detects the results overflow and the direction of the overflow, and saturates the result.

If you do not provide justification for the addition operator on line 36, a Polyspace verification generates an orange check that indicates a potential overflow. The verification does not take into account the saturation function of lines 37 to 43. In addition, the trace-back functionality of Polyspace Code Prover does not identify the reason for the orange check.

To justify overflows from operators that are legitimate, on the **Configuration Parameters** > **Code Generation** > **Comments** pane:

- Under **Overall control**, select the **Include comments** check box.
- Under **Auto generate comments**, select the **Operator annotations check box**.

When you generate code, the Embedded Coder software annotates the code with comments for Polyspace. For example:

```
/* Sum: '<Root>/Sum' incorporates:
  *  Inport: '<Root>/In1'
  *  Inport: '<Root>/In2'
  */
 qY_0 = sat_add_U.In1 +/*MW:OvOk*/ sat_add_U.In2;
```

When you run a verification using Polyspace Code Prover, the software uses the annotations to justify the operator-related orange checks and assigns the `Not a defect` classification to the checks.

Main Generation for Model Verification

When you run a verification, the software automatically reads the following information from the model:

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

The software then uses this information to generate a `main` function that:

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

If the `codeInfo` for the model does not contain the names of the inputs, the software considers all variables as entries, except for parameters and outputs.

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

- For class methods in the generated code, the variables that are written before and in the loop refer to the class members.
- For functions with global scope, the associated variables are also in the global scope.

`main` for Generated Code

The following example shows the `main` generator options that the software uses to generate the `main` function for code generated from a Simulink model.

```
init parameters    \\ -variables-written-before-loop
init_fct()          \\ -functions-called-before-loop
```

```
  while(1){        \\ start main loop
  init inputs      \\ -variables-written-in-loop
  step_fct()       \\ -functions-called-in-loop
}
terminate_fct()    \\ -functions-called-after-loop
```

Configure Data Range Settings

There are two approaches to code verification, which can produce results that are slightly different:

- **Contextual Verification** — Prove code does not generate run-time errors under predefined working conditions. This limits the scope of the verification to specific variable ranges, and verifies the code within these ranges.
- **Robustness Verification —** Prove code generate run-time errors for all verification conditions, including “abnormal” conditions for which the code was not designed. This can be thought of as “worst case” verification.

You perform contextual or robustness verification by the way you specify data ranges for model inputs, outputs, and tunable parameters within the model.

To specify data range settings for your model:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. In the Data Range Management section, specify how you want the verification to treat:

   a. **Input** — Select one of the following:

      - `Use specified minimum and maximum values` (Default) — Apply data ranges defined in blocks or base workspace to increase the precision of the verification. See “Specify Signal Ranges” on page 17-18.
      - `Unbounded inputs` — Assume all inputs are full-range values (`min...max`)

   b. **Tunable parameters** — Select one of the following:

      - `Use calibration data` (Default) — Use value of constant parameter specified in code.
      - `Use specified minimum and maximum values` — Use a parameter range defined in the block or base workspace. See “Specify Signal Ranges” on page 17-18. If no range is defined, use full range (`min...max`).

   c. **Output** — Select one of the following:

      - `No verification` (Default) — No assertion ranges on outputs.
      - `Verify outputs are within minimum and maximum values` — Use assertion ranges on outputs.

Note: This mode is incompatible with the Automatic Orange Tester.

In general, you should use the following combinations:

- To maximize verification precision, select `Use specified minimum and maximum values` for **Input** and **Tunable parameters**.
- To verify the extreme cases of program execution, select `Unbounded inputs` for **Input** and `Use calibration data` for **Tunable parameters**.

Related Examples

- "Specify Signal Ranges" on page 17-18

Embedded Coder Considerations

In this section...

Default Options

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

```
-sources path_to_source_code
-desktop
-D PST_ERRNO
-D main=main_rtwec
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-OS-target no-predefined-OS
-results-dir results
```

Note: *matlabroot* is the MATLAB installation folder.

Data Range Specification

You can constrain inputs, parameters, and outputs to lie within specified data ranges for Embedded Coder and AUTOSAR with Embedded Coder. See "Configure Data Range Settings" on page 16-7.

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

You can also manually define a DRS file in the Polyspace user interface. If you define a DRS file, the software appends the automatically generated information to the DRS file you create. Manually defined DRS information overrides automatically generated information for all variables.

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

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

The software supports the automatic generation of data range specifications for only the following signal and parameter storage classes:

- `SimulinkGlobal`
- `ExportedGlobal`
- `Struct (Custom)`

Recommended Polyspace options for Verifying Generated Code

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

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

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

You can specify other verification options for your Polyspace Project through the Polyspace **Configuration** pane. To open this pane:

1 In the Simulink model window, select **Code** > **Polyspace** > **Options** . The **Polyspace Model Link** pane opens.

2 Click **Configure**. The Polyspace user interface opens, displaying the Polyspace **Configuration** pane.

The following table describes options that you should specify in your Polyspace project before verifying code generated by Embedded Coder software.

Option	Recommende Value	Comments
Macros > **Preprocessor definitions** `-D`	See Comments	Defines macro compiler flags used during compilation. Use one `-D` for each line of the Embedded Coder generated `defines.txt` file. Polyspace Model Link™ SL does not do this by default.
Target & Compiler > **Target operating system** `-OS-target`	`Visual`	Specifies the operating system target for Polyspace stubs. This information allows the verification to use system definitions during preprocessing to analyze the included files.
Target & Compiler > **Target processor type** `-target`	`i386`	Specifies the target processor type. This allows the verification to consider the size of fundamental data types and the endianess of the target machine. You can configure and specify generic targets.
Environment Settings > **Code from DOS or Windows file system** `-dos`	On	You must select this option if the contents of the include or source directory comes from a DOS or Windows file system. The option allows the verification to deal with upper/lower case sensitivity and control characters issues. Concerned files are: • **Header files** – All include folders specified (`-I` option) • **Source files** – All source files selected for the verification (`-sources` option)

Option	Recommende Value	Comments
Check Behavior > Allow negative operands for left shifts `-allow-negative-operand-in-shift`	On	Allows a shift operation on a negative number. According to the ANSI standard, such a shift operation on a negative number is illegal. For example, -2 << 2 If you select this option, Polyspace considers the operation to be valid. For the given example, -2 << 2 = -8
Verification Assumptions > Ignore float rounding `-ignore-float-rounding`	On	Specifies how the verification rounds floats. If this option is not selected, the verification rounds floats according to the IEEE 754 standard – simple precision on 32-bits targets and double precision on targets that define double as 64-bits. When you select this option, the verification performs exact computation. Selecting this option can lead to results that differ from "real life," depending on the actual compiler and target. Some paths may be reachable (or not reachable) for the verification while they are not reachable (or are reachable) for the actual compiler and target. However, this option reduces the number of unproven checks caused by float approximation.

Option	Recommende Value	Comments
Precision > Precision level `-O`	2	Specifies the precision level for the verification. Higher precision levels provide higher selectivity at the expense of longer verification time. Begin with the lowest precision level. You can then address red errors and gray code before rerunning the Polyspace verification using higher precision levels. Benefits: A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate. The precision level specifies the algorithms used to model the program state space during verification: • `-O0` corresponds to static interval verification. • `-O1` corresponds to complex polyhedron model of domain values. • `-O2` corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons). • `-O3` is suitable only for units smaller than 1,000 lines of code. For such code, selectivity may reach as high as 98%, but verification may take up to an hour per 1,000 lines of code.

Option	Recommende Value	Comments
Precision > Verification level -to	See comments	Specifies the phase after which the verification stops. **C source compliance checking** – For C code, when checking coding rule compliance only. **C++ source compliance checking** – For C++ code, when checking coding rule compliance only. **Software safelty analysis level 0** – When verifying code for the first time. **Software safelty analysis level 4** – When performing subsequent verifications of code. Each verification phase improves the selectivity of your results, but increases the overall verification time. Improved selectivity can make results review more efficient, and hence make bugs in the code easier to isolate. Begin by running -to pass0 (Software Safety Analysis level 0) You can then address red errors and gray code before relaunching verification using higher integration levels.

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianess) from Simulink model hardware configuration settings. The software maps **Device vendor** and **Device type** settings on the Simulink **Configuration Parameters > Hardware Implementation** pane to **Target processor type** settings on the Polyspace **Configuration** pane.

Note: The software creates a generic target for the verification.

TargetLink Considerations

In this section...

TargetLink Support

For Windows, Polyspace Code Prover is tested with releases 3.4 and 3.5 of the dSPACE® Data Dictionary version and TargetLink Code Generator.

As Polyspace Code Prover extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing a verification.

Default Options

The following default options are set by Polyspace:

```
-I path_to_source_code
-desktop
-D PST_ERRNO
-I dspaceroot\matlab\TL\SimFiles\Generic
-I dspaceroot\matlab\TL\srcfiles\Generic
-I dspaceroot\matlab\TL\srcfiles\i86\LCC
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-ignore-constant-overflows
-scalar-overflows-behavior wrap-around
```

Note: *`dspaceroot`* and *`matlabroot`* are the dSPACE and MATLAB tool installation directories respectively.

Data Range Specification

You can constrain inputs, parameters, and outputs to lie within specified data ranges. See “Configure Data Range Settings” on page 16-7.

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

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

You can also manually define a DRS file in the Polyspace user interface. If you define a DRS file, the software appends the automatically generated information to the DRS file you create. Manually defined DRS information overrides automatically generated information for all variables.

DRS cannot be applied to static variables. Therefore, the compilation flags `-D static=` is set automatically. It has the effect of removing the static keyword from the code. If you have a problem with name clashes in the global name space you may need to either rename one of or variables or disable this option in Polyspace configuration.

Lookup Tables

The tool by default provides stubs for the lookup table functions. This behavior can be disabled from the Polyspace menu. The dSPACE data dictionary is used to define the range of their return values. Note that a lookup table that uses extrapolation will return full range for the type of variable that it returns.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to set the option `Clean code` and deselect the option `Enable sections/pragmas/inline/ISR/user attributes`.

When installing the Polyspace Model Link TL product, the `tlcgOptions` variable has been updated with `'PolyspaceSupport', 'on'` (see variable in `'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m'` file).

Related Examples

- "Run Analysis for TargetLink" on page 18-6

External Web Sites

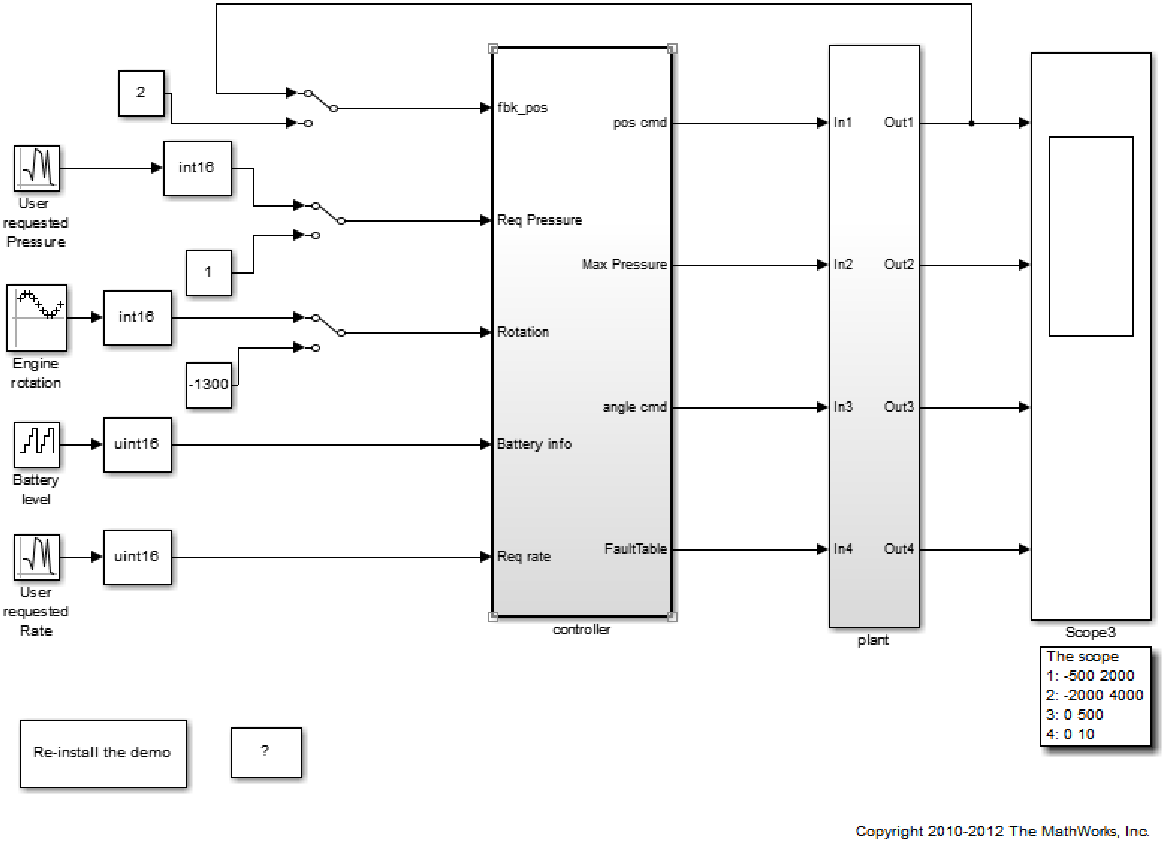
- dSPACE – TargetLink

Generate and Verify Code with Configured Model

You can generate Embedded Coder code from the configured model `psdemo_model_link_sl`. You can then run a Polyspace verification on the generated code.

To open `psdemo_model_link_sl` in the Simulink model window:

1 In the MATLAB Command Window, enter `psdemo_model_link_sl`.



To generate code and start the Polyspace analysis:

1. Double-click the Re-install the demo block to generate the handwritten code related to the S-function.
2. Right-click the subsystem `controller`.
3. From the context-menu, select **C/C++ Code** > **Build This Subsystem**.
4. In the Build code for Subsystem dialog box, click **Build** to generate code. When the code generation is complete, the code generation report opens.
5. Right-click the subsystem `controller`. From the context menu, select **Polyspace** > **Verify Code Generated for** > **Selected Subsystem** to start the verification.

   You can monitor progress from the Command Window.
6. Once the analysis is complete, right-click the `controller` subsystem. From the context menu, select **Polyspace** > **Open Results**. The results open in the Polyspace Code Prover environment.

View Results in Polyspace Code Prover

When a verification completes, you can view the results in the Polyspace user interface.

To view your results:

1 From the Simulink model window, select **Code** > **Polyspace** > **Open Results**.

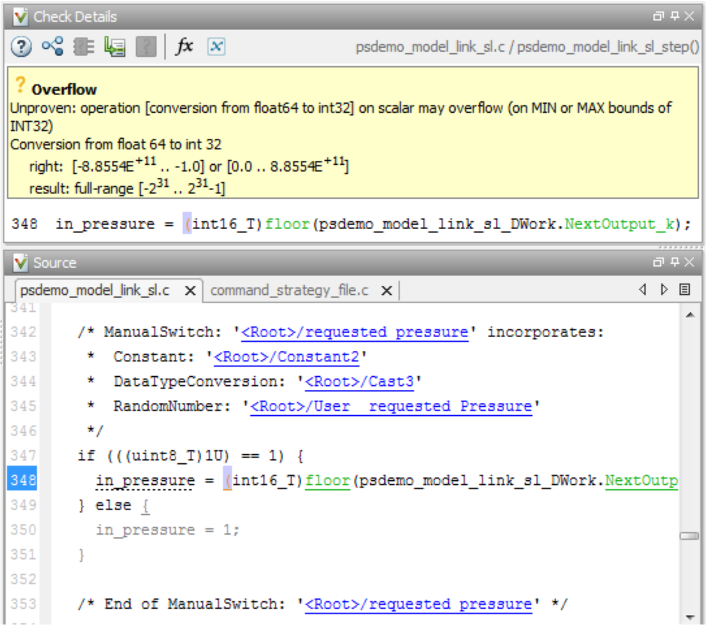
Note: If you set **Model reference verification depth** to `All` and selected **Model by model verification**, the Select the Result Folder to Open in Polyspace dialog box opens. The dialog box displays a hierarchy of referenced models from which the software generates code. To view the verification results for code generated from a specific model, select the model from the hierarchy. Then click **OK**.

You can also open results through a Model block or subsystem. From the Simulink model window, right-click the Model block or subsystem, and from the context menu, select **Polyspace** > **Open Results**.

After a few seconds, the Polyspace user interface opens.

2 On the **Results Summary** tab, click any check to review additional information.

In this example, the **Check Details** pane shows information about the orange check, and the **Source** pane shows the source code containing the orange check.



? **Overflow**
Unproven: operation [conversion from float64 to int32] on scalar may overflow (on MIN or MAX bounds of INT32)
Conversion from float 64 to int 32
right: [-8.8554E^{+11} .. -1.0] or [0.0 .. 8.8554E^{+11}]
result: full-range [-2^{31} .. 2^{31}-1]

```
in_pressure = (int16_T)floor(psdemo_model_link_sl_DWork.NextOutput_k);
```

```
/* ManualSwitch: '<Root>/requested pressure' incorporates:
 *  Constant: '<Root>/Constant2'
 *  DataTypeConversion: '<Root>/Cast3'
 *  RandomNumber: '<Root>/User  requested Pressure'
 */
if (((uint8_T)1U) == 1) {
  in_pressure = (int16_T)floor(psdemo_model_link_sl_DWork.NextOutp
} else {
  in_pressure = 1;
}

/* End of ManualSwitch: '<Root>/requested pressure' */
```

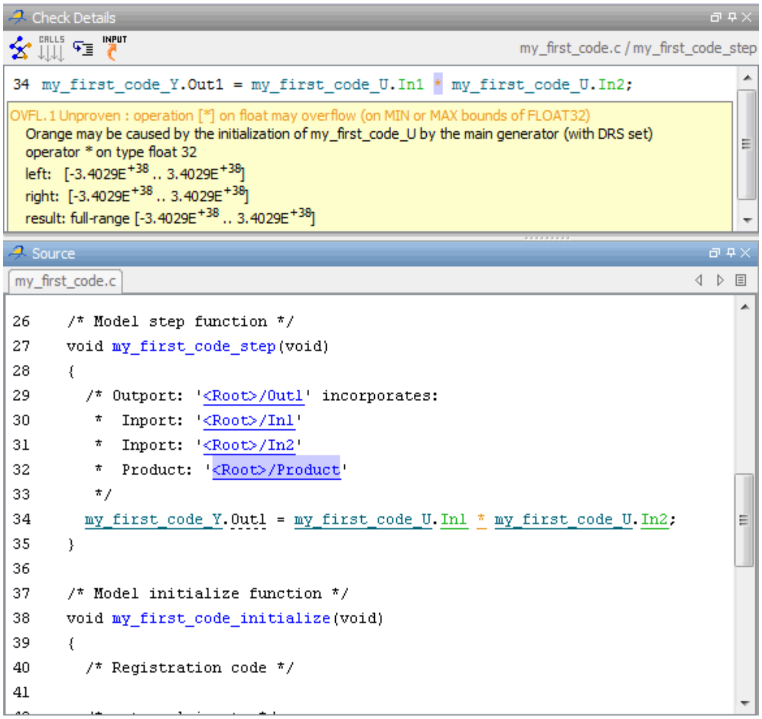
For more information on reviewing run-time checks, see “Review Results”.

For information on specific checks, see “Run-Time Checks”.

Identify Errors in Simulink Models

With Polyspace Code Prover, you can trace run-time checks in your verification results directly to your Simulink model.

Consider the following example, where the **Check Details** pane shows information about an orange check, and the **Source** pane shows the source code containing the orange check.



```
my_first_code_Y.Out1 = my_first_code_U.In1 * my_first_code_U.In2;
```

```
OVFL.1 Unproven : operation [*] on float may overflow (on MIN or MAX bounds of FLOAT32)
  Orange may be caused by the initialization of my_first_code_U by the main generator (with DRS set)
  operator * on type float 32
  left:  [-3.4029E+38 .. 3.4029E+38]
  right:  [-3.4029E+38 .. 3.4029E+38]
  result: full-range [-3.4029E+38 .. 3.4029E+38]
```

```
/* Model step function */
void my_first_code_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/Product'
   */
  my_first_code_Y.Out1 = my_first_code_U.In1 * my_first_code_U.In2;
}

/* Model initialize function */
void my_first_code_initialize(void)
{
  /* Registration code */

```

This orange check shows a potential overflow issue when multiplying the signals from the inports `In1` and `In2`. To fix this issue, you must return to the model.

To trace this run-time check to the model:

1. Click the blue underlined link (`<Root>/Product`) immediately before the check in the **Source** pane. The Simulink model opens, highlighting the block with the error.
2. Examine the model to find the cause of the check.

In this example, the highlighted block multiplies two full-range signals, which could result in an overflow. This could be a flaw in:

- Design — If the model is supposed to be robust for the full signal range, then the issue is a design flaw. In this case, you must change the model to accommodate the full signal range. For example, you could saturate the output of the previous block, or bound the signal with a Switch block.
- Specifications — If the model is supposed to work for specific input ranges, you can provide these ranges using block parameters or the base workspace. The verification will then read these ranges from the model. See "Specify Signal Ranges" on page 17-18.

Applying either solution should address the issue and cause the orange check to turn green.

More About

- "Troubleshoot Back to Model"

Troubleshoot Back to Model

In this section...

Back-to-Model Links Do Not Work

You may encounter issues with the back-to-model feature if:

- Your operating system is Windows Vista™ or Windows 7; and User Account Control (UAC) is enabled or you do not have administrator privileges.
- You have multiple versions of MATLAB installed.

To reconnect MATLAB and Polyspace:

1 Close Polyspace.
2 At the MATLAB command-line, enter `PolySpaceEnableCOMserver`.

   When you open your Polyspace results, the hyper-links will highlight the relevant blocks in your model.

Your Model Already Uses Highlighting

If your model extensively uses block coloring, the coloring from this feature may interfere with the colors already in your model. To change the color of blocks when they are linked to Polyspace results use this command:

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

Where *`color`* is one of the following:

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

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

Configure Code Analysis Options

Polyspace Configuration for Generated Code

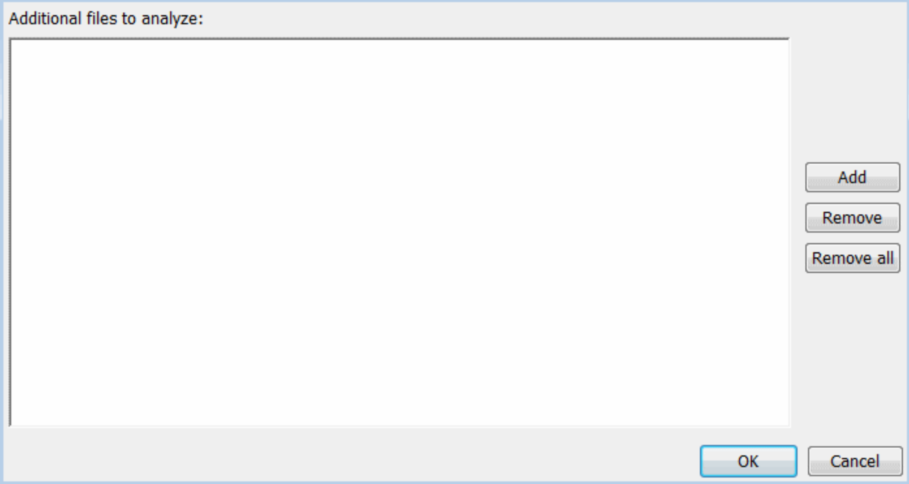
You do not have to manually create a Polyspace project or specify Polyspace options before running an analysis for your generated code. By default, Polyspace automatically creates a project and extracts the required information from your model. However, you can modify or specify additional options for your analysis:

- You may incorporate separately created code within the code generated from your Simulink model. See “Include Handwritten Code” on page 17-3.
- By default, the Polyspace analysis is contextual and treats tunable parameters as constants. You can specify a verification that considers robustness, including tunable parameters that lie within a range of values. See “Configure Data Range Settings”.
- You may customize the options for your analysis. For example, to specify the target environment or adjust precision settings. See “Configure Polyspace Analysis Options and Properties” on page 17-7 and “Recommended Polyspace options for Verifying Generated Code”.
- You may create specific configurations for batch runs. See “Save a Polyspace Configuration File Template” on page 17-8.
- If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. See “Set Custom Target Settings” on page 17-11.

Include Handwritten Code

Files such as S-function wrappers are, by default, not part of the Polyspace analysis. However, you can add these files manually.

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Select the **Enable additional file list** check box. Then click **Select files**. The Files Selector dialog box opens.



3. Click **Add**. The Select files to add dialog box opens.
4. Use the Select files to add dialog box to:

   - Navigate to the relevant folder
   - Add the required files.

   The software displays the selected files as a list under **Additional files to analyze**.

   Note: To remove a file from the list, select the file and click **Remove**. To remove all files from the list, click **Remove all**.

5. Click **OK**.

Configure Analysis Depth for Referenced Models

From the **Polyspace** pane, you can specify the analysis of generated code with respect to model reference hierarchy levels:

- **Model reference verification depth** — From the drop-down list, select one of the following:
  - `Current model only` — Default. The Polyspace runs code from the top level only. The software creates stubs to represent code from lower hierarchy levels.
  - `1` — The software analyzes code from the top level and the next level. For subsequent hierarchy levels, the software creates stubs.
  - `2` — The software analyzes code from the top level and the next two hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `3` — The software analyzes code from the top level and the next three hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `All` — The software analyzes code from the top level and all lower hierarchy levels.
- **Model by model verification** — Select this check box if you want the software to analyze code from each model separately.

Note: The same configuration settings apply to all referenced models within a top model. It does not matter whether you open the **Polyspace** pane from the top model window (**Code** > **Polyspace** > **Options**) or through the right-click context menu of a particular Model block within the top model. However, you can run analyses for code generated from specific Model blocks. See "Run Analysis for Embedded Coder" on page 18-5.

Check Coding Rules Compliance

You can check compliance with MISRA AC AGC and MISRA C:2004, and MISRA C:2012 coding rules directly from your Simulink model.

In addition, you can choose to run coding rules checking either with or without full code analysis.

To configure coding rules checking:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The **Polyspace** pane opens.
2. In the **Settings from** drop-down menu, select the type of analysis you want to perform.

   Depending on the type of code generated, different settings are available. The following tables describe the different settings.

   ### C Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA AC AGC checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA AC-AGC rule set.
`Project configuration and MISRA C 2004 checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C:2004 coding rules.
`Project configuration and MISRA C 2012 ACG checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C:2012 coding guidelines.
`MISRA AC AGC checking`	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.

Setting	Description
`MISRA C 2004 checking`	Check compliance with MISRA C:2004 coding rules. Polyspace stops after rules checking.
`MISRA C 2012 ACG checking`	Check compliance with MISRA C:2012 coding rules using generated code categories. Polyspace stops after guideline checking.

C++ Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA C ++ coding rules.
`Project configuration and JSF C ++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with JSF C++ coding rules.
`MISRA C++ rule checking`	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
`JSF C++ rule checking`	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Configure Polyspace Analysis Options and Properties

From Simulink, you can specify Polyspace options to change the configuration of the Polyspace Analysis. For example, you can specify the processor type and operating system of your target device.

For descriptions of options, see “Analysis Options for C Code” or “Analysis Options for C Code”.

There are two ways to configure analysis options:

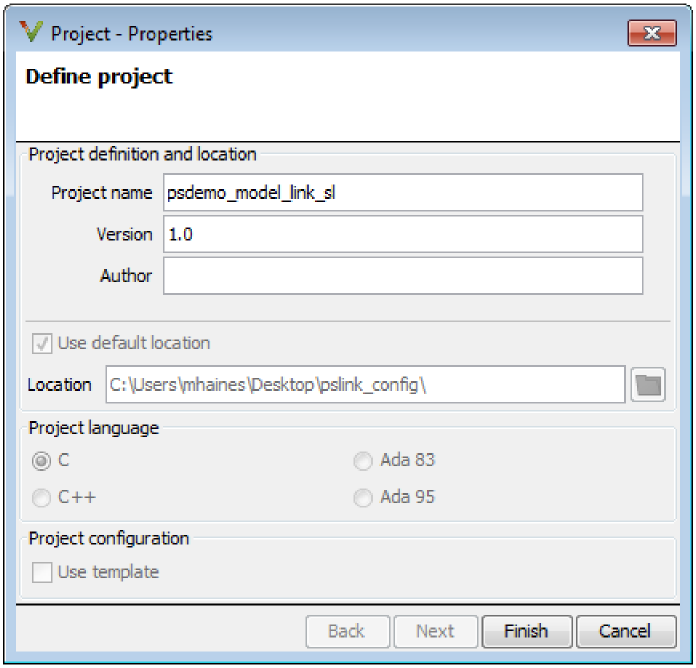
In this section...

Set Advanced Analysis Options

1. From Simulink, select **Code** > **Polyspace** > **Options**.
2. In the Polyspace parameter configuration pane, select **Configure**.

   The Polyspace Configuration window opens.
3. Set options required by your application.

   The first time you open the configuration, the software sets certain options by default depending on your code generator.
4. On the toolbar, click the **Project properties** icon.



Project definition and location

Project name: psdemo_model_link_sl

Version: 1.0

Author:

☑ Use default location

Location: C:\Users\mhaines\Desktop\pslink_config\

Project language

● C ○ Ada 83

○ C++ ○ Ada 95

Project configuration

□ Use template

Save a Polyspace Configuration File Template

During a batch run, you may want use different configurations. At the MATLAB command-line, use `pslinkfun('settemplate',...)` to apply a configuration defined by a configuration file template.

To create a configuration file template:

1 In the Simulink model window, select **Code** > **Polyspace** > **Options**. The Parameter Configuration window opens to the Polyspace pane.

2 Click **Configure**.

The Polyspace Configuration window opens. Use this pane to customize the target and cross compiler.

3 Save your changes and close.

4 Make a copy of the updated project configuration file, for example, `my_first_code_polyspace.psprj`.

5. Rename the copy, for example, `my_cross_compiler.psprj`. This is your new configuration file template.

To use a configuration template:

- Run the `pslinkfun` command in the MATLAB Command Window. For example:

  ```
  pslinkfun('settemplate','C:\Work\my_cross_compiler.psprj')
  ```
- Add the file in the Parameter Configuration window. See “Use a Custom Configuration File” on page 17-9.

Use a Custom Configuration File

If you already have a configuration you want to use, you can add the configuration file to your project.

1. From Simulink, select **Code** > **Polyspace** > **Options**.
2. In the Polyspace parameter configuration pane, select **Use custom project file**.
3. In the text box, enter the full path to a `.psprj` file, or click **Browse for project file** to browse for a `.psprj` file.

Remove Polyspace Options From Simulink Model

You can remove Polyspace configuration information from your Simulink model.

For a top model:

1. Select **Code** > **Polyspace** > **Remove Options from Current Configuration**.
2. Save the model.

For a Model block or subsystem:

1. Right-click the Model block or subsystem.
2. From the context menu, select **Polyspace** > **Remove Options from Current Configuration**.
3. Save the model.

See Also

`pslinkfun` | `pslinkoptions`

Related Examples

- “Save a Polyspace Configuration File Template” on page 17-8

More About

- “Embedded Coder Considerations”
- “TargetLink Considerations”
- “Recommended Polyspace options for Verifying Generated Code”

Set Custom Target Settings

If your target has specific setting, you can analyze your code in context of those settings. For example, if you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. The software automatically identifies the compiler from the Simulink model. If the compiler is 16-bit and you do not specify the relevant header files, the software produces an error when you try to run an analysis.

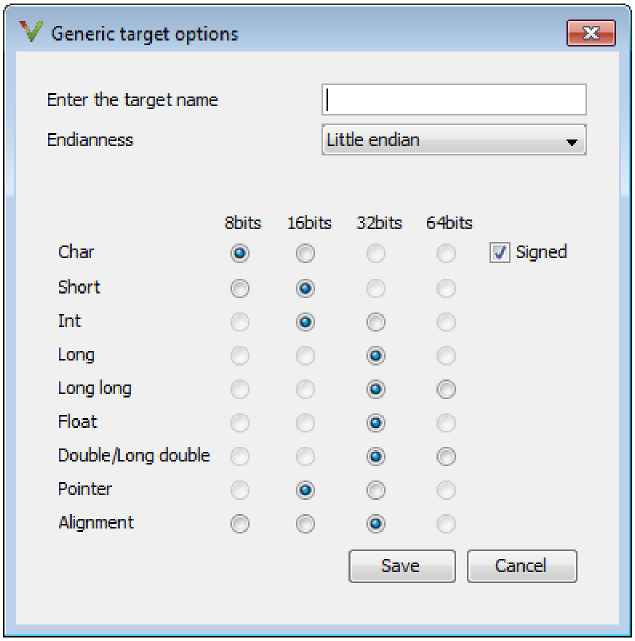
Note: For a 32-bit or 64-bit target processor, the software automatically specifies the default header file.

---

1. In the Simulink model window, select **Code** > **Polyspace** > **Options**. The Parameter Configuration window opens to the Polyspace pane.
2. Click **Configure**.

   The Polyspace Configuration window opens. Use this pane to customize the target and cross compiler.
3. From the **Configuration** tree, expand the **Target & Compiler** node.
4. In the **Target Environment** section, use the **Target processor type** option to define the size of data types.

   a. From the drop-down list, select `mcpu...(Advanced)`. The Generic target options dialog box opens.



Generic target options

Enter the target name

Endianness: Little endian

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

Use this dialog box to create a new target and specify data types for the target. Then click **Save**.

5 From the Configuration tree, select **Target & Compiler** > **Macros**. Use the **Preprocessor definitions** section to define preprocessor macros for your cross-compiler.

To add a macro, in the **Macros** table, select ➕. In the new line, enter the required text.

To remove a macro, select the macro and click ✖.

Note: If you use the LCC cross-compiler, then you must specify the `MATLAB_MEX_FILE` macro.

---

6 Select **Target & Compiler** > **Environment Settings**.

7 In the **Include folders** (or **Include**) section, specify a folder (or header file) path by doing one of the following:

- Select ➕ and enter the folder or file path.
- Select 📁 and use the dialog box to navigate to the required folder (or file).

You can remove an item from the displayed list by selecting the item and then clicking ✖.

8 Save your changes and close.

To use your configuration settings in other projects, see “Save a Polyspace Configuration File Template” on page 17-8.

Set Up Remote Batch Analysis

By default, the Polyspace software runs locally. To specify a remote analysis:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Select **Configure**.

3 In the Polyspace Configuration window, select the **Distributed Computing** pane.

4 Select the **Batch** check box.

5 If you use Polyspace Metrics as a results repository, select **Add to results repository**.

   Before running your must also make sure you are connected to a Server.

6 From the toolbar, select **Options** > **Preferences**. For help filling in this dialog, see "Configure Polyspace Preferences".

7 Close the configuration window and save your changes.

8 Select **Apply**.

Manage Results

In this section...

Polyspace creates a set of analysis results

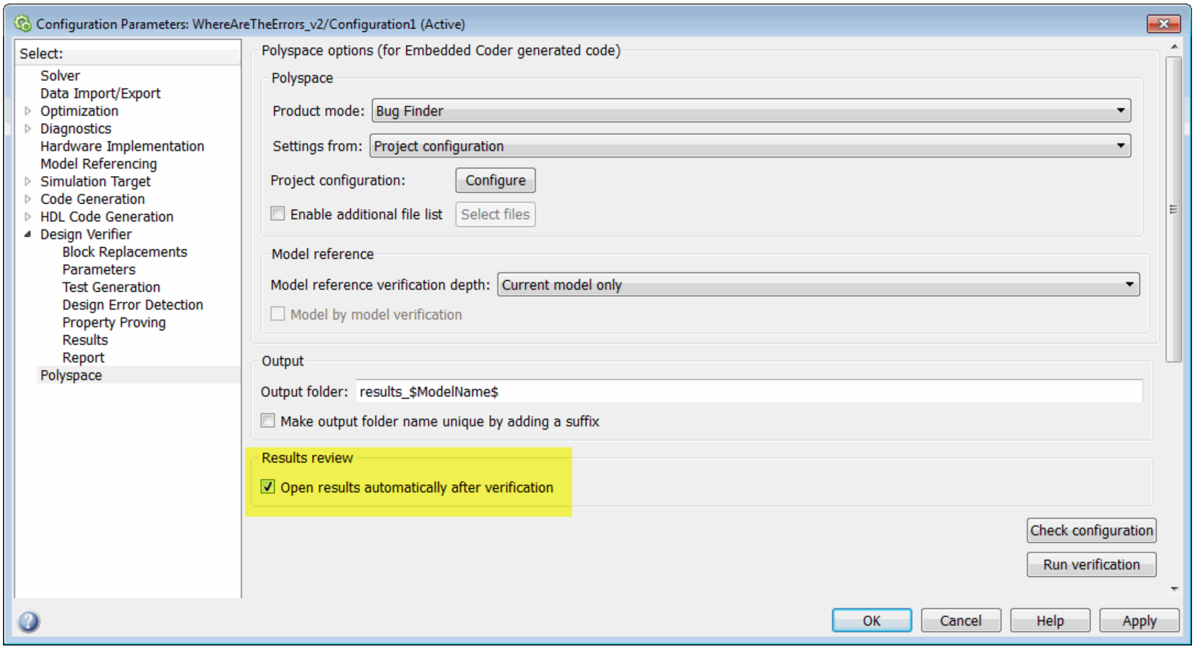
Open Polyspace Results Automatically

You can configure the software to automatically open your Polyspace results after you start the analysis. If you are doing a remote analysis, the Polyspace Metrics webpage opens. When the remote job is complete, you can download your results from Polyspace Metrics. If you are doing a local analysis, when the local job is complete, the Polyspace environment opens the results in the Polyspace interface.

To configure the results to open automatically:

1. From the model window, select **Code** > **Polyspace** > **Options**.

   The Polyspace pane opens.



Configuration Parameters: WhereAreTheErrors_v2/Configuration1 (Active)

Polyspace options (for Embedded Coder generated code)

Product mode: Bug Finder

Settings from: Project configuration

Project configuration: Configure

☐ Enable additional file list

Model reference verification depth: Current model only

☐ Model by model verification

Output folder: results_$ModelName$

☐ Make output folder name unique by adding a suffix

☑ Open results automatically after verification

2 In the Results review section, select **Open results automatically after verification**.

3 Click **Apply** to save your settings.

Specify Location of Results

By default, the software stores your results in `Current Folder\results_model_name`. Every time you rerun, your old results are over written. To customize these options:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens to the Polyspace pane.

2 In the **Output folder** field, specify a full path for your results folder. By default, the software stores results in the current folder.

3 If you want to avoid overwriting results from previous analyses, select **Make output folder name unique by adding a suffix**.

Instead of overwriting an existing folder, the software specifies a new location for the results folder by appending a unique number to the folder name.

Save Results to a Simulink Project

By default, the software stores your results in `*Current Folder*\results_*model_name*`. If you use a Simulink project for your model work, you can store your Polyspace results there as well for better organization. To add your results to a Simulink Project:

1. Open your Simulink project.
2. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens with the Polyspace pane displayed.
3. Select **Add results to current Simulink Project**.
4. Run your analysis.

   Your results are saved to the Simulink project you opened in step 1.

Specify Signal Ranges

If you constrain signals in your Simulink model to lie within specified ranges, Polyspace software automatically applies these constraints during verification of the generated code. This can improve the precision of your results.

You can specify a range for a model signal by:

- Applying constraints through source block parameters. See "Specify Signal Range through Source Block Parameters" on page 17-18.
- Constraining signals through the base workspace. See "Specify Signal Range through Base Workspace" on page 17-20.

Note: You can also manually define data ranges using the DRS feature in the Polyspace verification environment. If you manually define a DRS file, the software automatically appends any signal range information from your model to the DRS file. However, manually defined DRS information overrides information generated from the model for all variables.

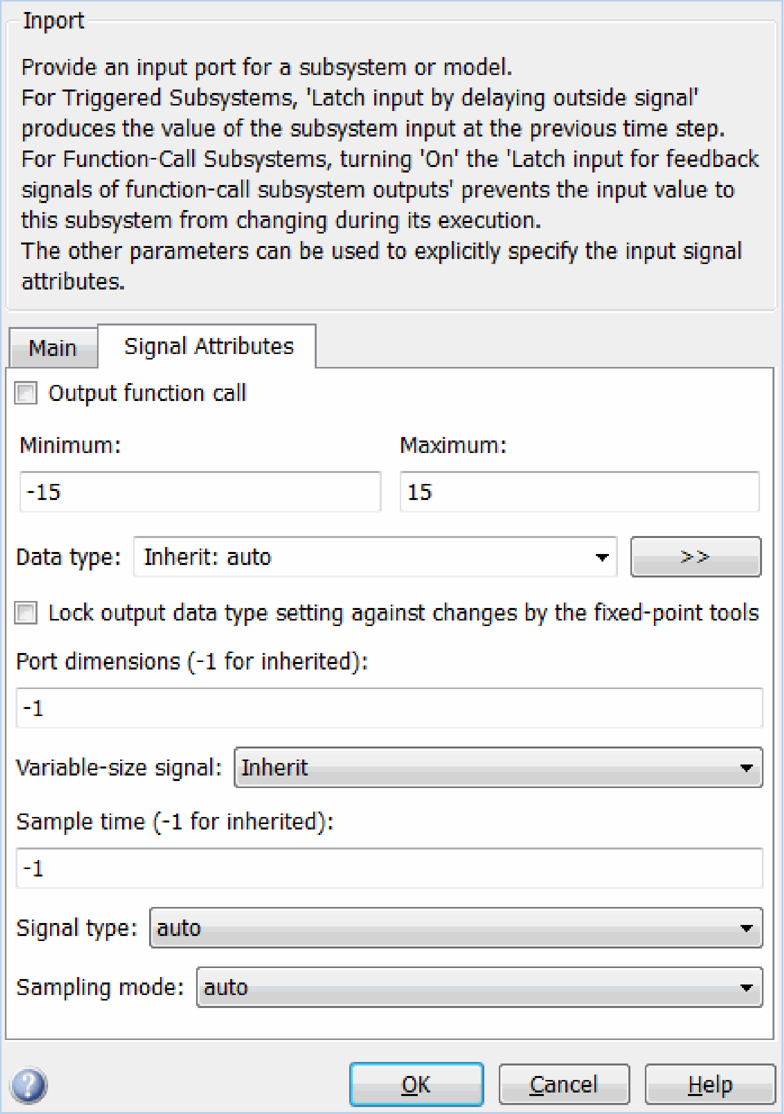
Specify Signal Range through Source Block Parameters

You can specify a signal range by applying constraints to source block parameters.

Specifying a range through source block parameters is often easier than creating signal objects in the base workspace, but must be repeated for each source block. For information on using the base workspace, see "Specify Signal Range through Base Workspace" on page 17-20.

To specify a signal range using source block parameters:

1. Double-click the source block in your model. The Source Block Parameters dialog box opens.
2. Select the **Signal Attributes** tab.
3. Specify the **Minimum** value for the signal, for example, -15.
4. Specify the **Maximum** value for the signal, for example, 15.



Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

□ Output function call

Minimum: -15

Maximum: 15

Data type: Inherit: auto

□ Lock output data type setting against changes by the fixed-point tools

Port dimensions (-1 for inherited): -1

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Sampling mode: auto

5 Click **OK**.

Specify Signal Range through Base Workspace

You can specify a signal range by creating signal objects in the MATLAB workspace. This information is used to initialize each global variable to the range of valid values, as defined by the min-max information in the workspace.

Note: You can also specify a signal range by applying constraints to individual source block parameters. This method can be easier than creating signal objects in the base workspace, but must be repeated for each source block. For more information, see "Specify Signal Range through Source Block Parameters" on page 17-18.

To specify an input signal range through the base workspace:

1 Configure the signal to use, for example, the `ExportedGlobal` storage class:

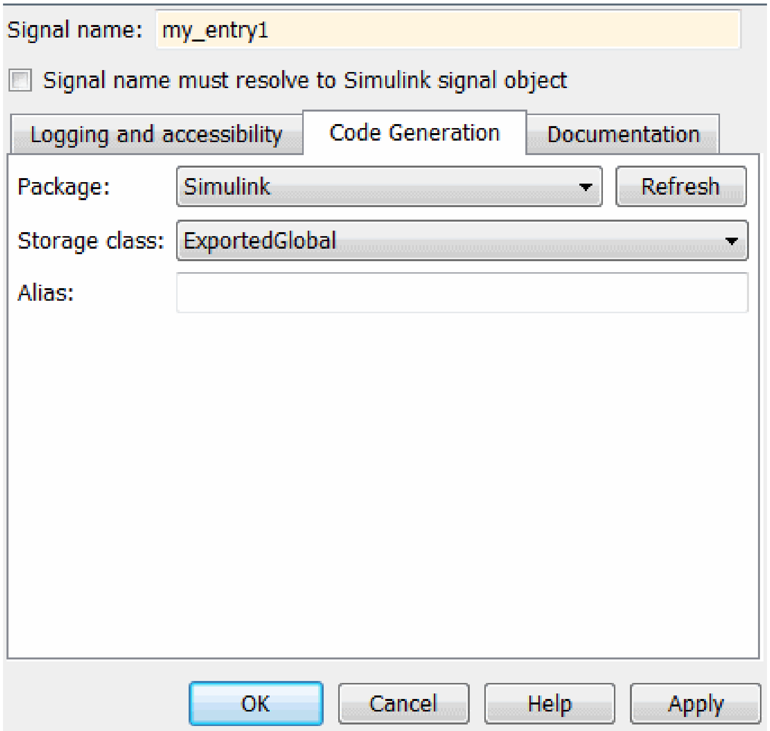
   a Right-click the signal. From the context menu, select **Properties**. The Signal Properties dialog box opens.

   b In the **Signal name** field, enter a name, for example, `my_entry1`.

   c Select the **Code Generation** tab.

   d From the **Package** drop-down menu, select `Simulink`.

   e In the **Storage class** drop-down menu, select `ExportedGlobal`.



Signal name: my_entry1

□ Signal name must resolve to Simulink signal object

Package: Simulink

Storage class: ExportedGlobal

Alias:

- **f** Click **OK**, which applies your changes and closes the dialog box.

2 Using Model Explorer, specify the signal range:

- **a** Select **Tools** > **Model Explorer** to open Model Explorer.
- **b** From the **Model Hierarchy** tree, select **Base Workspace**.
- **c** Click the `Add Simulink Signal` button to create a signal. Rename this signal, for example, `my_entry1`.
- **d** Set the **Minimum** value for the signal, for example, to `-15`.
- **e** Set the **Maximum** value for the signal, for example, to `15`.
- **f** From the **Storage class** drop-down list, select `ExportedGlobal`.
- **g** Click **Apply**.

18

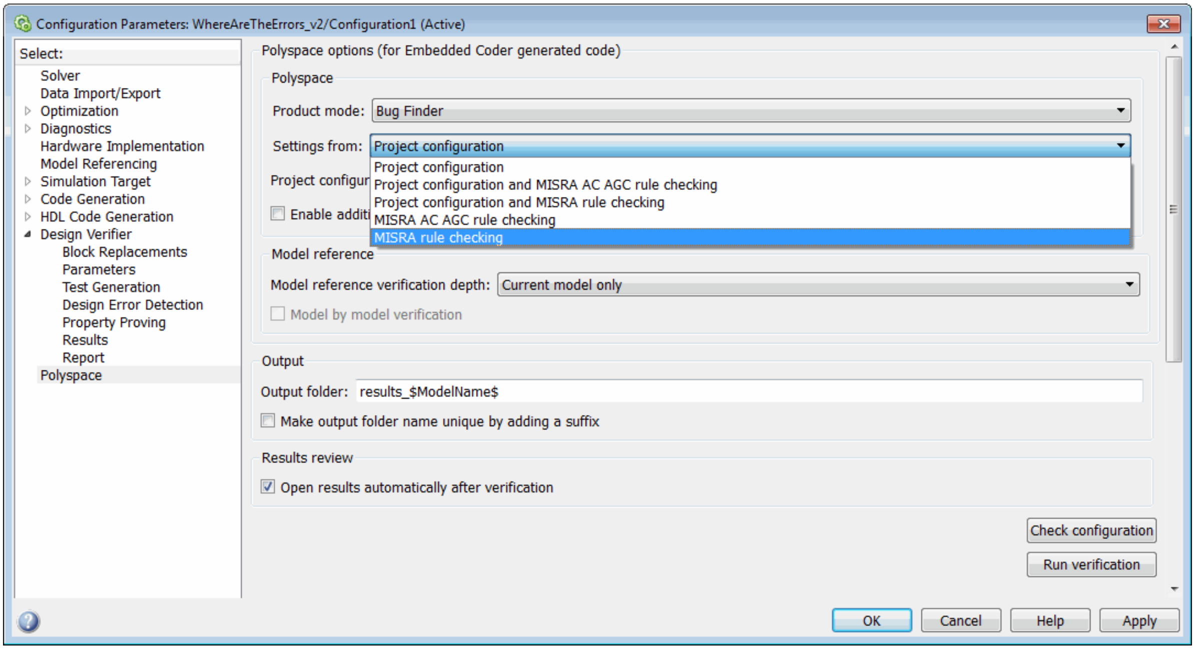
Run Polyspace on Generated Code

Specify Type of Analysis to Perform

Before running Polyspace, you can specify what type of analysis you want to run. You can choose to run code analysis, coding rules checking, or both.

To specify the type of analysis to run:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The **Configuration Parameter** window opens to the **Polyspace** options pane.



Configuration Parameters: WhereAreTheErrors_v2/Configuration1 (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration

Project configur

□ Enable additi

Model reference

Model reference verification depth: Current model only

□ Model by model verification

Output

Output folder: results_$ModelName$

□ Make output folder name unique by adding a suffix

Results review

☑ Open results automatically after verification

2 In the **Settings from** drop-down menu, select the type of analysis you want to perform.

Depending on the type of code generated, different settings are available. The following tables describe the different settings.

C Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA AC AGC rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA AC-AGC rule set.
`Project configuration and MISRA rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C coding rules.
`MISRA AC AGC rule checking`	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.
`MISRA rule checking`	Check compliance with MISRA C coding rules. Polyspace stops after rules checking.

C++ Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA C ++ coding rules.
`Project configuration and JSF C ++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with JSF C++ coding rules.
`MISRA C++ rule checking`	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
`JSF C++ rule checking`	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Run Analysis for Embedded Coder

To start Polyspace with:

- Code generated from the top model, from the Simulink model window, select **Code** > **Polyspace** > **Verify Code Generated for** > **Model**.
- All code generated as model referenced code, from the model window, select **Code** > **Polyspace** > **Verify Code Generated for** > **Referenced Model**.
- Model reference code associated with a specific block or subsystem, right-click the Model block or subsystem. From the context menu, select **Verify Code Generated for** > **Selected Subsystem**.

Note: You can also start the Polyspace software from the **Polyspace** configuration parameter pane by clicking **Run verification**.

When the Polyspace software starts, messages appear in the MATLAB Command window:

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder C:\PolySpace_Results\results_my_first_code
                                           for system my_first_code
### Checking Polyspace Model-Link Configuration:
### Parameters used for code verification:
 System                : my_first_code
 Results Folder        : C:\PolySpace_Results\results_my_first_code
 Additional Files      : 0
 Remote                : 0
 Model Reference Depth : Current model only
 Model by Model        : 0
 DRS input mode        : DesignMinMax
 DRS parameter mode    : None
 DRS output mode       : None
...
```

Follow the progress of the analysis in the MATLAB Command window. If you are running a remote, batch, analysis you can follow the later stages through the Polyspace Job Monitor.

The software writes status messages to a log file in the results folder.

Run Analysis for TargetLink

To start the Polyspace software:

1. In your model, select the Target Link subsystem.
2. In the Simulink model window select **Code** > **Polyspace** > **Verify Code Generated for** > **Selected Target Link Subsystem**.

   Messages appear in the MATLAB Command window:

   ```
   ### Starting Polyspace verification for Embedded Coder
   ### Creating results folder results_WhereAreTheErrors_v2
                            for system WhereAreTheErrors_v2
   ### Parameters used for code verification:
    System               : WhereAreTheErrors_v2
    Results Folder       : H:\Desktop\Test_Cases\ModelLink_Testers
                                     \results_WhereAreTheErrors_v2
    Additional Files     : 0
    Verifier settings    : PrjConfig
    DRS input mode       : DesignMinMax
    DRS parameter mode   : None
    DRS output mode      : None
    Model Reference Depth : Current model only
    Model by Model       : 0
   ```

   The exact messages depend on the code generator you use and the Polyspace product. The software writes status messages to a log file in the results folder.

Follow the progress of the software in the MATLAB Command Window. If you are running a remote, batch analysis, you can follow the later stages through the Polyspace Job Monitor

Note: Verification of a 3,000 block model will take approximately one hour to verify, or about 15 minutes for each 2,000 lines of generated code.

Monitor Progress

In this section...

Local Analyses

For a local Polyspace runs, you can follow the progress of the software in the MATLAB Command Window. The software also saves the status messages to a log file in the results folder.

Remote Batch Analyses

For a remote analysis, you can follow the initial stages of the analysis in the MATLAB Command window.

Once the compilation phase is complete, you can follow the progress of the software using the Polyspace Job Monitor.

From Simulink, select **Code** > **Polyspace** > **Open Job Monitor**

For more information, see "Monitor Progress".

Using Polyspace Software in the Eclipse IDE

Install Polyspace Plug-In for Eclipse

In this section...
"Install Polyspace Plug-In for Eclipse IDE" on page 19-2
"Uninstall Polyspace Plug-In for Eclipse IDE" on page 19-4

Install Polyspace Plug-In for Eclipse IDE

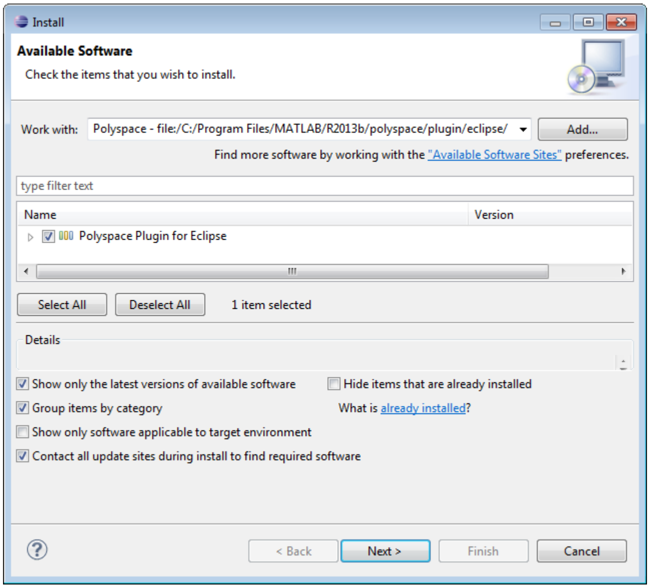
You can install the Polyspace plug-in only after you:

- Install and set up Eclipse Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java 7. See Java documentation at www.java.com.
- Uninstall any previous Polyspace plug-ins. For more information, see "Uninstall Polyspace Plug-In for Eclipse IDE" on page 19-4.

To install the Polyspace plug-in:

1. From the Eclipse editor, select **Help** > **Install New Software**. The Install wizard opens, displaying the Available Software page.
2. Click **Add** to open the Add Repository dialog box.
3. In the **Name** field, specify a name for your Polyspace site, for example, `Polyspace_Eclipse_PlugIn`.
4. Click **Local**, to open the Browse for Folder dialog box.
5. Navigate to the `*MATLAB_Install*\polyspace\plugin\eclipse` folder. Then click **OK**.

   `*MATLAB_Install*` is the installation folder for the Polyspace product.
6. Click **OK** to close the Add Repository dialog box.
7. On the Available Software page, select `Polyspace Plugin for Eclipse`.



Install

Available Software

Check the items that you wish to install.

Work with: Polyspace - file:/C:/Program Files/MATLAB/R2013b/polyspace/plugin/eclipse/ [Add...]

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☑ Polyspace Plugin for Eclipse	

[Select All] [Deselect All] 1 item selected

Details

☑ Show only the latest versions of available software ☐ Hide items that are already installed

☑ Group items by category What is already installed?

☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

[< Back] [Next >] [Finish] [Cancel]

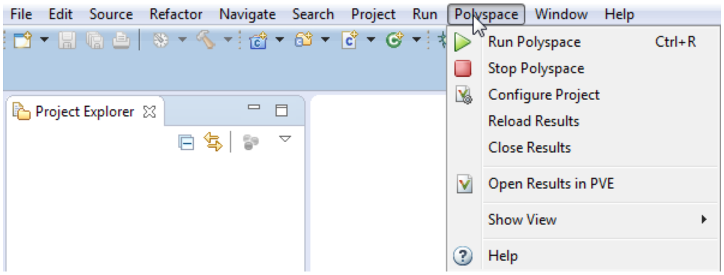
8 Click **Next**.

9 On the Install Details page, click **Next**.

10 On the Review Licenses page, review and accept the licence agreement. Then click **Finish**.

Once you install the plug-in, in the Eclipse editor, you'll see:

- A **Polyspace** menu
- A **Polyspace Run**, **Results Summary**, and **Check Details** view.



Uninstall Polyspace Plug-In for Eclipse IDE

Before installing a new Polyspace plug-in, you must uninstall any previous Polyspace plug-ins:

1. In Eclipse, select **Help** > **About Eclipse**.
2. Select **Installation Details**.
3. Select the Polyspace plug-in and select **Uninstall**.

   Follow the uninstall wizard to remove the Polyspace plug-in. You must restart Eclipse for changes to take effect.

Related Examples

- “Verify Code in the Eclipse IDE”

More About

- “Verification in Eclipse”

Verify Code in the Eclipse IDE

In this section...

Workflow for Code Verification in Eclipse

You can use Polyspace software to verify code that you develop within the Eclipse Integrated Development Environment (IDE).

A typical workflow is:

1. Create an Eclipse project and develop code within your project.
2. Configure verification options.
3. Start the verification.
4. Review the verification results. Fix run-time errors and restart the verification.

Install the Polyspace plug-in for Eclipse IDE before you verify code in Eclipse IDE. For more information, see "Install Polyspace Plug-In for Eclipse" on page 19-2.

Create Eclipse Project

If your source files do not belong to an Eclipse project, then create a project using the Eclipse editor:

1. Select **File** > **New** > **C Project**.
2. Clear the **Use default location** check box.
3. Click **Browse** to navigate to the folder containing your source files, for example, `C:\Test\Source_C`.
4. In the **Project name** field, enter a name, for example, `Demo_C`.
5. In the **Project Type** tree, under **Executable**, select **Empty Project** .

6 Under **Toolchains**, select your installed toolchain, for example, `MinGW GCC`.

7 Click **Finish**. An Eclipse project is created.

For information on developing code within Eclipse IDE, refer to www.eclipse.org.

Configure Polyspace Verification

To configure your verification:

1 In **Project Explorer**, select the project or files that you want to verify.

2 Select **Polyspace** > **Configure Project** to open the **Configuration** pane in the Polyspace verification environment.

3 Select your options for the verification process.

4 Select **File** > **Save** to save your options.

For more information, see "Analysis Options".

Note: Your Eclipse compiler options for include paths (`-I`) and symbol definitions (`-D`) are automatically added to the list of Polyspace analysis options.

To view the `-I` and `-D` options in the Eclipse editor :

1 Select **Project** > **Properties** to open the Properties for Project dialog box.

2 In the tree, under **C/C++ General** , select **Paths and Symbols** .

3 Select **Includes** to view the `-I` options or **Symbols** to view the `-D` options.

Start Verification

To start a Polyspace verification from the Eclipse editor:

1 Select the file, files, or class that you want to verify.

2 Either right-click and select **Run Polyspace Code Prover**, or select **Polyspace** > **Run Polyspace**.

You can see the progress of the verification in the **Polyspace Run** view. If you see an error or warning during the compilation phase, double-click it to go to the corresponding location in the source code. Once the verification is over, the results are displayed on the **Results Summary** tab.

3. To stop a verification, select **Polyspace** > **Stop Polyspace**. Alternatively you can use the 🟥 button in the **Polyspace Run** view.

Review Results

You can examine results of the verification either in Eclipse or the Polyspace user interface.

- **Eclipse**:

  After you run a verification in Eclipse, your results open automatically on the **Results Summary** tab. Select a check to see detailed information on the **Check Details** tab. If you close Eclipse or run Polyspace on another Eclipse project, your results are closed. To reopen your results in Eclipse, select **Polyspace** > **Reload Results**.
- **Polyspace user interface**:

  The results in Eclipse are overwritten every time a new verification is performed. However, Polyspace automatically imports **Status**, **Classification**, and **Comment** information to the new verification. If you want to save your earlier results:

  1. Select **Polyspace** > **Open Results in PVE** to open your results in the Polyspace user interface.
  2. Upload your results to Metrics by selecting **Metrics** > **Upload to Metrics**

  In addition to the **Results Summary** and **Check Details** views also available in Eclipse, in the Polyspace user interface, you can use other views to navigate in your source code or view tooltips with information about variable ranges.

Related Examples

- "Review Results"

20

Using Polyspace Software in Visual Studio

Install Polyspace Add-In for Visual Studio

Install Polyspace Add-In

The Polyspace Add-in is supported for Visual Studio 2008, 2010. You can install the Polyspace add-in only after you:

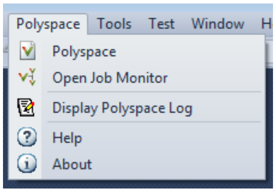
- Install Visual Studio.
- Uninstall any previous Polyspace add-ins. For more information see "Uninstall Polyspace Add-In for Visual Studio" on page 20-3.

To install the Polyspace add-in:

1. In the Visual Studio editor, select **Tools** > **Options** to open the Options dialog box.
2. Select the **Environment** > **Add-in/Macros Security** pane to display the list of Visual Studio add-in folders.
3. Select the following check boxes:
   - **Allow macros to run**
   - **Allow Add-in components to load**
4. Click **Add** to open the Browse For Folder dialog box.
5. Navigate to `*MATLAB_Install*\polyspace\plugin\msvc\*VS_version*`
   - *`MATLAB_Install`* is the installation folder for the Polyspace product.
   - *`VS_version`* corresponds to the version of Visual Studio that you have installed, for example, `2010`.
6. Click **OK** to close the Browse for Folder dialog box.
7. To close the Options dialog box, click **OK**.

You must restart Visual Studio for the changes to take effect. After you install the add-in, the Visual Studio editor has:

- A **Polyspace** menu



- A **Polyspace Log** view

Uninstall Polyspace Add-In for Visual Studio

Before installing a new Polyspace add-in, you must uninstall any previous Polyspace add-ins.

1 In the Visual Studio editor, select **Tools** > **Options** to open the Options dialog box.

2 Select the **Environment** > **Add-in/Macros Security** pane to display the list of Visual Studio add-in folders.

3 Select the Polyspace add-in and select **Remove**.

4 To close the Options dialog box, click **OK**.

You must restart Visual Studio for the changes to take effect.

Related Examples

- "Run Polyspace in Visual Studio"

More About

- "Verification in Visual Studio"

Create Project Using Visual Studio Information

In this section...
"Trace Visual Studio Build" on page 20-4
"Import Visual Studio Project" on page 20-7

Trace Visual Studio Build

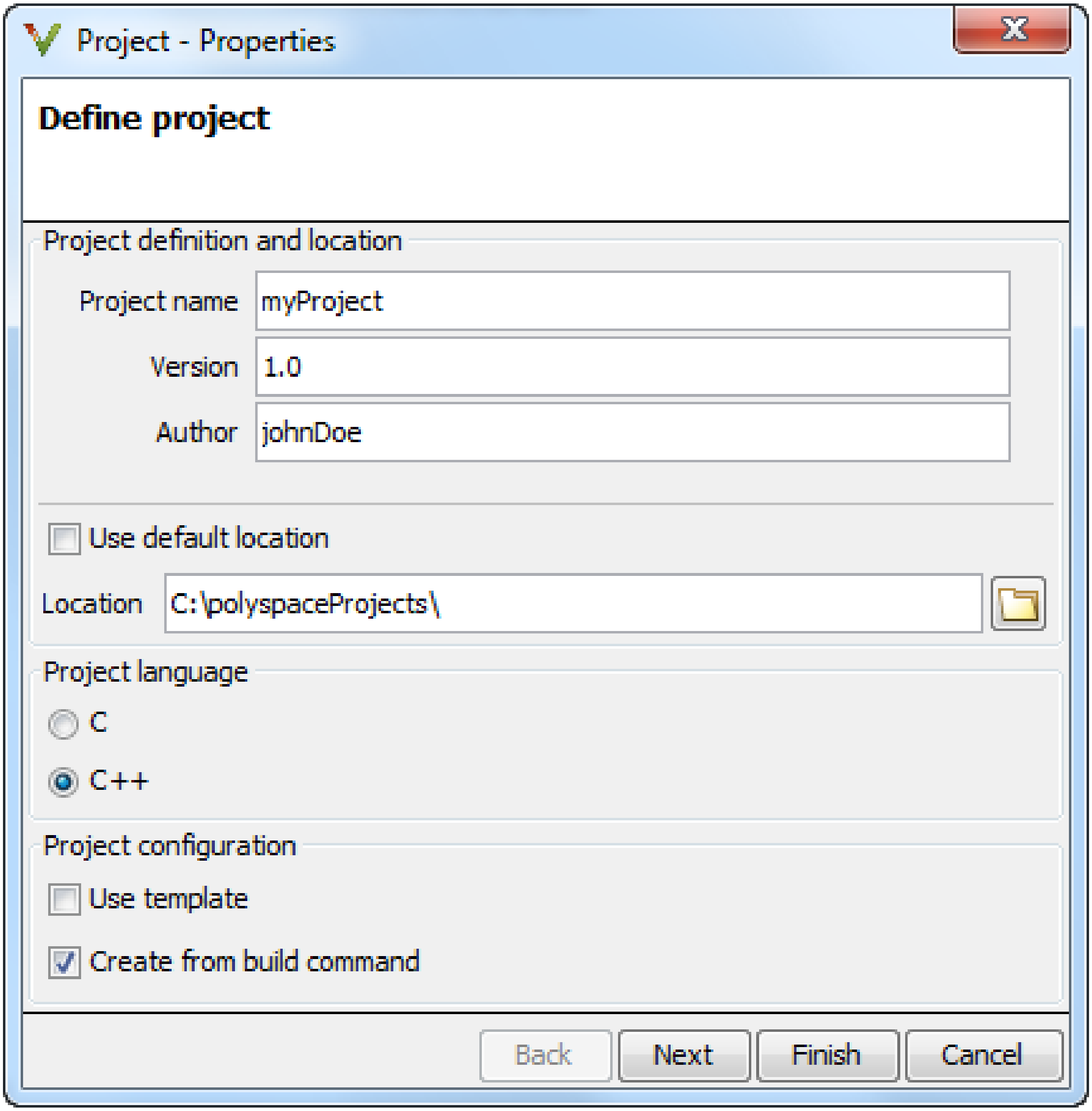
To create a Polyspace project, you can trace your Visual Studio build. For Polyspace to trace your Visual Studio build, you must install both `x86` and `x64` versions of the Visual C++ Redistributable for Visual Studio 2012 from the Microsoft website.

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

2 In the Project – Properties window, enter your project information.

   a Choose **C++** as **Project Language**.

   b Under **Project Configuration**, select **Create from build command** and click **Next**.



Project - Properties

Define project

Project definition and location

Project name: myProject

Version: 1.0

Author: johnDoe

□ Use default location

Location: C:\polyspaceProjects\

Project language

○ C

● C++

Project configuration

□ Use template

☑ Create from build command

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

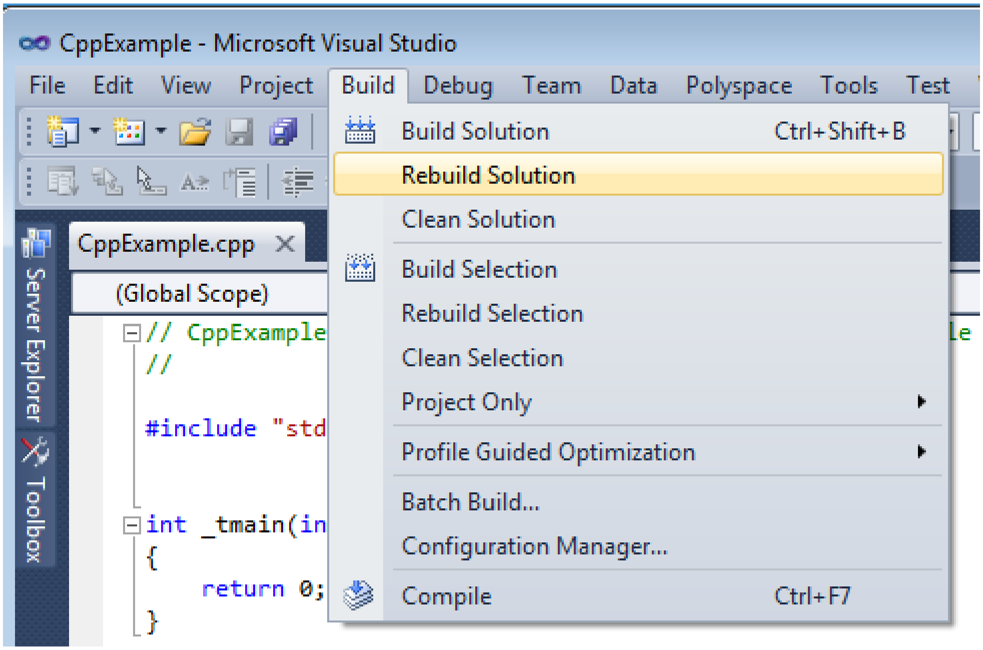
4 In the field **Specify working directory for running build command**, enter `C:\`.

Click ▷ Run.

This action opens the Visual Studio environment.

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

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



```
// CppExample
//

#include "std

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

6 After the project builds, close Visual Studio.

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

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

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

Import Visual Studio Project

Note: This feature is will be removed in a future release.

You can directly create a Polyspace project from a Visual Studio project file with extension `.vcproj`. The Visual Studio import retrieves the following information from a Visual Studio project:

- **Source** files
- **Include** folders
- Some **Target & Compiler** options
- **Preprocessor Macros**

Note: For Visual Studio 2010 or Visual Studio 2012, you cannot directly import your project.

1. In the Polyspace interface, select **Tools** > **Import Visual Studio Project**.
2. In the Import Visual Studio dialog box, specify the **Visual Studio project** that you want to use.
3. You can:
   - **Create new Polyspace project**: Enter full path to a new Polyspace project.
   - **Update existing Polyspace project**: The dropdown list contains all projects currently open in the **Project Browser**. Select the project you want to update.
4. Click **Import**.

More About

- "Troubleshooting Project Creation from Visual Studio Build"

Troubleshooting Project Creation from Visual Studio Build

In this section...
"Cannot Create Project from Visual Studio Build" on page 20-8
"Compilation Error After Creating Project from Visual Studio Build" on page 20-8

Cannot Create Project from Visual Studio Build

If you are trying to import a Visual Studio 2010 or Visual Studio 2012 project and `polyspace-configure` does not work properly, do the following:

1. Stop the `MSBuild.exe` process.
2. Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
3. Specify `MSBuild.exe` with the`/nodereuse:false` option.
4. Restart the Polyspace configuration tool:

```
polyspace-configure.exe -lang cpp <MSVS path>/msbuild sample.sln
```

Compilation Error After Creating Project from Visual Studio Build

If you automatically set up your project from a Visual Studio 2010 build, you can face compilation errors. By default, Polyspace assigns the latest dialect `visual11.0` to your project. This assignment can cause compilation errors. For more information on the **Dialect** option, see istian.

To avoid the errors, do one of the following:

- After automatic project setup:

  1. Open the project in the user interface. On the **Configuration** pane, select **Target & Compiler**.
  2. Check the **Dialect**. If it is set to `visual11.0`, change it to `visual10`.

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

- Before automatic project setup:

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

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

   to

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

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

Code Verification in Visual Studio

You can apply the powerful code verification functionality of Polyspace software to code that you develop within the Visual Studio Integrated Development Environment (IDE).

A typical workflow is:

1. Use the Visual Studio editor to create a project and develop code within this project.
2. Set up the Polyspace verification by configuring analysis options and settings, and then start the verification.
3. Monitor the verification.
4. Review the verification results.

Before you can verify code in Visual Studio, you must install the Polyspace add-in for Visual Studio. For more information , see “Install Polyspace Add-In for Visual Studio” on page 20-2.

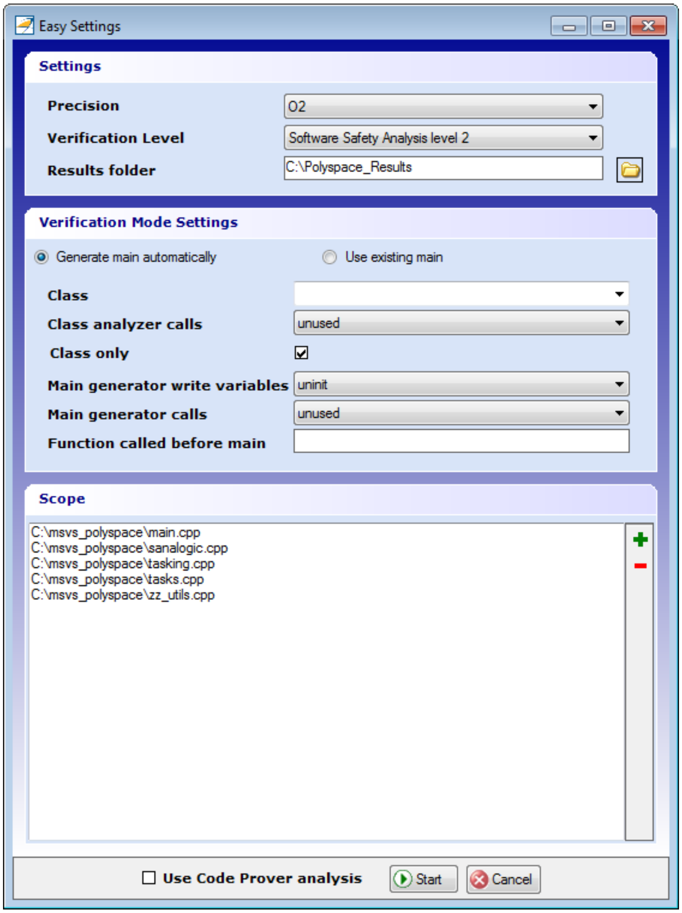
Run Polyspace in Visual Studio

To set up and start a verification:

1 In the **Solution Explorer** view, select one or more files that you want to analyze.

2 Right-click the selection, and select **Polyspace Verification**.

The Easy Settings dialog box opens.



Settings

Precision	O2
Verification Level	Software Safety Analysis level 2
Results folder	C:\Polyspace_Results

Verification Mode Settings

● Generate main automatically ○ Use existing main

Class	
Class analyzer calls	unused
Class only	☑
Main generator write variables	uninit
Main generator calls	unused
Function called before main	

Scope

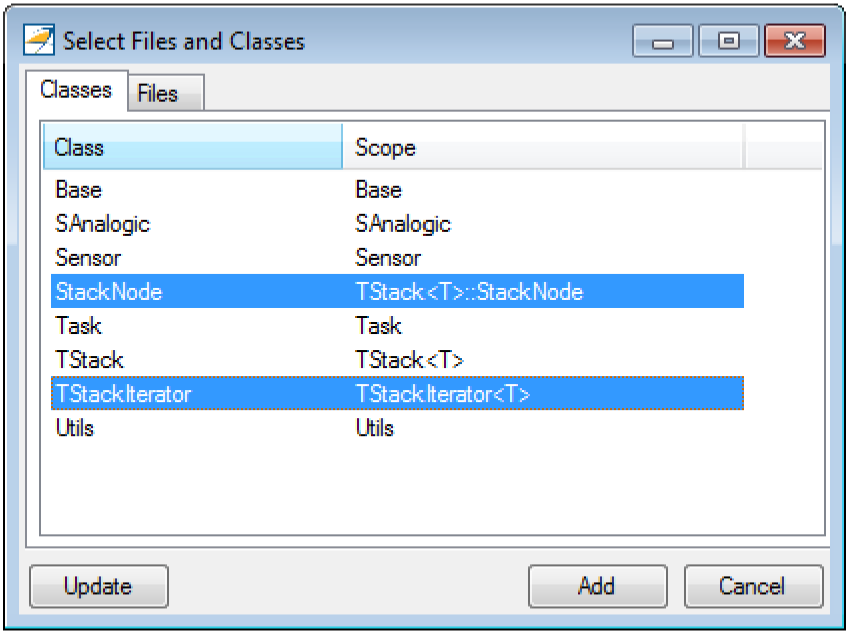
```
C:\msvs_polyspace\main.cpp
C:\msvs_polyspace\sanalogic.cpp
C:\msvs_polyspace\tasking.cpp
C:\msvs_polyspace\tasks.cpp
C:\msvs_polyspace\zz_utils.cpp
```

☐ **Use Code Prover analysis**

3 In the Easy Settings dialog box, you can specify the following options for your verification:

- Under **Settings**, configure the following:
  - **Precision** — Precision of analysis
  - **Passes** — Level of analysis
  - **Results folder** – Location where software stores analysis results
- Under **Verification Mode Settings**, configure the following:
  - **Generate main** — Polyspace generates a `main` or **Use existing** — Polyspace uses an existing main
  - **Class** — Name of class to analyze
  - **Class analyzer calls** — Functions called by generated `main`
  - **Class only** — Analysis of class contents only
  - **Main generator write** — Type of initialization for global variables
  - **Main generator calls** — Functions (not in a class) called by generated `main`
  - **Function called before main** — Function called before the generated `main`
- Under **Scope**, you can modify the list of files and C++ classes to analyze.

  a Select ✚. The Select Files and Classes dialog box opens.



Class	Scope
Base	Base
SAnalogic	SAnalogic
Sensor	Sensor
StackNode	TStack<T>::StackNode
Task	Task
TStack	TStack<T>
TStackIterator	TStackIterator<T>
Utils	Utils

b Select the classes that you want to verify, then click **Add**.

In the Configuration pane in the Polyspace environment, you can configure advanced options not in the Easy Settings dialog box. See “Customize Polyspace Options” on page 20-17.

4 Make sure the **Use Code Prover analysis** check box is selected.

5 Click **Start** to start the analysis.

To follow the progress of an analysis, see “Monitor Progress in Visual Studio” on page 20-18

Configuration File and Default Options

Some options are set by default while others are extracted from the Visual Studio project and stored in the associated Polyspace configuration file.

- The following table shows Visual Studio options that are extracted automatically, and their corresponding Polyspace options:

Visual Studio Option	Polyspace Option
`/D <name>`	`-D` *`<name>`*
`/U <name>`	`-U` *`<name>`*
`/MT`	`-D_MT`
`/MTd`	`-D_MT -D_DEBUG`
`/MD`	`-D_MT -D_DLL`
`/MDd`	`-D_MT -D_DLL -D_DEBUG`
`/MLd`	`-D_DEBUG`
`/Zc:wchar_t`	`-wchar-t-is keyword`
`/Zc:forScope`	`-for-loop-index-scope in`
`/FX`	`-support-FX-option-results`
`/Zp[1,2,4,8,16]`	`-pack-alignment-value [1,2,4,8,16]`

- Source and `include` folders (`-I`) are also extracted automatically from the Visual Studio project.
- Default options passed to the kernel depend on the Visual Studio release: `-dialect Visual7.1` (or `-dialect visual8`) `-OS-target Visual -target i386 -desktop`

Import Visual Studio Project Information into Polyspace Project

You can extract information from a Visual Studio project file (`vcproj`) to configure your Polyspace project.

This Visual Studio import feature can retrieve the following information from a Visual Studio project:

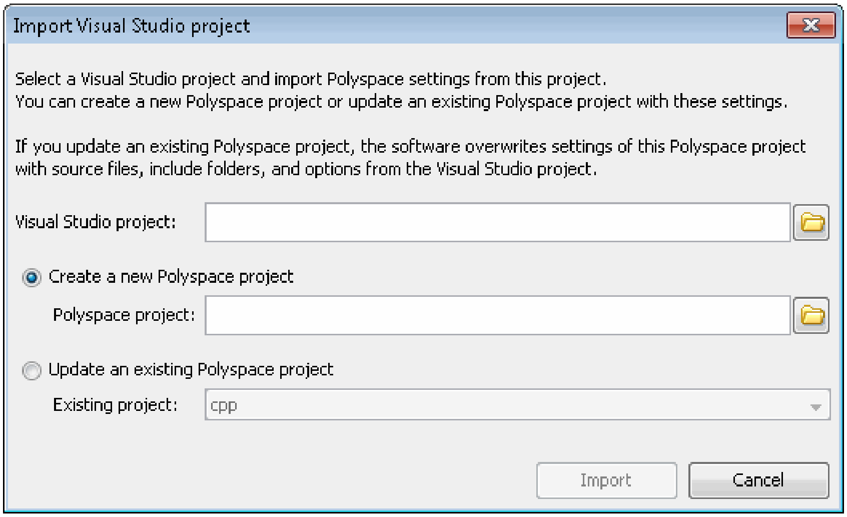
- Source files
- Include folders
- Preprocessing directives (`-D`, `-U`)
- Polyspace specific options about dialect used

Note: This feature supports Visual Studio versions 2008, 2010.

To import Visual Studio information into your Polyspace project:

1 In the Polyspace interface, select **File** > **Import Visual Studio Project**.

The Import Visual Studio project dialog box opens.



Import Visual Studio project

Select a Visual Studio project and import Polyspace settings from this project.
You can create a new Polyspace project or update an existing Polyspace project with these settings.

If you update an existing Polyspace project, the software overwrites settings of this Polyspace project with source files, include folders, and options from the Visual Studio project.

Visual Studio project:

● Create a new Polyspace project

Polyspace project:

○ Update an existing Polyspace project

Existing project: cpp

2 Select the Visual Studio project you want to use.

3 Select the Polyspace project you want to use or create a new project.

4 Click **Import**.

   The Polyspace project is updated with the Visual Studio settings.

When you import a Visual Studio project, if all the source files are C files (with file extension `.c`), then the project will be a C project. Otherwise, the project will be a C++ project.

Customize Polyspace Options

In the Easy Settings dialog box in Visual Studio, you specify only a subset of the Polyspace analysis options.

To customize other analysis options:

1. Select the files you wish to analyze.
2. Right-click on your selection and select **Edit Polyspace Configuration** from the context menu.
3. In the Polyspace Code Prover configuration window, use the different panes to customize your analysis options.

   For more information about specific options, see “Analysis Options for C++ Code”.
4. Save your changes and close the configuration window.

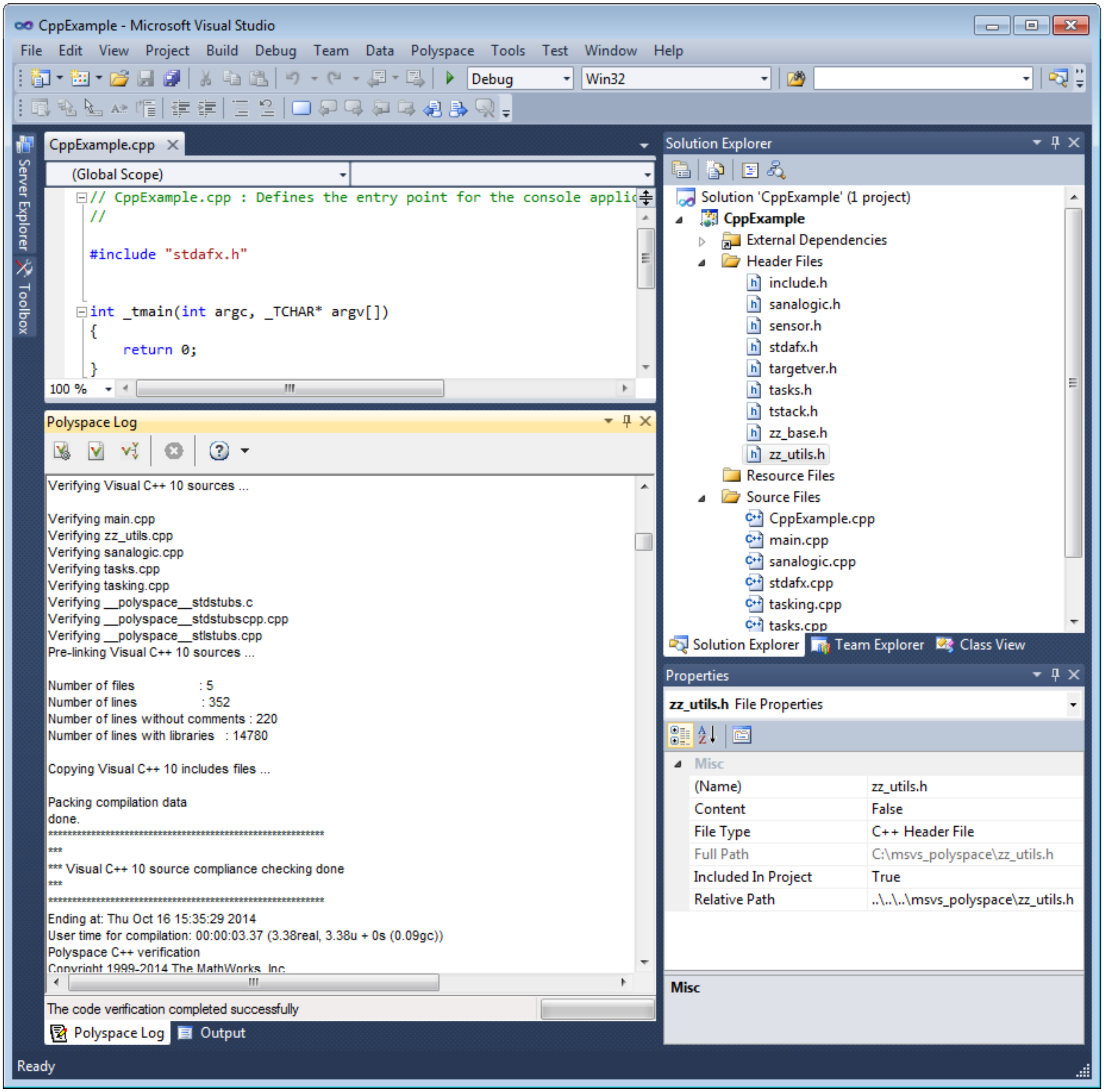
   Next time you run an analysis, Polyspace uses the `*ProjectName*_UserSettings.psprj` settings.

Monitor Progress in Visual Studio

Local Verification

1 Open the **Polyspace Log** view to follow the progress of your verification.

If Polyspace finds compilation issues, the errors are highlighted as links. Click a link to display the file and line that produced the error.



```
// CppExample.cpp : Defines the entry point for the console applic
//

#include "stdafx.h"


int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

```
Verifying Visual C++ 10 sources ...

Verifying main.cpp
Verifying zz_utils.cpp
Verifying sanalogic.cpp
Verifying tasks.cpp
Verifying tasking.cpp
Verifying __polyspace__stdstubs.c
Verifying __polyspace__stdstubscpp.cpp
Verifying __polyspace__stlstubs.cpp
Pre-linking Visual C++ 10 sources ...

Number of files              : 5
Number of lines               : 352
Number of lines without comments : 220
Number of lines with libraries   : 14780

Copying Visual C++ 10 includes files ...

Packing compilation data
done.
*************************************************************
***
*** Visual C++ 10 source compliance checking done
***
*************************************************************
Ending at: Thu Oct 16 15:35:29 2014
User time for compilation: 00:00:03.37 (3.38real, 3.38u + 0s (0.09gc))
Polyspace C++ verification
Copyright 1999-2014 The MathWorks, Inc.
```

The code verification completed successfully

zz_utils.h File Properties

(Name)	zz_utils.h
Content	False
File Type	C++ Header File
Full Path	C:\msvs_polyspace\zz_utils.h
Included In Project	True
Relative Path	..\..\..\msvs_polyspace\zz_utils.h

Misc

2 To stop a verification, on the **Polyspace Log** toolbar, click **X**.

Remote Verification

1 Open the **Polyspace Log** view to follow the progress of your verification.

If Polyspace finds compilation issues, the errors are highlighted as links. Click a link to display the file and line that produced the error.

To stop a verification during the compilation phase, on the **Polyspace Log** toolbar, click **X**.

After compilation, Polyspace sends your verification to the remote server.

2 Select **Polyspace** > **Job Monitor**.

3 In the Polyspace Job Monitor, right-click your project and select **View Log File**

To stop a remote verification after compilation, use the Job Monitor interface.

Related Examples

- “Run Polyspace in Visual Studio” on page 20-11
- “Open Results in Polyspace Environment” on page 20-21

Open Results in Polyspace Environment

After your verification finishes running in Visual Studio, open the Polyspace environment to view your results. If you ran a server verification, download the results before opening the Polyspace environment.

To view your results:

- From the Polyspace Log window, select ✓.
- Select **Polyspace** > **Polyspace**.

  Then, open your results from the Polyspace interface. For instructions, see “Open Results”.

Related Examples

- “Review Results”
- “Run Polyspace in Visual Studio” on page 20-11

Glossary

Atomic

In computer programming, atomic describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible.

Atomicity

In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or no pieces are committed.

Batch mode

Execution of verification from the command line, rather than via the launcher Graphical User Interface.

Category

One of four types of orange check: *potential bug, inconclusive check, data set issue* and *basic imprecision*.

Certain error

See ”red check.”

Check

A test performed during a verification and subsequently colored red, orange, green or gray in the viewer.

Code verification

The Polyspace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.

Dead Code

Code which is inaccessible at execution time under all circumstances due to the logic of the software executed prior to it.

Development Process

The process used within a company to progress through the software development lifecycle.

Green check

Code has been proven to be free of runtime errors.

Gray check

Unreachable code; dead code.

Imprecision

Approximations are made during a verification, so data values possible at execution time are represented by supersets including those values.

mcpu

Micro Controller/Processor Unit

Orange check

A warning that represents a possible error which may be revealed upon further investigation.

Polyspace Approach The manner of using verification to achieve a particular goal, with reference to a collection of techniques and guiding principles.

Precision An verification which includes few inconclusive orange checks is said to be precise

Progress text Output during verification to indicate what proportion of the verification has been completed. Could be considered as a “textual progress bar”.

Red check Code has been proven to contain definite runtime errors (every execution will result in an error).

Review Inspection of the results produced by Polyspace verification.

Scaling option Option applied when an application submitted for verification proves to be bigger or more complex than is practical.

Selectivitiy The ratio (green checks + gray checks + red checks) / (total amount of checks)

Unreachable code Dead code.

Verification The Polyspace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.