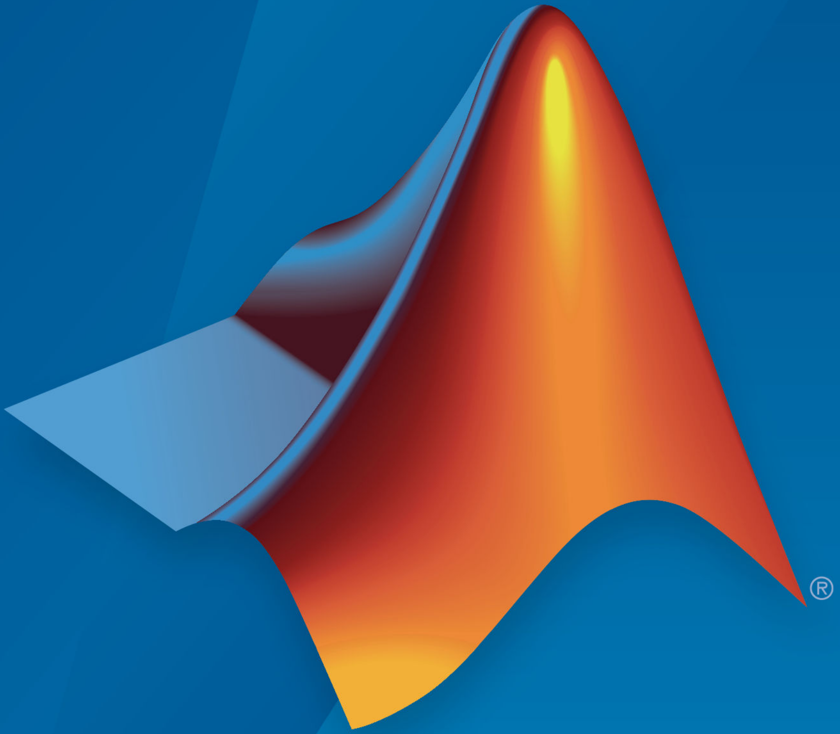
Polyspace® Code Prover™

Getting Started Guide



R2019a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Code Prover™ Getting Started Guide

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)
September 2015	Online Only	Revised for Version 9.4 (Release 2015b)
March 2016	Online Only	Revised for Version 9.5 (Release 2016a)
September 2016	Online Only	Revised for Version 9.6 (Release 2016b)
March 2017	Online Only	Revised for Version 9.7 (Release 2017a)
September 2017	Online Only	Revised for Version 9.8 (Release 2017b)
March 2018	Online Only	Revised for Version 9.9 (Release 2018a)
September 2018	Online Only	Revised for Version 9.10 (Release 2018b)
March 2019	Online Only	Revised for Version 10.0 (Release 2019a)

Contents

4 Polyspace Bug Finder and Polyspace Code Prover

Introduction to Polyspace Code Prover

Polyspace Code Prover Product Description

Prove the absence of run-time errors in software

Polyspace Code Prover™ is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and other run-time errors in C and C ++ source code. It produces results without requiring program execution, code instrumentation, or test cases. Polyspace Code Prover uses semantic analysis and abstract interpretation based on formal methods to verify software interprocedural, control, and data flow behavior. You can use it to verify handwritten code, generated code, or a combination of the two. Each code statement is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

Polyspace Code Prover displays range information for variables and function return values, and can prove which variables exceed specified range limits. Code verification results can be used to track quality metrics and check conformance with your software quality objectives. Polyspace Code Prover can be used with the Eclipse™ IDE to verify code on your desktop.

Support for industry standards is available through IEC Certification Kit (for IEC 61508 and ISO 26262) and DO Qualification Kit (for DO-178).

Polyspace Products for Code Analysis and Verification

In this section...

Polyspace products use static analysis to check code for run-time errors, coding standard violations, security vulnerabilities and other issues. A static analysis tool such as Polyspace Code Prover can cover all possible execution paths through a program and track data flow along these paths following certain mathematical rules. The analysis can be much deeper than dynamic testing and expose potential run-time errors that might not be otherwise found in regular software testing. A static analysis tool such as Polyspace Bug Finder™ can scan a program quickly for more obvious run-time errors and coding constructs that potentially lead to run-time errors or unexpected results.

A software development process can use analysis results from Polyspace to complement dynamic testing. Using a product such as Code Prover, you can drastically reduce efforts in dynamic testing and focus only on areas where static analysis is unable to prove the absence of a run-time error. Using a product such as Bug Finder, you can maintain a list of potentially problematic coding practices and automatically check for these practices during development.

Using Polyspace Products in Software Development

The Polyspace suite of products supports all phases of a software development process:

- *Prior to code submission*:

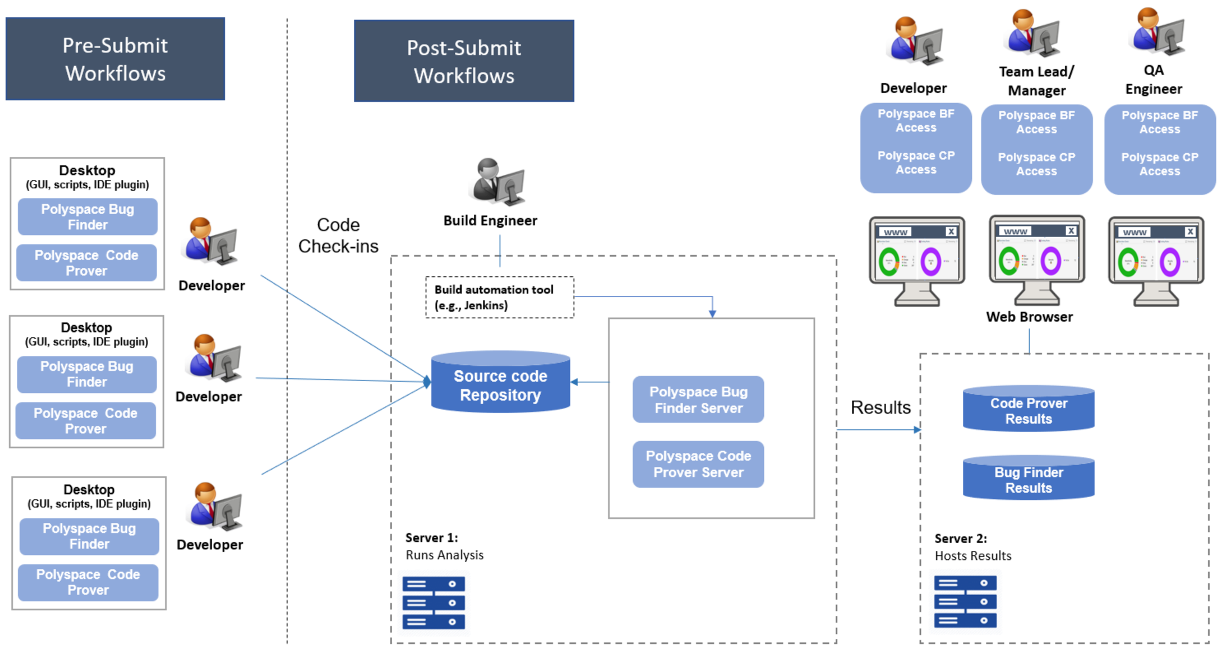
  Developers can run the Polyspace desktop products to check their code during development or right before submission to meet predefined quality goals.

  The desktop products can be integrated into IDEs such as Eclipse or run with scripts during compilation. The analysis results can be reviewed in IDEs such as Eclipse or in the graphical user interface of the desktop products.

- *After code submission*:

  The Polyspace server products can run automatically on newly committed code as a build step in a continuous integration process (using tools such as Jenkins). The analysis runs on a server and the results are uploaded to a web interface for collaborative review.

See "Polyspace Products for C/C++ Code" on page 1-4.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

When you use both the desktop and server products, your pre-submission workflow can transition smoothly to the post-submission workflow. See "Using Desktop and Server Products Together" on page 1-5.

Polyspace Products for C/C++ Code

Polyspace provides these products for desktop usage:

- **Polyspace Bug Finder** to check code for semantic errors that a compiler cannot detect (such as use of = instead of ==), concurrency issues, security vulnerabilities and other defects in C and C++ source code. The analysis can also detect some run-time errors.
- **Polyspace Code Prover** to perform a much deeper check and prove absence of overflow, divide-by-zero, out-of-bounds array access and other run-time errors in C and C++ source code.

Depending on your quality goals, you can run one or both products. See "Choose Between Polyspace Bug Finder and Polyspace Code Prover" on page 4-2.

Polyspace provides these products for server usage:

- **Polyspace Bug Finder Server™** to run Bug Finder automatically on a server and upload the results to a web interface for review, and **Polyspace Bug Finder Access™** to review the uploaded results.

  Typically, Polyspace Bug Finder Server runs on a few build servers and checks newly committed code as part of software build and testing. Each reviewer (developer, quality assurance engineer or development manager) has a Polyspace Bug Finder Access license to review the uploaded analysis results.
- **Polyspace Code Prover Server** to run Code Prover automatically on a server and upload the results to a web interface for review, and **Polyspace Code Prover Access** to review the uploaded results.

  Typically, Polyspace Code Prover Server runs on a few build servers and checks newly committed code as part of software build and testing. Each reviewer (developer, quality assurance engineer or development manager) has a Polyspace Code Prover Access license to review the uploaded analysis results.

Using Desktop and Server Products Together

In a software development workflow, you benefit most from using the desktop and server products together. Developers can run the desktop products while coding and fix or justify the issues found. At this stage, it is easy to rework the code because it is still under development.

After code submission, the server products can run a more comprehensive analysis. The analysis will reveal fewer issues if the developer has already fixed them before submission. If the developer has triaged issues for fixing later or justified them, this information can be carried over to the server-side analysis so that fewer results need to be reviewed. The remaining results can be uploaded to the Polyspace Access web interface. Quality engineers can review these results and based on the severity of the results, assign them to developers for fixing.

The desktop and server products can be coordinated in these ways:

- You can use the same analysis configuration with both the desktop and server products. If you use the same analysis configuration, you see the same analysis results on the desktop and server side.

  At the same time, you can perform a more comprehensive checking on the server side. If you are running Bug Finder, you can increase the number of checkers for the server-side analysis compared to the desktop-side analysis. If you are running Code Prover, you can use stricter assumptions for the server-side analysis compared to the desktop-side analysis.

  The server side analysis can also run on more complete applications as opposed to the desktop side analysis, which runs on individual modules.
- If you enter comments on the desktop side to justify an analysis result, these comments can be reused on the server side. If you justify analysis results on the desktop side or set a status on them for fixing later, you are saved from repeating this work for the server-side analysis results.

  If you enter the comments in your source code as code annotations using a specific syntax, the server-side analysis can read the code annotations and import them to the server-side analysis results.
- You can configure your analysis on the desktop but run the analysis on a dedicated server. In this workflow:

  - You perform a one-time setup to enable communication between the desktop and server products using the distributed computing product, **MATLAB® Parallel Server™** .
  - During development, you trigger the analysis from a desktop product but the analysis runs on a server using a server product. The results are downloaded back to the desktop product for review.

  Since the analysis is offloaded to a server, this workflow saves processing power on the developer's desktop.

Polyspace Products for Ada Code

Polyspace provides these products for verifying Ada code:

- **Polyspace Client™ for Ada** to check Ada code for run-time errors on a desktop.
- **Polyspace Server for Ada** to check Ada code for run-time errors on a server.

You can either use the desktop product to run the analysis on your desktop, or a combination of the desktop and server products to run the analysis on a server. The analysis results are downloaded to your desktop for review.

If you have a Polyspace Code Prover Access license and have set up the web interface of Polyspace Code Prover Access, you can upload each individual Ada result from the Ada desktop products to the web interface for collaborative review.

See also:

- https://www.mathworks.com/products/polyspace-ada.html
- https://www.mathworks.com/products/polyspaceserverada/

See Also

Related Examples

- “Install Polyspace Desktop Products” (Polyspace Bug Finder)
- “Install Polyspace Server and Access Products” (Polyspace Bug Finder Server)

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 1-9
- "Decrease Development Time" on page 1-9
- "Improve the Development Process" on page 1-10

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).

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

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.

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.

Install Polyspace Desktop Products

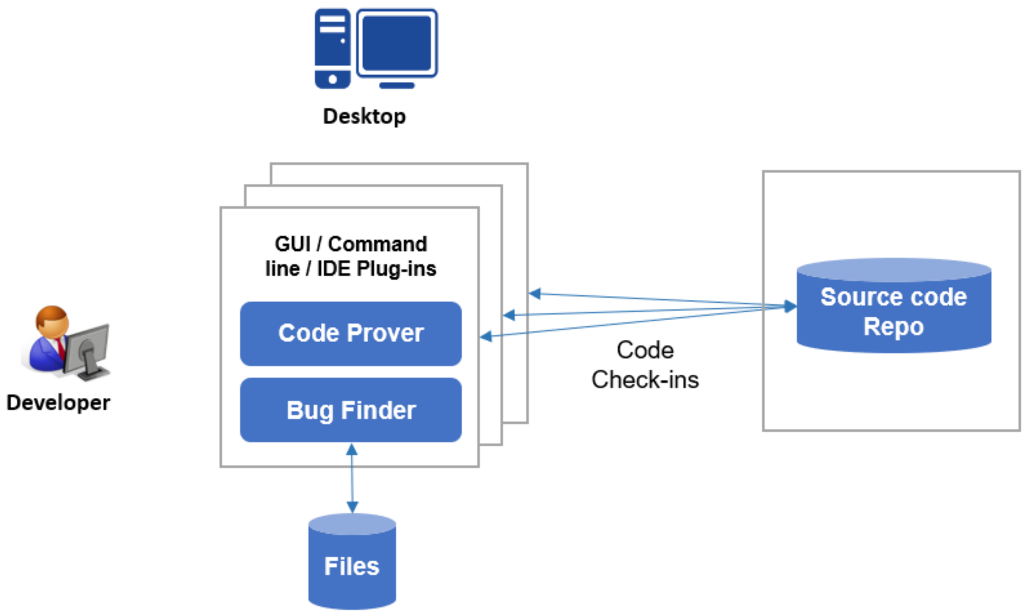
Polyspace checks C/C++ code for bugs, run-time errors, coding standard violations and other issues using static analysis. With the desktop products (Polyspace Bug Finder and Polyspace Code Prover), you can perform the checks on individual desktops prior to code submission.

For an overview of all Polyspace products, see “Polyspace Products for Code Analysis and Verification” on page 1-3.

Workflow

Using the Polyspace desktop products, individual developers can check their code for bugs and run-time errors during development.

For uniform standards across a project or team, all developers in the project or team can use a predefined set of checks. Developers can qualify their code for submission based on these predefined checks. After code submission to a shared repository, a more extensive post-submission analysis can run on a server using the Polyspace server products.



The workflow consists of two steps:

- *Running Polyspace analysis*:

  During development, individual developers start the analysis from their IDE-s, using scripts or from the user interface of the desktop products.

  For code generated from:

  - Simulink® models, the analysis can be started directly from Simulink after code generation.
  - MATLAB code, the analysis can be started directly in the MATLAB Coder App after code generation.

  To save processing power on the developer's desktop, the analysis can also be offloaded to a server and the results downloaded to the desktop for review.
- *Reviewing Polyspace results*:

After analysis, developers review the results (bugs, run-time errors, coding standard violations, and so on) in the user interface of the desktop products.

If using Eclipse or an Eclipse-based IDE, developers can review the results directly in the IDE.

Note that these steps describe a workflow prior to code submission. After code submission, a build automation tool can run a Polyspace analysis on a server. The analysis results can be uploaded to a web browser for collaborative review by developers or quality engineers. See “Install Polyspace Server and Access Products” (Polyspace Code Prover Server).

Product Installation

For this workflow, you must install the following on individual desktops.

Polyspace Products to Run Analysis

Install Polyspace Bug Finder and/or Polyspace Code Prover to run the analysis.

Installation

Run the MathWorks® installer. Choose a license for Polyspace desktop products. You can get the installer and license by purchasing the product or requesting a trial. For detailed instructions, see “Installation, Licensing, and Activation”.

Note that you require Polyspace Bug Finder to install Polyspace Code Prover.

After you install a Polyspace desktop product, you can open the Polyspace user interface or run command-line executables. You can start an analysis in the user interface or from the Windows® or Linux® command line. To start the analysis from other environments, you have to perform additional steps:

- To run Polyspace from Eclipse or an Eclipse-based IDE, install the Polyspace plugin. See “Install Polyspace Plugin for Eclipse” on page 2-2.
- To run Polyspace with MATLAB scripts, install MATLAB. Then, perform a one-time setup to link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

- To run Polyspace from Simulink, install MATLAB, Simulink, and Embedded Coder® (and its prerequisite products). Then, perform a one-time setup to link your Polyspace and Simulink installations. See "Integrate Polyspace with MATLAB and Simulink".
- To run Polyspace from the MATLAB Coder App, install MATLAB and Embedded Coder (and its prerequisite products). Then, perform a one-time setup to link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".
- To offload the analysis to a server, install Polyspace Bug Finder only on your desktop. On the server side, install the Polyspace server products and MATLAB Parallel Server to handle analysis jobs from multiple desktops. See "Install Products for Submitting Polyspace Analysis from Desktops to Remote Server" on page 2-5.

Polyspace Products to Review Results

The Polyspace Bug Finder and/or Polyspace Code Prover installation is sufficient to review the results.

You can review both Bug Finder and Code Prover results with only a Bug Finder desktop license. For instance, if you offload the analysis to a server and only review the downloaded analysis results on your desktop, you require only a Bug Finder license.

In Eclipse or Eclipse-based IDEs, if you install the Polyspace plugin, you can see the results directly in the IDE.

Install Polyspace with Other MathWorks Products

If you install Polyspace with other MathWorks products such as MATLAB, you have to run the MathWorks installer twice.

- In the first run, choose the license that corresponds to the other MathWorks products, such as MATLAB, Simulink or Embedded Coder.
- In the second run, choose the license that corresponds to the Polyspace products.

In this workflow, products such as MATLAB and Simulink are installed in a different root folder from the Polyspace products. However, you can link the two installations and use MATLAB scripts to run Polyspace. See "Integrate Polyspace with MATLAB and Simulink".

If you install both the Polyspace desktop and server products, you also have to run the installer twice with separate licenses. The desktop and server products are installed in separate root folders. For instance, in Windows, the default root folders for an R2019a installation are:

- Polyspace desktop products: `C:\Program Files\Polyspace\R2019a`.

  This folder contains executables to run analysis with the products, Polyspace Bug Finder and/or Polyspace Code Prover.
- Polyspace server products: `C:\Program Files\Polyspace Server\R2019a`.

  This folder contains executables to run analysis with the products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server.

See Also

More About

- “Run Polyspace Code Prover on C/C++ Code” on page 3-3
- “Review Polyspace Code Prover Analysis Results” on page 3-11
- “Install Polyspace Server and Access Products” (Polyspace Code Prover Server)

Install Polyspace Code Prover

Install Polyspace Plugin for Eclipse

This topic shows how to install or uninstall the Polyspace plugin for Eclipse.

Install Polyspace Plugin for Eclipse IDE

The Polyspace plugin is supported for Eclipse versions 4.7 to 4.9. You can install the Polyspace plugin only after you:

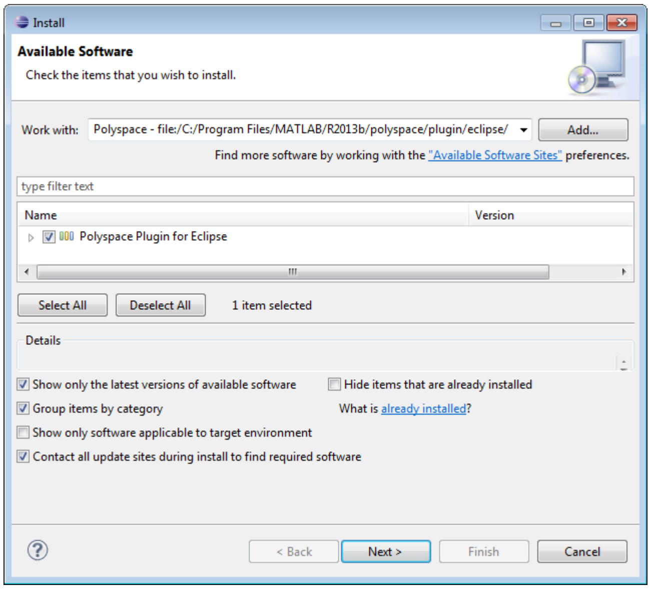
- Install and set up Eclipse Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java® 8 or newer. See Java documentation at www.java.com.

  If you run into issues because of incompatible Java versions, see “Eclipse Java Version Incompatible with Polyspace Plug-in”.
- Uninstall any previous Polyspace plugins. For more information, see “Uninstall Polyspace Plugin for Eclipse IDE” on page 2-4.

To install the Polyspace plugin:

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/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☑ Polyspace Plugin for Eclipse	

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

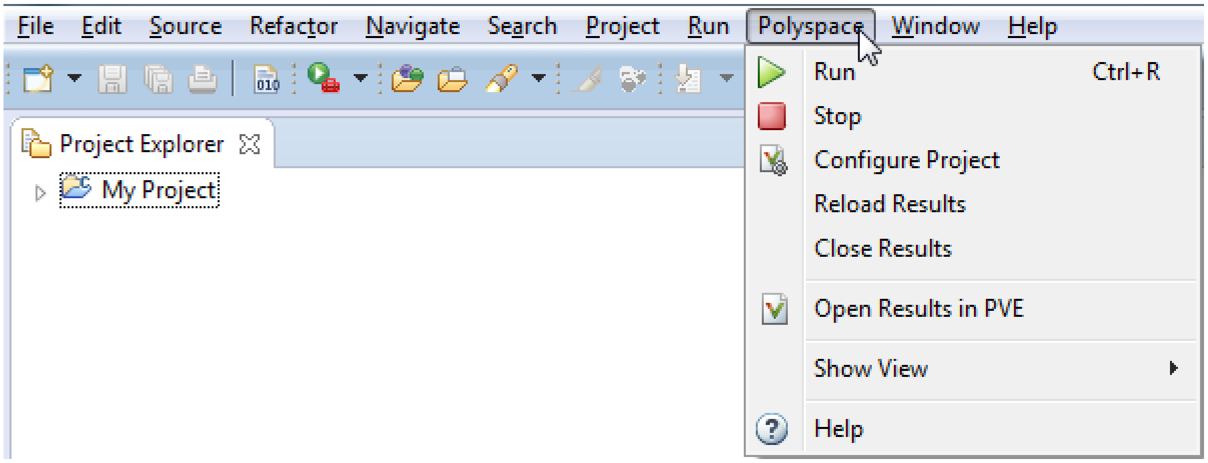
8 Click **Next**.

9 On the Install Details page, click **Next**.

10 On the Review Licenses page, review and accept the license agreement. Then click **Finish**.

Once you install the plugin, in the Eclipse editor, you’ll see:

- A **Polyspace** menu
- A **Polyspace Run - Code Prover**, **Results List - Code Prover**, and **Result Details** view.



Uninstall Polyspace Plugin for Eclipse IDE

Before installing a new Polyspace plugin, you must uninstall any previous Polyspace plugins:

1. In Eclipse, select **Help > About Eclipse**.
2. Select **Installation Details**.
3. Select the Polyspace plugin and select **Uninstall**.

   Follow the uninstall wizard to remove the Polyspace plugin. You must restart Eclipse for changes to take effect.

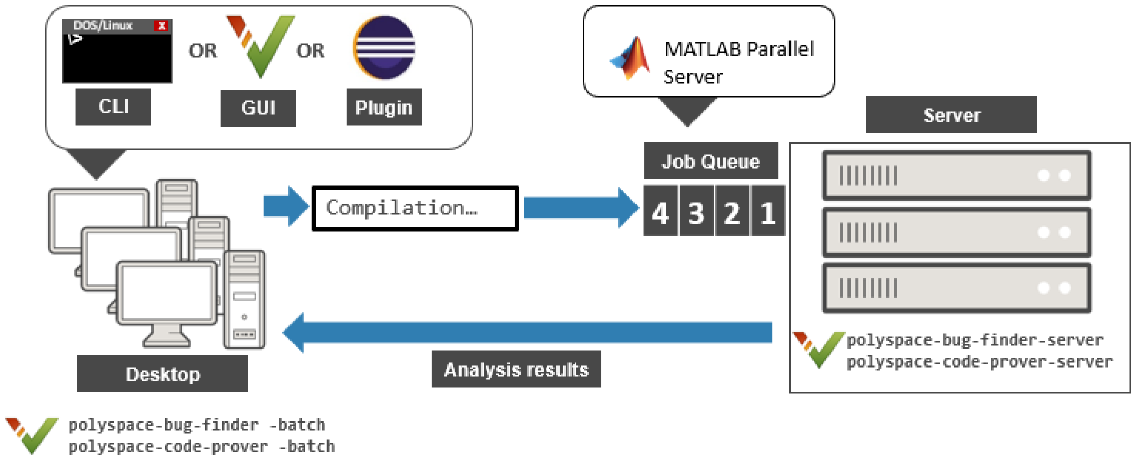
See Also

More About

- “Run Polyspace Analysis in Eclipse”

Install Products for Submitting Polyspace Analysis from Desktops to Remote Server

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated remote servers. This topic shows how to set up the dispatch of Polyspace analysis from desktop clients to remote servers. Once configured, you can send the Polyspace analysis to a remote server and view the downloaded results on your desktop.


DOS/Linux
CLI
OR
GUI
OR
Plugin
MATLAB Parallel Server
Server
Job Queue
4 3 2 1
Compilation…
Desktop
Analysis results
polyspace-bug-finder-server
polyspace-code-prover-server
polyspace-bug-finder -batch
polyspace-code-prover -batch

Choose Between Local and Remote Analysis

To determine when to use local or remote analysis, use the rules listed in this table.

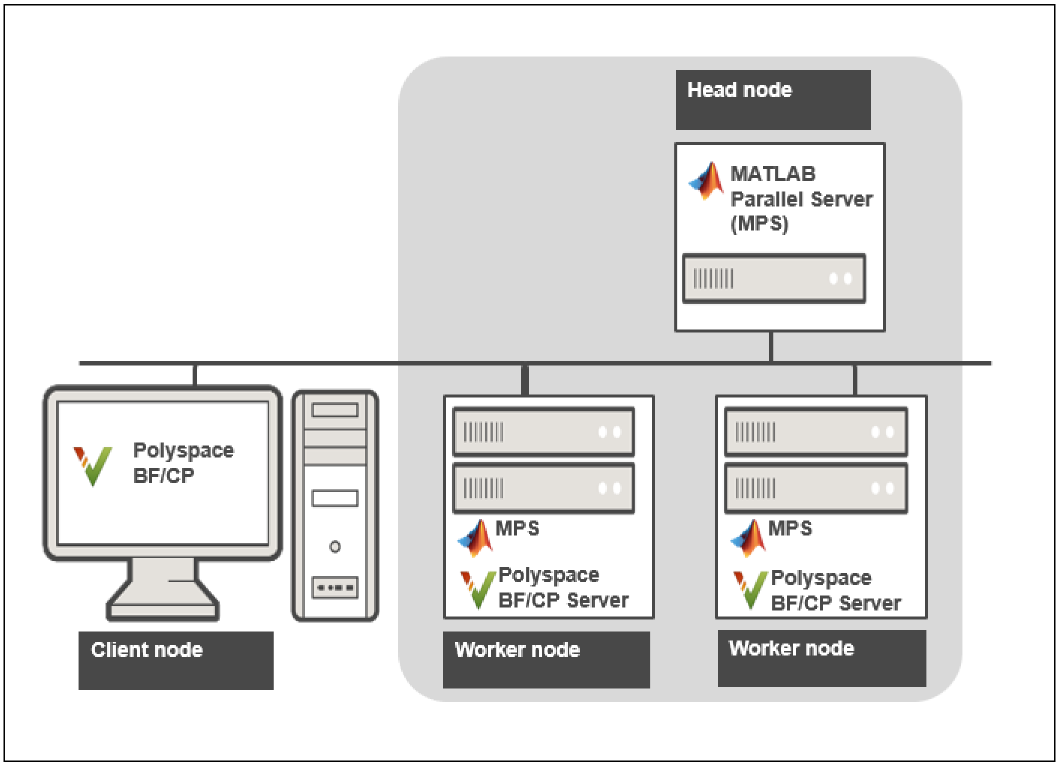
Type	When to Use
Remote	Source files are large and execution time of analysis is lengthy. Typically, a Code Prover analysis takes significantly longer than a Bug Finder analysis and benefits from running on a dedicated server.
Local	Source files are small and execution time of analysis is short.

Requirements for Remote Analysis

A typical distributed network for running remote analysis consists of these parts:

- **Client nodes**: On the client node, you configure your Polyspace project or scripts, and then submit a job that runs Polyspace.
- **Head node**: The head node distributes the submitted jobs to worker nodes.
- **Worker node(s)**: The Polyspace analysis runs on a worker node.

In this workflow, you install the product MATLAB Parallel Server to manage submissions from multiple clients. An analysis job is created for each submission and placed in a queue. As soon as a worker node is available, the next analysis job from the queue is run on the worker.



In the simplest remote analysis configuration, the same computer can serve as the head node and worker node. Note that you can run one Polyspace analysis on one worker only. You cannot distribute the analysis over multiple workers. Only if you submit more than one analysis job, you can distribute the jobs over multiple workers.

This table lists the product requirements for remote analysis.

Location	Requirements	Installation
Client node	Polyspace Bug Finder A Polyspace Bug Finder license is sufficient to trigger a Bug Finder or Code Prover analysis on the server, and review the downloaded analysis results.	Run the MathWorks installer on the client desktops. Choose a license for Polyspace desktop products. For detailed instructions, see “Installation, Licensing, and Activation”.
Head node	MATLAB Parallel Server (earlier called MATLAB Distributed Computing Server)	Run the MathWorks installer on the server(s). Choose a license for MATLAB Parallel Server installation. For detailed instructions, see “Integrate MATLAB Job Scheduler for Network License Manager” (MATLAB Parallel Server).
Worker nodes	• MATLAB Parallel Server (earlier called MATLAB Distributed Computing Server) • Polyspace Bug Finder Server • Polyspace Code Prover Server (if you choose to run Code Prover)	To install: • MATLAB Parallel Server, run the MathWorks installer on the server(s). Choose a license for MATLAB Parallel Server installation. • Polyspace Bug Finder Server and/or Polyspace Code Prover Server, run the MathWorks installer. Choose a license for Polyspace server products.

Configure and Start Server

On the computers that act as the worker nodes of the server, you install MATLAB Parallel Server and the Polyspace server products in two separate folders. The MATLAB Parallel Server installation must know where the Polyspace server products are located so that it can route the Polyspace analysis. To link the two installations, specify the paths to the root folder of the Polyspace server products in your MATLAB Parallel Server installations.

Then configure and start MATLAB Parallel Server (the `mjs` service) on all computers that act as the head node and worker nodes.

Configure mjs Service Settings

Before starting services, you must configure the `mjs` service settings.

1. Navigate to *`matlabroot`*`\toolbox\distcomp\bin`, where *`matlabroot`* is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files \MATLAB\R2019a`.
2. Modify the file `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux). To edit and save the file, you have to open your editor in administrator mode.

   Read the instructions in the file and uncomment the lines as needed. At a minimum, you might have to uncomment these lines:

   - Hostname:

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

     in Windows or

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

     in Linux. Explicitly specify your computer host name.
   - Security level:

     ```
     REM set SECURITY_LEVEL=
     ```

     in Windows or

```
#SECURITY_LEVEL=""
```

in Linux. Explicitly specify a security level.

Otherwise, you might see an error later when starting the job scheduler.

Specify Polyspace Installation Paths

When you offload an analysis using a Polyspace desktop product installation, the server must run the analysis using a Polyspace server product installation from the same release. For instance, if you offload an analysis from an R2019a desktop product, the analysis must run using the R2019a server product. To ensure that the correct Polyspace server product is used, you must specify the installation paths of the Polyspace server products in your MATLAB Parallel Server installations.

If you use multiple releases of Polyspace desktop and server products, the MATLAB Parallel Server release must be the later one. For instance, if you offload analysis jobs using both R2019a and R2019b Polyspace desktop and server products, the MATLAB Parallel Server installation must be an R2019b one.

To specify the Polyspace installation paths, navigate to `*matlabroot*\toolbox\distcomp\bin\`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2019a`. Then, depending on the releases of the Polyspace products that run the analysis, use one of these methods:

- **Specify Paths to Polyspace Releases R2019a and Later**

  Uncomment and modify the following line in the file `mjs_polyspace.conf`. To edit and save the file, you have to open your editor in administrator mode.

  ```
  POLYSPACE_SERVER_R#####_ROOT=polyspaceserverroot
  ```

  Here, `R#####` is the release number, for instance, `R2019a`, and *`polyspaceserverroot`* is the installation path of the server products, for instance:

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

  To specify multiple releases, add a declaration for each release. For instance, the specification for an R2019a and R2019b Windows installation of the server products can be like this:

```
POLYSPACE_SERVER_R2019A_ROOT=C:\Program Files\Polyspace Server\R2019a
POLYSPACE_SERVER_R2019B_ROOT=C:\Program Files\Polyspace Server\R2019b
```

- **Specify Paths to Polyspace Releases Prior to R2019a**

  Prior to R2019a, you need to specify a Polyspace release only if you want MATLAB Parallel Server to handle submissions from clients with multiple releases of Polyspace. You need to specify only the earlier release of Polyspace. For instance, if an R2018b installation of MATLAB Parallel Server needs to handle analysis jobs from both an R2018b and R2018a installation, specify only the path to the R2018a installation.

  Edit the file `mjs_def.bat` or `mjs_def.sh` (located in *`matlabroot`*`\toolbox\distcomp\bin\`) to refer to the earlier release. Find the line with `MJS_ADDITIONAL_MATLABROOTS` and edit it as follows. To edit and save the file, you have to open your editor in administrator mode.

  ```
  set MJS_ADDITIONAL_MATLABROOTS=othermatlabroot
  ```

  Here, *`othermatlabroot`* is the installation path of the Polyspace products from the earlier release, for instance:

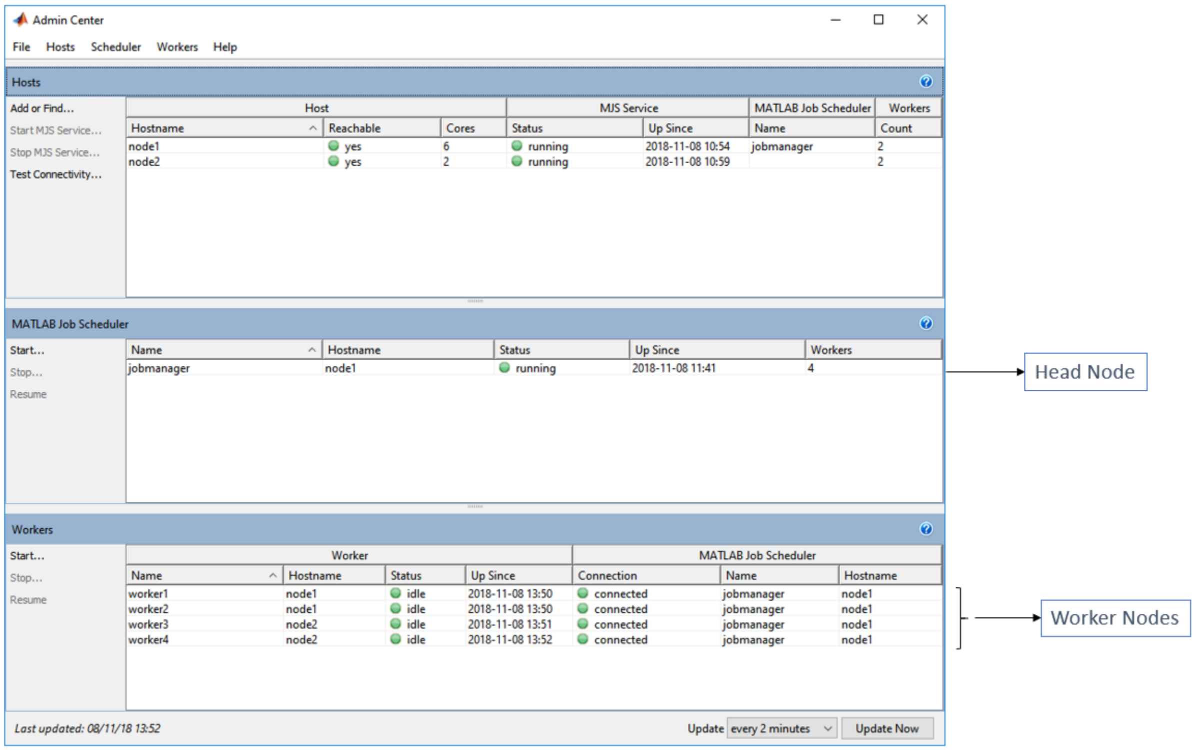
  ```
  C:\Program Files\MATLAB\R2018a
  ```

Start mjs Service and Assign as Head Node or Worker Node

To configure a server with multiple workers, start the service that runs a job scheduler (the `mjs` service) on the computer that acts as the head node and all computers that act as worker nodes. In the simplest configuration, the same computer can act as the head node and a worker node.

To set up a cluster with one head node and several workers, on the computer that acts as the head node:

1. Open the **Admin Center** window. Navigate to *`matlabroot`*`/toolbox/distcomp/bin` and execute the file `admincenter.bat` (Windows) or `admincenter.sh` (Linux). Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2019a`.



Hosts

Host			MJS Service		MATLAB Job Scheduler	Workers
Hostname	Reachable	Cores	Status	Up Since	Name	Count
node1	● yes	6	● running	2018-11-08 10:54	jobmanager	2
node2	● yes	2	● running	2018-11-08 10:59		2

MATLAB Job Scheduler

Name	Hostname	Status	Up Since	Workers
jobmanager	node1	● running	2018-11-08 11:41	4

Workers

Worker				MATLAB Job Scheduler		
Name	Hostname	Status	Up Since	Connection	Name	Hostname
worker1	node1	● idle	2018-11-08 13:50	● connected	jobmanager	node1
worker2	node1	● idle	2018-11-08 13:50	● connected	jobmanager	node1
worker3	node2	● idle	2018-11-08 13:51	● connected	jobmanager	node1
worker4	node2	● idle	2018-11-08 13:52	● connected	jobmanager	node1

Last updated: 08/11/18 13:52

Update: every 2 minutes

2 In the **Hosts** section, add the host names of all computers that you want to use as head and worker nodes of the cluster. Start the `mjs` service.

   The service uses the settings specified in the file `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux).

3 Right-click each host. Select either **Start MJS** (head node) or **Start Workers** (worker nodes).

   The hosts appear in the **MATLAB Job Scheduler** or **Workers** section. In each section, select the host and click **Start** to start the MATLAB Job Scheduler or the workers.

Selecting a computer as host starts the `mjs` service on that computer. You must have permission to start services on other computers in the network. For instance, on Windows, you must be in the Administrators group for other computers where you want to start the `mjs` service. Otherwise, you have to start the `mjs` services individually on each computer that acts as a worker.

For more details and command-line workflows, see:

- "Integrate MATLAB Job Scheduler for Network License Manager" (MATLAB Parallel Server)
- `mjs`

Configure Client

Configure the client node so that it can communicate with the computer that serves as the head node of the MATLAB Parallel Server cluster.

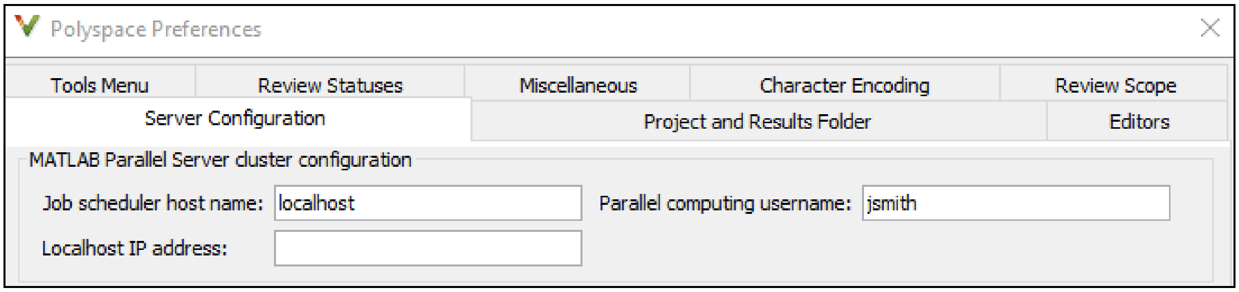
Configure the client node through the Polyspace environment preferences:

1. Select **Tools** > **Preferences**.
2. Click the **Server Configuration** tab. Under **MATLAB Parallel Server cluster configuration**:

   a. In the **Job scheduler host name** field, specify the computer for the head node of the cluster. This computer hosts the MATLAB job scheduler.

      If the port used on the computer hosting the MATLAB job scheduler is different from 27350, enter the port name explicitly with the notation *`hostName`*`:`*`portNumber`*.

   b. Due to the network setting, the job scheduler may be unable to connect back to your local computer. If so, enter the IP address of the client computer in the **Localhost IP address** field.



MATLAB Parallel Server cluster configuration

Job scheduler host name: localhost

Parallel computing username: jsmith

Localhost IP address:

Offload Polyspace Analysis from Desktop to Server

Once the configuration is over, you can offload an analysis from a Polyspace desktop product installation to a remote server. You can do one of the following:

- Start a remote analysis from the user interface of the Polyspace desktop products.

  See “Send Polyspace Analysis from Desktop to Remote Servers”.
- Start a remote analysis with Windows or Linux scripts.

  See “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”. In the simplest configuration, the same computer can be used as a client and server. For a simple tutorial that uses this configuration and walks through all the steps for offloading a Polyspace analysis, see “Send Code Prover Analysis from Desktop to Locally Hosted Server” on page 3-17.
- Start a remote analysis with MATLAB scripts.

  See “Run Analysis on Server”.

See Also

Related Examples

- “Set Up Polyspace Metrics” on page 2-16

- “Send Polyspace Analysis from Desktop to Remote Servers”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”
- “Send Code Prover Analysis from Desktop to Locally Hosted Server” on page 3-17
- “Job Manager Cannot Write to Database”
- “Integrate MATLAB with Third-Party Schedulers” (MATLAB Parallel Server)
- “Troubleshoot Common Problems” (MATLAB Parallel Server)

Set Up Polyspace Metrics

Note For easier collaborative reviews, use Polyspace Code Prover Access . In addition to a more intuitive web dashboard, with Polyspace Access you can:

- Review and justify results directly from your web browser.
- Integrate a defect-tracking tool such as Jira with the web interface and create tickets to track Polyspace findings.
- Share analysis results using web links.

For more information, see the Polyspace Code Prover Access documentation.

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.

This topic shows how to set up a Polyspace Metrics server to store Polyspace results.

Requirements for Polyspace Metrics

You can use Polyspace Metrics to:

- Store Polyspace results.
- Evaluate and monitor software quality metrics based on those results.

You require a computer that acts as a server and hosts the Polyspace Metrics interface. Results from several client desktops can be uploaded to the Polyspace Metrics interface.

This table lists the requirements for Polyspace Metrics.

Location	Task	Requirements
Client desktops	The client desktops: • Run Polyspace and upload results to the server. • Download results from the server for detailed review.	You must install Polyspace Bug Finder and/or Polyspace Code Prover.
Server	The server: • Runs Polyspace Metrics service. • Hosts results uploaded from server and computes and displays quality metrics. You can load the server address and view the metrics.	You must install Polyspace Bug Finder and/or Polyspace Code Prover. However, you do not require activation to run the Polyspace Metrics service.

You cannot merge two different Polyspace metrics databases. However, if you install a newer version of Polyspace on top of an older version, Polyspace Metrics automatically updates the database to the newest version.

Configure and Start Polyspace Metrics Server

This section shows you how to start the host server for Polyspace Metrics. After you complete this step, you must also configure the client side settings so that the user interface on the Polyspace desktops can interact with the Metrics server.

1. In the Polyspace user interface, select **Tools** > **Metrics Server Settings**.

   Alternatively, run the following command:

   ```
   polyspaceroot\polyspace\bin\polyspace-server-settings.exe
   ```

   Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.
2. Under **Polyspace Metrics Settings**, specify this information:

   - **User name used to start the service** — Your user name.
   - **Password** — Your password (Windows only).
   - **Communication port** — Polyspace communication port number (default 12427). This number must be the same as the communication port number specified in the Polyspace Interface preferences. See "Configure Client Side" on page 2-19.
   - **Folder where analysis data will be stored** — Results repository for Polyspace Metrics server.

   If you want to start Polyspace Metrics as a service, select **Install as service**. If you select this option, the Polyspace Metrics service starts automatically each time you restart the computer. You do not have to start the Metrics service explicitly. However, when you use the option, starting the server might require additional privileges, for instance, root privileges in Linux.
3. To start the Polyspace Metrics server, click **Start Server**.

The software stores the information that you specify through the Metrics Server Settings window in the following file:

- On a Windows system, `\%APPDATA%\Polyspace_RLDatas\polyspace.conf \polyspace.conf`.
- On a Linux system, `/etc/Polyspace/polyspace.conf`

You can edit this file directly for specific purposes. For instance, Polyspace Metrics uses Tomcat 8.0.22 to run the Metrics user interface. To specify your own version of Tomcat, add the following line to this file:

```
tomcat_install_dir = tomcat_path
```

Here, *`tomcat_path`* is the path to your Tomcat installation (on Windows, it is the value of the environment variable `CATALINA_HOME`).

To start Polyspace Metrics web server at the command line, use one of these commands:

- Windows: `perl` *`polyspaceroot`*`\toolbox\polyspace\psdistcomp\bin\setup-polyspace-cluster.pl`
- Linux: `./`*`polyspaceroot`*`/toolbox/polyspace/psdistcomp/bin/setup-polyspace-cluster`

Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`. For more help in using the commands, use the `-h` option.

Configure Client Side

Once you have set up your Polyspace metrics server, you must set the client-side settings so that the Polyspace interface can communicate with your Metrics server.

1. Select **Tools** > **Preferences**.
2. Click the **Server Configuration** tab.
3. Select **Use Polyspace Metrics**.

   Specify this information:

   a. If you want Polyspace to detect a server on the network that uses port 12427 (default port number), click **Automatically detect the Polyspace Metrics Server**.
   b. If you use a different port number for your Metrics server or you want to specify the server name, click **Use the following server and port**. Fill in your server name or IP address, and communication port number.

      You must specify the same communication port number for all clients that use the Polyspace Metrics service.
4. Under the **Polyspace Metrics web interface configuration** section:

   a. Specify a **Port used to download results**, default is 12428. If you change this port number, you must also change it in on the server side.

b. Specify which protocol to use HTTP or HTTPS. If you select HTTPS for your web protocol, there are additional steps to set up the Metrics web server for HTTPS on page 2-20.
c. Specify a web server port number for your chosen protocol. Default port numbers are:

   - HTTP — 8080
   - HTTPS — 8443

   If you change the port number from the default, you must configure the same port number for the Polyspace Metrics server. See “Configure and Start Polyspace Metrics Server” on page 2-18.

5. Under the **Upload and download settings** section:

   - Upload settings — After you review results from the Metrics repository, you can upload your comments and justifications back to the repository using **Metrics > Upload to Metrics**.

     If you want Polyspace to automatically upload your justifications to Polyspace Metrics when you save, select **Upload justifications automatically in the Polyspace Metrics repository...**.
   - Download settings — In Polyspace Metrics, when you click an item to view, Polyspace downloads your results and opens them in the Polyspace environment. Select where to download your Polyspace Metrics results, either:

     - To the project folder, or, if a project does not exist, a default folder.
     - Ask every time where to download results.

To view Polyspace Metrics, in the address bar of your web browser, enter:

`protocol://ServerName:WSPN`

- *`protocol`* is `http` or `https`.
- *`ServerName`* is the name or IP address of your Polyspace Metrics server.
- *`WSPN`* is the web server port number, the default is 8080 or 8443.

Configure Web Server for HTTPS

By default, the data transfer between the Polyspace desktop products and the Polyspace Metrics web interface is not encrypted. You can enable HTTPS for the web protocol, which encrypts the data transfer. To set up HTTPS, you must change the server configuration and set up a keystore for the HTTPS certificate.

Before you start the following procedure, you must complete “Configure and Start Polyspace Metrics Server” on page 2-18 and “Configure Client Side” on page 2-19.

To configure HTTPS access to Polyspace Metrics:

1. Open the Metrics Server Settings dialog box as stated in “Configure and Start Polyspace Metrics Server” on page 2-18.
2. Click **Stop Server**. The software stops the Polyspace Metrics services. Now, you can make the changes required for HTTPS.
3. Open the file `*metricsRootFolder*\tomcat\conf\server.xml` in a text editor. Here, *`metricsRootFolder`* is the name that you specified for **Folder where analysis data will be stored**. Look for the following text:

   ```
   <!-
     <Connector port="8443" SSLEnabled="true" scheme="https"
     secure="true" clientAuth="false" sslProtocol="TLS"
     keystoreFile="<datadir>/.keystore" keystorePass="polyspace"/>
   ->
   ```

   If the text is not in your `server.xml` file:

   a. Delete the entire `..\conf\` folder.
   b. In the Metrics Server Settings dialog box, restart services by clicking **Start Server**.
   c. Click **Stop Server** to stop the services again so that you can finish setting up the server for HTTPS.

   The `conf` folder is regenerated, including the `server.xml` file. The file now contains the text required to configure the HTTPS web server.
4. Follow the commented-out instructions in `server.xml` to create a keystore for the HTTPS certificate.
5. In the Metrics Server Settings dialog box, to restart the Polyspace Metrics service with the changes, click **Start Server**.

To view Polyspace Metrics, in the address bar of your web browser, enter:

`https``://`*`ServerName`*`:`*`WSPN`*

- *`ServerName`* is the name or IP address of the Polyspace Metrics server.
- *`WSPN`* is the web server port number.

Change Web Server Port Number for Metrics Server

If you change or specify a non-default value for the web server port number of your Polyspace Code Prover client, you must manually configure the same value for your Polyspace Metrics server.

1. Select **Metrics** > **Metrics Server Settings**.
2. In the Metrics Server Settings dialog box, select **Stop Server** to stop the Polyspace Metrics server daemon.
3. In *`metricsRootFolder`*`\tomcat\conf\server.xml`, edit the `port` attribute of the `Connector` element for your web server protocol. Here, *`metricsRootFolder`* is the name that you specified for **Folder where analysis data will be stored** when setting up Polyspace Metrics.

   - For HTTP:

     ```
     <Connector port="8080"/>
     ```
   - For HTTPS:

     ```
     <Connector port="8443" SSLEnabled="true" scheme="https"
     secure="true" clientAuth="false" sslProtocol="TLS"
     keystoreFile="<datadir>/.keystore" keystorePass="polyspace"/>
     ```
4. In the same file, edit the `port` attribute of the `Server` element for your web server protocol.

   ```
   <Server port="8005" shutdown="SHUTDOWN">
   ```
5. In the Metrics Server Settings dialog box, select **Start Server** to restart the server with the new port numbers.
6. On the Polyspace toolbar, select **Tools** > **Preferences**.
7. In the **Server Configuration** tab, change the **Web server port number** to match your new value for the `port` attribute in the `Connector` element.

See Also

Related Examples

- “Generate Code Quality Metrics”

Get Started with Polyspace Code Prover

Compiler Requirements

Polyspace fully supports the most common compilers used to develop embedded applications. If you compile your code with one of these compilers, you can run analysis simply by specifying your compiler and target processor. See the full list of compilers on the reference page for option `Compiler (-compiler)`.

If you do not compile your code using a supported compiler, you can specify a generic compiler. If you face compilation errors from compiler-specific language extensions, you can explicitly define these extensions to work around the errors. Use the options `Preprocessor definitions (-D)` and `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Run Polyspace Code Prover on C/C++ Code

Polyspace Code Prover is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. A Code Prover analysis produces results without requiring program execution, code instrumentation, or test cases. Code Prover uses semantic analysis and abstract interpretation based on formal methods to determine control flow and data flow in the code. You can use Code Prover on handwritten code, generated code, or a combination of the two. In the analysis results, each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

You can run Code Prover on C/C++ code from the Polyspace user interface, in a supported development environment (IDE) such as Eclipse or using scripts. See:

- "Run Polyspace in User Interface" on page 3-3
- "Run Polyspace on Windows or Linux Command Line" on page 3-7
- "Run Polyspace in Eclipse" on page 3-8
- "Run Polyspace in MATLAB" on page 3-8

To follow the steps in this tutorial, copy the files `example.c` and `include.h` from *`polyspaceroot`*`\polyspace\examples\cxx\Code_Prover_Example\sources` to another folder. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Run Polyspace in User Interface

Open Polyspace User Interface

Double-click the `polyspace` executable in *`polyspaceroot`*`\polyspace\bin`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

If you set up a shortcut to Polyspace on your desktop or the **Start** menu in Windows, double-click the shortcut.

Add Source Files

To run a verification, you have to create a new Polyspace project. A Polyspace project points to source and include folders on your file system.

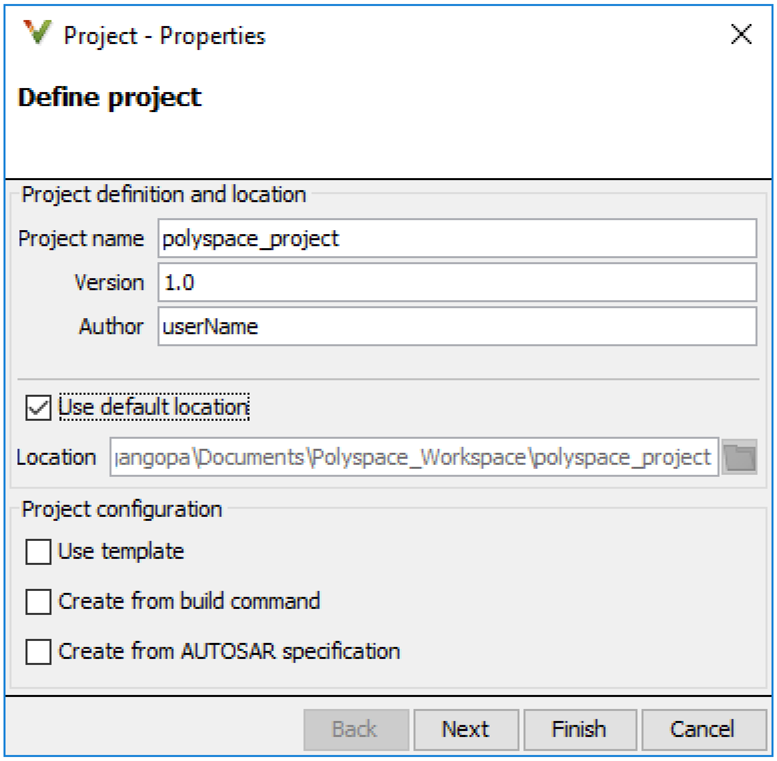
On the left of the **Start Page** pane, click **Start a new project**. Alternatively, select **File > New Project**.

After you provide a project name, on the next screens:

- Add your source folder.

  In this tutorial, add the path to the folder in which you saved the file `example.c`. Click **Next**.
- Add your include folder.

  In this tutorial, add the path to the folder in which you saved the file `include.h`. This folder can be the same as the previous folder. Click **Finish**.



Project - Properties

Define project

Project definition and location

Project name: polyspace_project

Version: 1.0

Author: userName

☑ Use default location

Location: iangopa\Documents\Polyspace_Workspace\polyspace_project

Project configuration

☐ Use template

☐ Create from build command

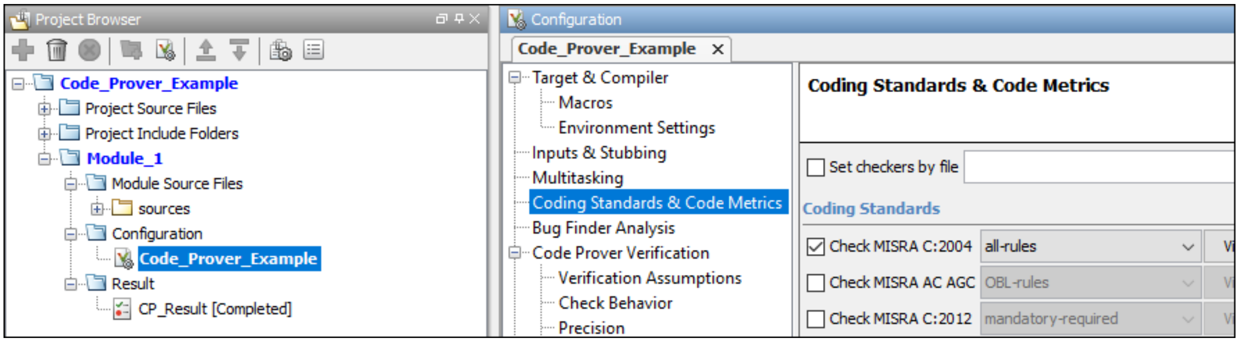
☐ Create from AUTOSAR specification

After you finish adding your source and include folders, you see a new project on the **Project Browser** pane. Your source folders are copied to the first module in the project. You can right-click a project to add more folders later. If you add folders later, you must explicitly copy them to a module.

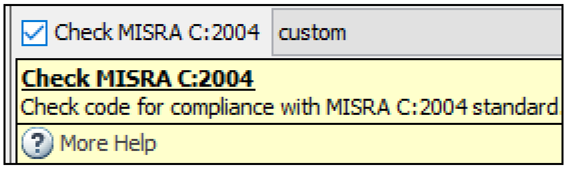
Configure and Run Polyspace

You can change the default options associated with a Polyspace analysis.

Click the **Configuration** node in your project module. On the **Configuration** pane, change options as needed. For instance, on the **Coding Rules & Code Metrics** node, select **Check MISRA C:2004**.



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


Check MISRA C:2004 custom
Check MISRA C:2004
Check code for compliance with MISRA C:2004 standard
More Help

To start verification, click **Run Code Prover** in the top toolbar. If the button indicates Bug Finder, click the arrow beside the button to switch to Code Prover.

Follow the progress of verification on the **Output Summary** window. After the verification, the results open automatically.

Additional Information

See:

- “Add Source Files for Analysis in Polyspace User Interface”

- "Run Polyspace Analysis on Desktop"

Run Polyspace on Windows or Linux Command Line

You can run Code Prover from the Windows or Linux command line with batch (`.bat`) files or shell (`.sh`) scripts.

Use the `polyspace-code-prover` command to run a verification.

To save typing the full path to the command, add the path *`polyspaceroot`*`\polyspace\bin` to the `Path` environment variable on your operating system. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Navigate to the folder where you saved the files (using `cd`). Enter the following:

```
polyspace-code-prover -sources example.c -I . -results-dir . -main-generator
```

Here, `.` indicates the current folder. The options used are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- `-results-dir`: Specify path where Polyspace Code Prover results will be saved.
- `Verify module or library (-main-generator)`: Specify that a `main` function must be generated if not found in the source files

After verification, the results are saved in the file `ps_results.pscp`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
polyspace ps_results.pscp
```

Instead of specifying comma-separated sources directly on the command line, you can list the sources in a text file (one file per line). Use the option `-sources-list-file` to specify this text file.

Additional Information

See:

- “Run Polyspace Analysis from Command Line”
- `polyspace-code-prover`

Run Polyspace in Eclipse

If you develop code in Eclipse or an Eclipse-based IDE, you can run Code Prover directly from your IDE.

After installing the Eclipse plugin on page 2-2, you can run Polyspace directly on the files in your Eclipse projects.

In the **Project Explorer** pane in Eclipse, select your project. To use Code Prover for the analysis, select **Polyspace** > **Code Prover**. To start the analysis, select **Polyspace** > **Run** (`Ctrl + R`).

After analysis, the results open automatically in Eclipse.

Additional Information

See “Run Polyspace Analysis in Eclipse”.

Run Polyspace in MATLAB

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

To run an analysis, use a `polyspace.Project` object. The object has two properties:

- `Configuration`: Specify the analysis options such as sources, includes, compiler and results folder using this property.
- `Results`: After analysis, read the analysis results to a MATLAB table using this property.

To run the analysis, use the `run` method of this object.

To run Polyspace on the example file `example.c` in *`polyspaceroot`*`\polyspace\examples\cxx\Code_Prover_Examples\sources`, enter the following at the MATLAB command prompt.

```
proj = polyspace.Project

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


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

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

After verification, the results are saved in the file `ps_results.pscp`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
resultsFile = fullfile(proj.Configuration.ResultsDir,'ps_results.pscp');
polyspaceCodeProver(resultsFile)
```

Additional Information

See:

- "Run Polyspace Analysis by Using MATLAB Scripts"
- `polyspace.Project`
- polyspace.Project.Configuration Properties

See Also

Related Examples

- “Review Polyspace Code Prover Analysis Results” on page 3-11
- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Review Polyspace Code Prover Analysis Results

Polyspace Code Prover checks C/C++ code exhaustively and proves the absence of certain types of run-time errors (static analysis or verification). Whatever means you use for running the analysis, afterwards, you open the results in the Polyspace user interface (or if you ran the analysis in Eclipse, the results open in Eclipse).

To follow the steps in this tutorial, run Polyspace using the steps in “Run Polyspace Code Prover on C/C++ Code” on page 3-3. Alternatively, in the Polyspace user interface, open example results using **Help** > **Examples** > **Code_Prover_Example.psprj**. If you have loaded the example results earlier and made some changes, to load a fresh copy, select **Help** > **Examples** > **Restore Default Examples**.

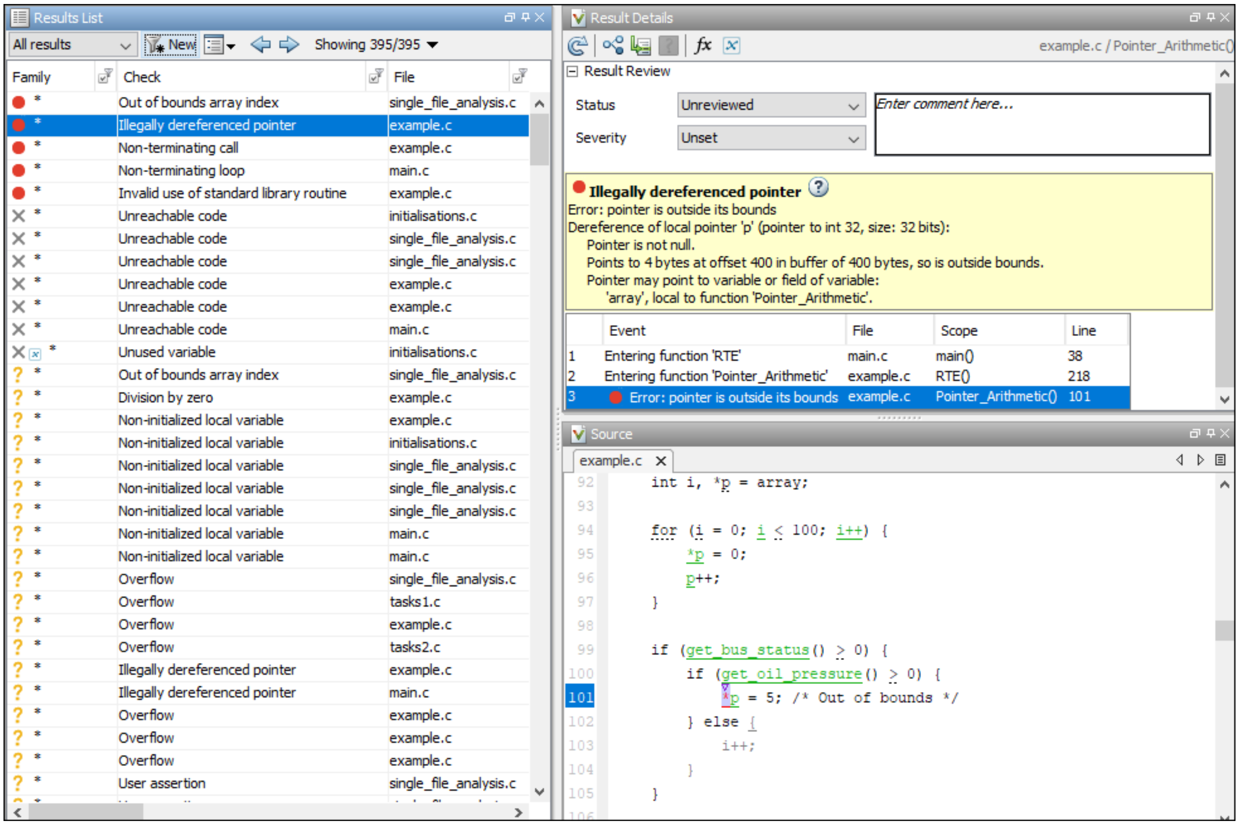
Interpret Results

Review each Polyspace result. Find the root cause of the issue.

Start from the list of results on the **Results List** pane.

- If the **Results List** pane covers the entire window, select **Window > Reset Layout > Results Review**.
- If you do not see a flat list of results, but instead see them grouped, from the list, select **None**.

Click the **Family** column header to sort the results based on how critical they are. Select the red **Illegally dereferenced pointer** check in the file `example.c`. A red check indicates that the error happens on all execution paths considered in the analysis.



Family	Check	File
● *	Out of bounds array index	single_file_analysis.c
● *	Illegally dereferenced pointer	example.c
● *	Non-terminating call	example.c
● *	Non-terminating loop	main.c
● *	Invalid use of standard library routine	example.c
✕ *	Unreachable code	initialisations.c
✕ *	Unreachable code	single_file_analysis.c
✕ *	Unreachable code	single_file_analysis.c
✕ *	Unreachable code	example.c
✕ *	Unreachable code	example.c
✕ *	Unreachable code	main.c
✕ *	Unused variable	initialisations.c
? *	Out of bounds array index	single_file_analysis.c
? *	Division by zero	example.c
? *	Non-initialized local variable	example.c
? *	Non-initialized local variable	initialisations.c
? *	Non-initialized local variable	single_file_analysis.c
? *	Non-initialized local variable	single_file_analysis.c
? *	Non-initialized local variable	single_file_analysis.c
? *	Non-initialized local variable	main.c
? *	Non-initialized local variable	main.c
? *	Overflow	single_file_analysis.c
? *	Overflow	tasks1.c
? *	Overflow	example.c
? *	Overflow	tasks2.c
? *	Illegally dereferenced pointer	example.c
? *	Illegally dereferenced pointer	main.c
? *	Overflow	example.c
? *	Overflow	example.c
? *	Overflow	example.c
? *	User assertion	single_file_analysis.c

Result Review

Status: Unreviewed
Severity: Unset
Enter comment here...

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'.

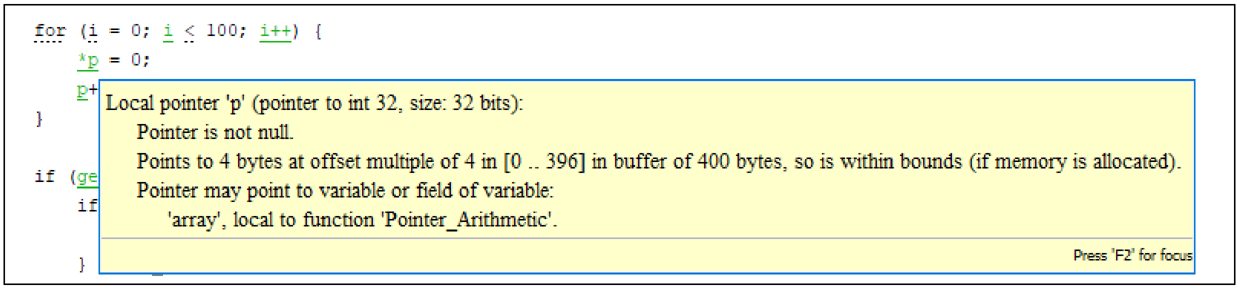
	Event	File	Scope	Line
1	Entering function 'RTE'	main.c	main()	38
2	Entering function 'Pointer_Arithmetic'	example.c	RTE()	218
3	● Error: pointer is outside its bounds	example.c	Pointer_Arithmetic()	101

```
    int i, *p = array;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5; /* Out of bounds */
        } else {
            i++;
        }
    }
```

See the source code on the **Source** pane and further information about the result on the **Result Details** pane.

For the **Illegally dereferenced pointer** result, the message on the **Result Details** pane indicates that the pointer p has an allowed buffer of 400 bytes. It points to a location that begins at 400 bytes from the beginning of the buffer and points to a data type of 4 bytes.

To investigate further and find the root cause of the issue, right-click the variable p on the **Source** pane and select **Search For All References**. Click each search result to navigate to the corresponding location on the source code. At each location, place your cursor on the variable p to see a tooltip that describes the variable value at that point in the code.



```
for (i = 0; i < 100; i++) {
    *p = 0;
    p+
}

if (ge
    if

    }
```

Local pointer 'p' (pointer to int 32, size: 32 bits):
    Pointer is not null.
    Points to 4 bytes at offset multiple of 4 in [0 .. 396] in buffer of 400 bytes, so is within bounds (if memory is allocated).
    Pointer may point to variable or field of variable:
        'array', local to function 'Pointer_Arithmetic'.

You see that the pointer variable `p` is initialized to a 100-element `int` array. The pointer traverses the array in a `for` loop with 100 iterations and points to the end of the array. On the line with the red **Illegally dereferenced pointer** check, this pointer is dereferenced, resulting in dereference of a memory location outside the array.

Additional Information

See:

- "Interpret Polyspace Code Prover Results"
- "Code Prover Result and Source Code Colors"
- "Polyspace Code Prover Results"

Address Results Through Bug Fix or Comments

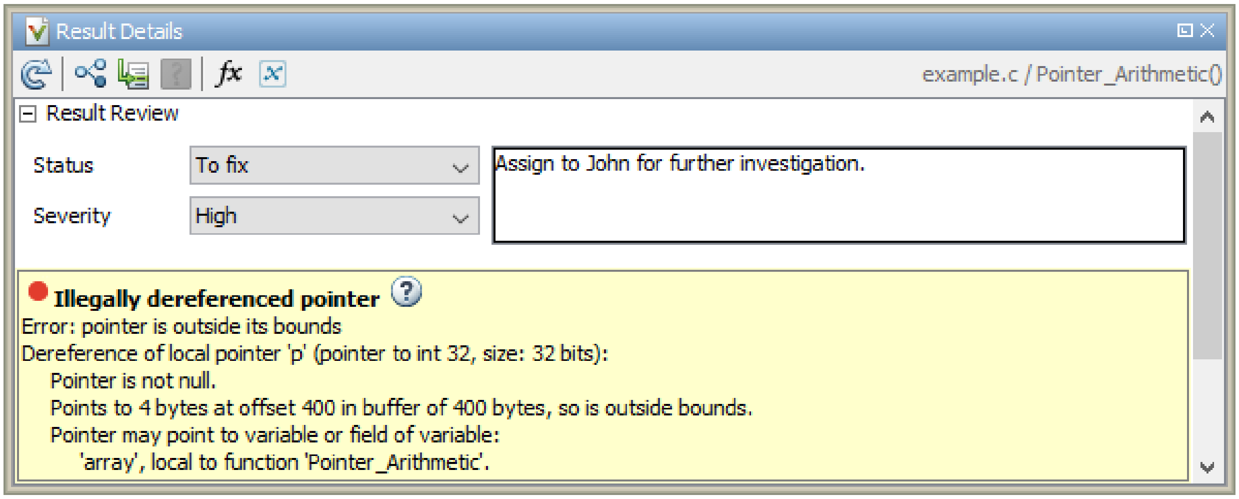
Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add comments to your Polyspace results to fix the code later or to justify the result. You can use the comments to keep track of your review progress.

Right-click the variable `p` on the **Source** pane. Select **Open Editor**. The code opens in a text editor. Fix the issue. For instance, you can make the pointer point to the beginning of the array after the `for` loop. Changes to the code are highlighted below.

```
...
int i, *p = array;

for (i = 0; i < 100; i++) {
    *p = 0;
     p++;
}

p = array;

if (get_bus_status() > 0)
...
```

If you rerun the analysis, you do not see the red **Illegally dereferenced pointer** check.

Alternatively, if you do not want to fix the defect immediately, assign a status **To investigate** to the result. Optionally, add comments with further explanation.



Status: To fix

Severity: High

Assign to John for further investigation.

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'.

If you assign a status **No action planned**, the result does not appear in subsequent runs on the same project.

Additional Information

See:

- “Address Polyspace Results Through Bug Fixes or Comments”
- “Annotate Code and Hide Known or Acceptable Results”

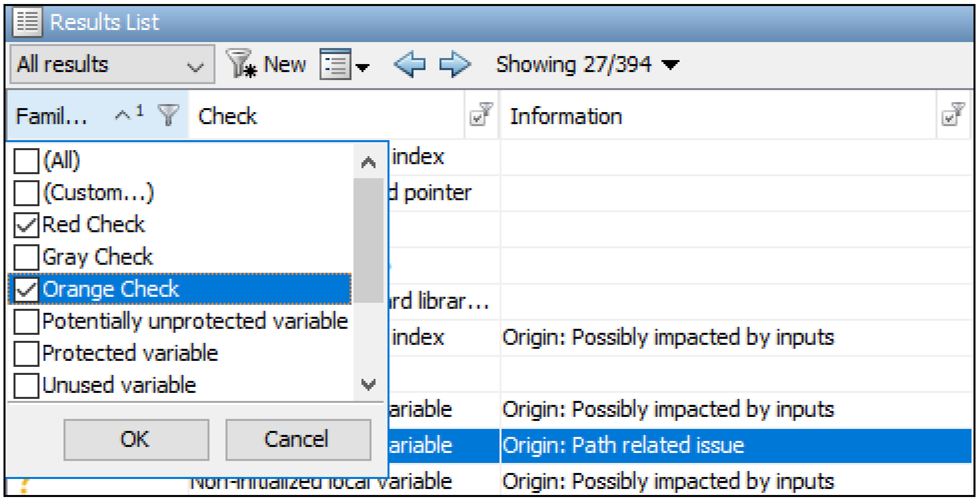
Manage Results

When you open the results of a Code Prover analysis, you see a list of run-time checks, coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.

For instance, you can:

- Review only red and critical orange checks.

  Click the **Family** column header to sort checks by color. Alternatively, you can filter out results other than red and orange checks. To begin filtering, click the icon on the column header.



You can review only the path-related orange checks because they are likely to be more critical. To filter out other checks, use the filters on the **Information** column. Clear the **All** filter and then select the filter **Origin: Path related issue**.

- Review only the new results since the last analysis.

  On the **Results List** pane toolbar, click the **New** button.
- Review results in certain files or functions.

  On the **Results List** pane toolbar, from the list, select **File**.

Additional Information

See:

- “Filter and Group Results”
- “Prioritize Check Review”

Send Code Prover Analysis from Desktop to Locally Hosted Server

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated servers. This topic shows a simple server-client configuration for offloading the Polyspace analysis. In this configuration, the same computer acts as a client that submits a Polyspace analysis and a server that runs the analysis.

You can extend this tutorial to more complex configurations. For full setup and workflow instructions, see related links below.

Client-Server Workflow for Running Bug Finder Analysis

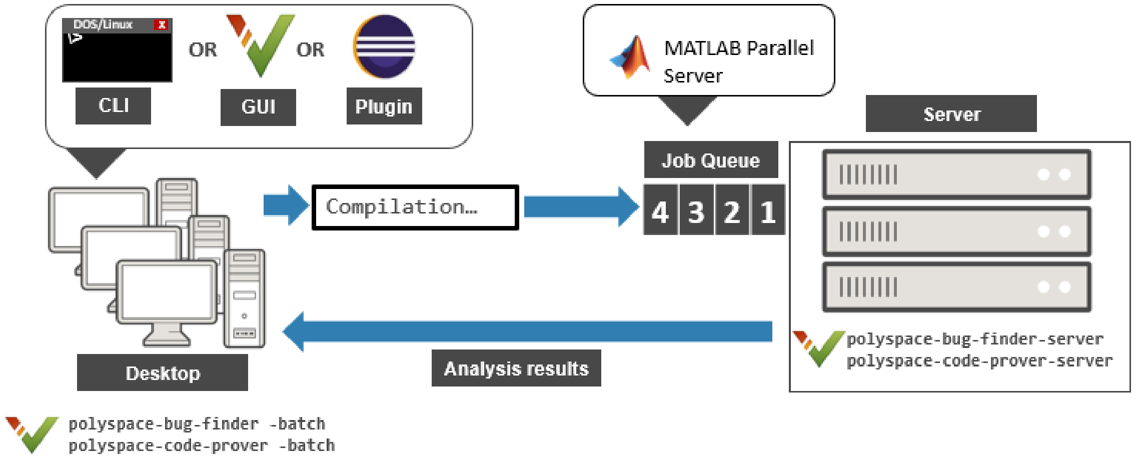
After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers. This tutorial uses the same computer for the entire workflow.

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

   You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.
2. **Head node**: The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

   You require the product MATLAB Parallel Server on the computer that acts as the head node.
3. **Worker nodes**: When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

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


DOS/Linux
CLI
OR
GUI
OR
Plugin
MATLAB Parallel Server
Server
Job Queue
4 3 2 1
Compilation...
Desktop
Analysis results
polyspace-bug-finder-server
polyspace-code-prover-server
polyspace-bug-finder -batch
polyspace-code-prover -batch

See also “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server” on page 2-5.

Prerequisites

This tutorial uses the same computer as client and server. You must install the following on the computer:

- Client-side product: Polyspace Bug Finder
- Server-side products: MATLAB Parallel Server, Polyspace Bug Finder Server and Polyspace Code Prover Server.

For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server” on page 2-5.

You must know the host name of your computer. For instance, in Windows, open a command-line terminal and enter:

```
hostname
```

Configure and Start Server

Stop Previous Services

If you started the services of MATLAB Parallel Server previously, make sure that you have stopped all services. In particular, you might have to:

- Check your temporary folder, for instance, `C:\Windows\Temp` in Windows, and remove the `MDCE` folder if it exists.
- Stop all services explicitly.

  Open a command-line terminal. Navigate to *`matlabroot`*`\toolbox\distcomp\bin` (using `cd`) and enter the following:

  ```
  mjs uninstall -clean
  ```

  Here, *`matlabroot`* is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2019a`.

If this is the first time you are starting the services, you do not have to do these steps.

Configure mjs Service Settings

Before starting services, you have to configure the `mjs` service settings. Navigate to *`matlabroot`*`\toolbox\distcomp\bin`, where *`matlabroot`* is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2019a`. Modify these two files. To edit and save these files, you have to open your editor in administrator mode.

- `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux)

Read the instructions in the file and uncomment the lines as needed. At a minimum, you might have to uncomment these lines:

- Hostname:

  ```
  REM set HOSTNAME=myHostName
  ```

  in Windows or

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

  in Linux. Remove the `REM` or `#` and explicitly specify your computer host name.
- Security level:

  ```
  REM set SECURITY_LEVEL=
  ```

  in Windows or

  ```
  #SECURITY_LEVEL=""
  ```

  in Linux. Remove the `REM` or `#` and explicitly specify a security level.

Otherwise, you might see an error later when starting the job scheduler.

- `mjs_polyspace.conf`

  Modify and uncomment the line that refers to a Polyspace server product root. The line should refer to the release number and root folder of your Polyspace server product installation. For instance, if the R2019a release of Polyspace Bug Finder Server is installed in the root folder `C:\Program Files\Polyspace Server\R2019a`, modify the line to:

  ```
  POLYSPACE_SERVER_R2019A_ROOT=C:\Program Files\Polyspace Server\R2019a
  ```

  Otherwise, the MATLAB Parallel Server installation cannot locate the Polyspace Bug Finder Server installation to run the analysis.

Start Services

Start the `mjs` services and assign the current computer as both the head node and a worker node.

Navigate to *`matlabroot`*`\toolbox\distcomp\bin`, where *`matlabroot`* is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2019a`. Run these commands (directly at the command line or using scripts):

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

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

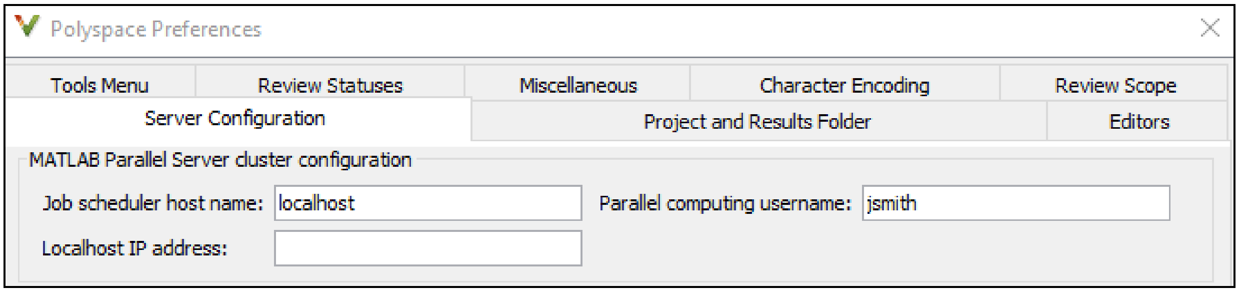
Instead of the command line, you can also start the services from the Admin Center interface. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server” on page 2-5.

For more information on the commands, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

Configure Client

Open the user interface of the desktop product, Polyspace Bug Finder (or Polyspace Code Prover). Navigate to *`polyspaceroot`*`\polyspace\bin`, where *`polyspaceroot`* is the Polyspace desktop product installation folder, for instance, `C:\Program Files\Polyspace\R2019a` and double-click the `polyspace` executable.

Select **Tools > Preferences**. On the **Server configuration** tab, enter the host name of your computer for **Job scheduler host name**.



MATLAB Parallel Server cluster configuration

Job scheduler host name: localhost

Parallel computing username: jsmith

Localhost IP address:

You are now set up for the server-client workflow.

Send Analysis from Client to Server

Run Code Prover on the file `example.c` provided with your installation.

Before running these steps, to avoid entering full paths to the Polyspace executables, add the path `*polyspaceroot*\polyspace\bin` to the `PATH` environment variable on your operating system. Here *`polyspaceroot`* is the Polyspace desktop product installation folder, for instance, `C:\Program Files\Polyspace\R2019a`. To check if the path is already added, open a command line terminal and enter:

```
polyspace-bug-finder -h
```

If the path to the command is already added, you see the full list of options.

1. Copy the file `example.c` and all header files from *`polyspaceroot`*`\polyspace\examples\cxx\Code_Prover_Example\sources` to a folder with write permissions.
2. Open a command terminal. Navigate to the folder where you saved `example.c` and enter the following:

   ```
   polyspace-code-prover -sources example.c -I . -main-generator
      -results-dir . -batch -scheduler hostname
   ```

   Here, *`hostname`* is the host name of your computer. To run a Bug Finder analysis, use `polyspace-bug-finder` instead of `polyspace-code-prover`. Note that you can run the `polyspace-code-prover` command with a Polyspace Bug Finder license only, provided you use the `-batch` option.

After compilation, the analysis is submitted to a server and returns a job ID.

3 See the status of the current job.

```
polyspace-jobs-manager listjobs -scheduler hostname
```

You can locate the current job using the job ID.

4 Once the job is completed, you can explicitly download the results.

```
polyspace-jobs-manager download -job jobID -results-folder .
   -scheduler hostname
```

Here, *`jobID`* is the job ID from the submission.

The results folder contains the downloaded results file (with extension `.pscp`). Open the results in the user interface of the desktop product, Polyspace Bug Finder.

See Also

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server” on page 2-5
- “Send Polyspace Analysis from Desktop to Remote Servers”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”

Polyspace Bug Finder and Polyspace Code Prover

Choose Between Polyspace Bug Finder and Polyspace Code Prover

Polyspace Bug Finder and Polyspace Code Prover detect run-time errors through static analysis. Though the products have a similar user interface and the mathematics underlying the analysis can sometimes be the same, the goals of the two products are different.

Bug Finder quickly analyzes your code and detects many types of defects. Code Prover checks *every* operation in your code for a set of possible run-time errors and tries to prove the absence of the error for all execution paths[2]. For instance, for *every* division in your code, a Code Prover analysis tries to prove that the denominator cannot be zero. Bug Finder does not perform such exhaustive verification. For instance, Bug Finder also checks for a division by zero error, but it might not find all operations that can cause the error.

The two products involve differences in setup, analysis and results review, because of this difference in objectives. In the following sections, we highlight the primary differences between a Bug Finder and a Code Prover analysis (also known as verification). Depending on your requirements, you can incorporate one or both kinds of analyses at appropriate points in your software development life cycle.

How Bug Finder and Code Prover Complement Each Other

2. For each operation in your code, Code Prover considers all execution paths leading to the operation that do not have a previous error. If an execution path contains an error prior to the operation, Code Prover does not consider it. See “Code Prover Analysis Following Red and Orange Checks”.

- “Zero False Negatives with Code Prover” on page 4-8

Overview

Use both Bug Finder and Code Prover regularly in your development process. The products provide a unique set of capabilities and complement each other. For possible ways to use the products together, see “Workflow Using Both Bug Finder and Code Prover” on page 4-8.

This table provides an overview of how the products complement each other. For details, see the sections below.

Feature	Bug Finder	Code Prover
Number of checkers	251	28 (Critical subset)
Depth of analysis	Fast. For instance: • Faster analysis. • Easier set up and review. • Fewer runs for clean code. • Results in real time.	Exhaustive. For instance: • All operations of a type checked for certain critical errors. • More rigorous data and control flow analysis.
Reporting criteria	Probable defects	Proven findings
Bug finding criteria	Few false positives	Zero false negatives

Faster Analysis with Bug Finder

How much faster the Bug Finder analysis is depends on the size of the application. The Bug Finder analysis time increases linearly with the size of the application. The Code Prover verification time increases at a rate faster than linear.

One possible workflow is to run Code Prover to analyze modules or libraries for robustness against certain errors and run Bug Finder at integration stage. Bug Finder analysis on large code bases can be completed in a much shorter time, and also find integration defects such as **Declaration mismatch** and **Data race**.

More Exhaustive Verification with Code Prover

Code Prover tries to prove the absence of:

- **Division by Zero** error on *every* division or modulus operation
- **Out of Bounds Array Index** error on *every* array access
- **Non-initialized Variable** error on *every* variable read
- **Overflow** error on *every* operation that can overflow

and so on.

For each operation:

- If Code Prover can prove the absence of the error for all execution paths, it highlights the operation in green.
- If Code Prover can prove the presence of a definite error for all execution paths, it highlights the operation in red.
- If Code Prover cannot prove the absence of an error or presence of a definite error, it highlights the operation in orange. This small percentage of orange checks indicate operations that you must review carefully, through visual inspection or testing. The orange checks often indicate hidden vulnerabilities. For instance, the operation might be safe in the current context but fail when reused in another context.

  You can use information provided in the Polyspace user interface to diagnose the checks. See “More Rigorous Data and Control Flow Analysis with Code Prover” on page 4-6. You can also provide contextual information to reduce unproven code even further, for instance, constrain input ranges externally.

Bug Finder does not aim for exhaustive analysis. It tries to detect as many bugs as possible and reduce false positives. For critical software components, running a bug finding tool is not sufficient because despite fixing all defects found in the analysis, you can still have errors during code execution (false negatives). After running Code Prover on your code and addressing the issues found, you can expect the quality of your code to be much higher. See “Zero False Negatives with Code Prover” on page 4-8.

More Specific Defect Types with Bug Finder

Code Prover checks for types of run-time errors where it is possible to mathematically prove the absence of the error. In addition to detecting errors whose absence can be mathematically proven, Bug Finder also detects other defects.

For instance, the statement `if(a=b)` is semantically correct according to the C language standard, but often indicates an unintended assignment. Bug Finder detects such unintended operations. Although Code Prover does not detect such unintended operations, it can detect if an unintended operation causes other run-time errors.

Examples of defects detected by Bug Finder but not by Code Prover include good practice defects (Polyspace Bug Finder), resource management defects (Polyspace Bug Finder), some programming defects (Polyspace Bug Finder), security defects (Polyspace Bug Finder), and defects in C++ object oriented design (Polyspace Bug Finder).

For more information, see:

- “Defects” (Polyspace Bug Finder): List of defects that Bug Finder can detect.
- “Run-Time Checks”: List of run-time errors that Code Prover can detect.

Easier Setup Process with Bug Finder

Even if your code builds successfully in your compilation toolchain, it can fail in the compilation phase of a Code Prover verification. The strict compilation in Code Prover is related to its ability to prove the absence of certain run-time errors.

- Code Prover strictly follows the ANSI® C99 Standard, unless you explicitly use analysis options that emulate your compiler.

  To allow deviations from the ANSI C99 Standard, you must use the options. If you create a Polyspace project from your build system, the options are automatically set.
- Code Prover does not allow linking errors that common compilers can permit.

  Though your compiler permits linking errors such as mismatch in function signature between compilation units, to avoid unexpected behavior at run time, you must fix the errors.

For more information, see “Troubleshoot Compilation and Linking Errors”.

Bug Finder is less strict about certain compilation errors. Linking errors, such as mismatch in function signature between different compilation units, can stop a Code Prover verification but not a Bug Finder analysis. Therefore, you can run a Bug Finder analysis with less setup effort. In Bug Finder, linking errors are often reported as a defect after the analysis is complete.

Fewer Runs for Clean Code with Bug Finder

To guarantee absence of certain run-time errors, Code Prover follows strict rules once it detects a run-time error in an operation. Once a run-time error occurs, the state of your program is ill-defined and Code Prover cannot prove the absence of errors in subsequent code. Therefore:

- If Code Prover proves a definite error and displays a red check, it does not verify the remaining code in the same block.

  Exceptions include checks such as **Overflow**, where the analysis continues with the result of overflow either truncated or wrapped around.
- If Code Prover suspects the presence of an error and displays an orange check, it eliminates the path containing the error from consideration. For instance, if Code Prover detects a **Division by Zero** error in the operation `1/x`, in the subsequent operation on `x` in that block, `x` cannot be zero.
- If Code Prover detects that a code block is unreachable and displays a gray check, it does not detect errors in that block.

For more information, see “Code Prover Analysis Following Red and Orange Checks”.

Therefore, once you fix red and gray checks and rerun verification, you can find more issues. You need to run verification several times and fix issues each time for completely clean code. The situation is similar to dynamic testing. In dynamic testing, once you fix a failure at a certain point in the code, you can uncover a new failure in subsequent code.

Bug Finder does not stop the entire analysis in a block after it finds a defect in that block. Even with Bug Finder, you might have to run analysis several times to obtain completely clean code. However, the number of runs required is fewer than Code Prover.

Results in Real Time with Bug Finder

Bug Finder shows some analysis results while the analysis is still running. You do not have to wait until the end of the analysis to review the results.

Code Prover shows results only after the end of the verification. Once Bug Finder finds a defect, it can display the defect. Code Prover has to prove the absence of errors on all execution paths. Therefore, it cannot display results during analysis.

More Rigorous Data and Control Flow Analysis with Code Prover

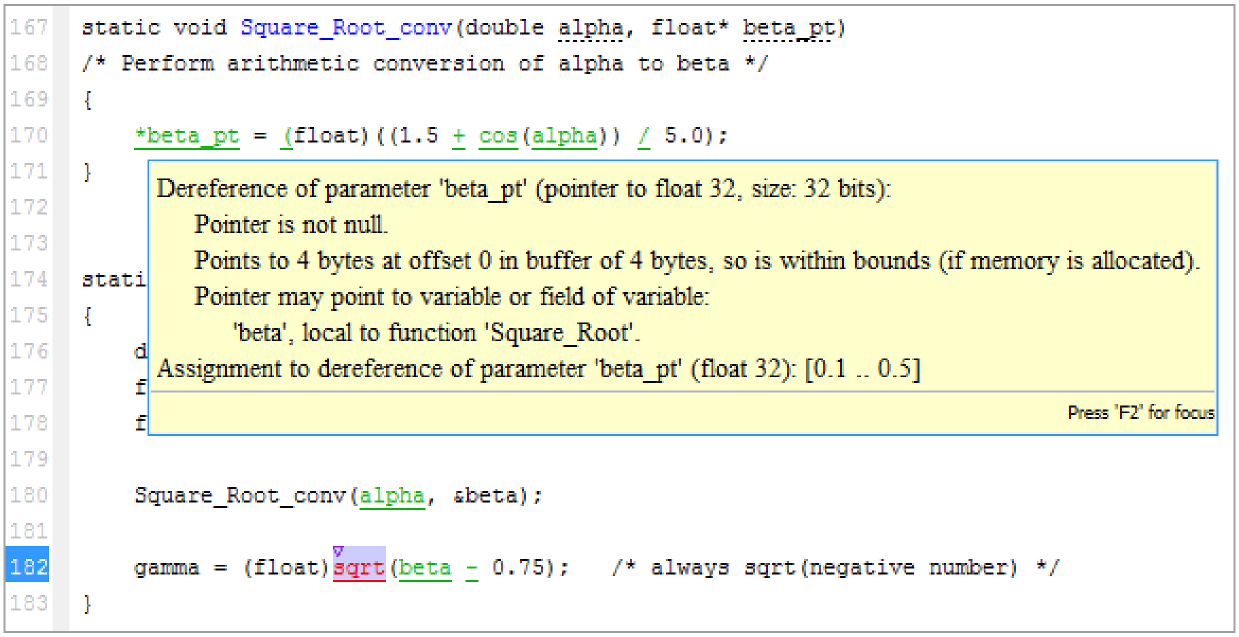
For each operation in your code, Code Prover provides:

- Tooltips showing the range of values of each variable in the operation.

  For a pointer, the tooltips show the variable that the pointer points to, along with the variable values.
- Graphical representation of the function call sequence that leads to the operation.

By using this range information and call graph, you can easily navigate the function call hierarchy and understand how a variable acquires values that lead to an error. For instance, for an **Out of Bounds Array Index** error, you can find where the index variable is first assigned values that lead to the error.

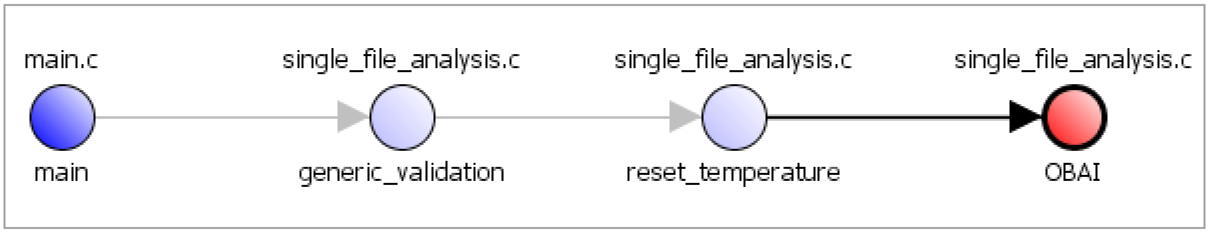
When reviewing a result in Bug Finder, you also have supporting information to understand the root cause of a defect. For instance, you have a traceback from where Bug Finder found a defect to its root cause. However, in Code Prover, you have more complete information, because the information helps you understand all execution paths in your code.



```
static void Square_Root_conv(double alpha, float* beta_pt)
/* Perform arithmetic conversion of alpha to beta */
{
    *beta_pt = (float)((1.5 + cos(alpha)) / 5.0);
}


stati
{
    d
    f
    f

    Square_Root_conv(alpha, &beta);

    gamma = (float)sqrt(beta - 0.75);   /* always sqrt(negative number) */
}
```

```
Dereference of parameter 'beta_pt' (pointer to float 32, size: 32 bits):
    Pointer is not null.
    Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).
    Pointer may point to variable or field of variable:
        'beta', local to function 'Square_Root'.
Assignment to dereference of parameter 'beta_pt' (float 32): [0.1 .. 0.5]
                                                                                  Press 'F2' for focus
```

Data Flow Analysis in Code Prover



Control Flow Analysis in Code Prover

Few False Positives with Bug Finder

Bug Finder aims for few false positives, that is, results that you are not likely to fix. By default, you are shown only the defects that are likely to be most meaningful for you.

Bug Finder also assigns an attribute called impact to the defect types based on the criticality of the defect and the rate of false positives. You can choose to analyze your code only for high-impact defects. You can also enable or disable a defect that you do not want to review[3].

Zero False Negatives with Code Prover

Code Prover aims for an exhaustive analysis. The software checks every operation that can trigger specific types of error. If a code operation is green, it means that the operation cannot cause those run-time errors that the software checked for[4]. In this way, the software aims for zero false negatives.

If the software cannot prove the absence of an error, it highlights the suspect operation in red or orange and requires you to review the operation.

Workflow Using Both Bug Finder and Code Prover

If you have both Bug Finder and Code Prover, based on the above differences, you can deploy the two products appropriately in your software development workflow. For instance:

3. You can also disable certain Code Prover defects related to non-initialization.
4. The Code Prover result holds only if you execute your code under the same conditions that you supplied to Code Prover through the analysis options.

- All developers in your organization can run Bug Finder on newly developed code. For maintaining standards across your organization, you can deploy a common configuration that looks only for specific defect types.

  Code Prover can be deployed as part of your unit testing suite.
- You can run Code Prover only on critical components of your project, while running Bug Finder on the entire project.
- You can run Code Prover on modules of code at the unit testing level, and run Bug Finder when integrating the modules.

  You can run Code Prover before unit testing. Code Prover exhaustively checks your code and tries to prove the presence or absence of errors. Instead of writing unit tests for your entire code, you can then write tests only for unproven code. Using Code Prover before unit testing reduces your testing efforts drastically.

Depending on the nature of your software development workflow and available resources, there are many other ways you can incorporate the two kinds of analysis. You can run both products on your desktop during development or as part of automated testing on a remote server. Note that it is easier to interpret and fix bugs closer to development. You will benefit from using both products if you deploy them early and often in your development process.

There are two important considerations if you are running both Bug Finder and Code Prover on the same code.

- Both products can detect violations of coding rules such as MISRA C rules and JSF C++ rules.

  However, if you want to detect MISRA C:2012 coding rule violations alone, use Bug Finder. Bug Finder supports all the MISRA C:2012 coding rules. Code Prover does not support a few rules.
- If a result is found in both a Bug Finder and Code Prover analysis, you can comment on the Bug Finder result and import the comment to Code Prover.

  For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add comments to coding rule violations in Bug Finder and import the comments to the same violations in Code Prover. To import comments, open your result set and select **Tools > Import Comments**.
- You can use the same project for both Bug Finder and Code Prover analysis. The following set of options are common between Bug Finder and Code Prover:

- “Target and Compiler”
- “Macros”
- “Environment Settings”
- “Inputs and Stubbing”
- “Multitasking”
- “Coding Standards & Code Metrics”
- “Reporting”, except `Bug Finder and Code Prover report (-report-template)`

You might have to change more of the default options when you run the Code Prover verification because Code Prover is stricter about compilation and linking errors.