Release Notes for Polyspace® Products for C/C++

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Release Notes for Polyspace® Products for C/C++

© COPYRIGHT 2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Contents

R2013a

R2012b

R2012a

R2011b

R2011a

R2010b

R2010a

R2009b

R2009a

R2008b

R2008a

R2013a

Version: 8.5
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Modified Polyspace installer

In R2013a:

- The Polyspace® installer does not create the *Polyspace_Common* folder.
- The Polyspace product does not depend on environment variables, that is, the installer does not create POLYSPACE_* variables.
- The Remote Launcher Manager does not open automatically during the installation process. In addition, if the service or daemon was running before the installation of the new product, the service or daemon is not automatically restarted. You must manually open the Remote Launcher Manager, configure your Polyspace Server, and then start the service or daemon.
- You must manually install Polyspace plug-in (or add-in) files. The Polyspace installer does not automatically activate these files. For example, before you run a verification from the Eclipse™ IDE, you must manually install the Polyspace plug-in file.

The following table gives information about new locations for files and folders.

Binary files	Location in previous release	Location in R2013a
Polyspace verification environment	*Polyspace_Install*\PVE\Polyspace.exe *Polyspace_Install*/PVE/bin/polyspace	*Polyspace_Install*\polyspace\bin\polyspace[.exe]
Polyspace for C/C++	*Polyspace_Install*\Verifier\wbin *Polyspace_Install*/Verifier/bin	*Polyspace_Install*\polyspace\bin
Spooler	*Polyspace_Common*\RemoteLauncher\wbin\PSQueueSpooler.exe *Polyspace_Common*/RemoteLauncher/bin/polyspace-spooler	*Polyspace_Install*\polyspace\bin\polyspace-spooler[.exe]

Binary files	Location in previous release	Location in R2013a
Automatic Orange Tester	*Polyspace_Common*\AutomaticOrangeTester\PSTAutomaticOrangeTester.exe *Polyspace_Common*\AutomaticOrangeTester\bin\polyspace-automatic-orange-tester	*Polyspace_Install*\polyspace\bin\polyspace-automatic-orange-tester[.exe]
Polyspace plug-in for Eclipse	*Polyspace_Common*\PolyspaceForEclipse	*Polyspace_Install*\polyspace\plugin\eclipse
Model Link plug-ins	*Polyspace_Common*\PolyspaceModelLink	*Polyspace_Install*\polyspace\toolbox\pslink
Polyspace plug-in for IBM® Rational® Rhapsody®	*Polyspace_Common*\PolyspaceUMLLink	*Polyspace_Install*\polyspace\plugin\rhapsody
Polyspace add-in for Visual Studio®	*Polyspace_Common*\VisualInterface[2005/2008/2010]	*Polyspace_Install*\polyspace\plugin\msvc\[2008|2010] Visual Studio 2005 is no longer supported
Remote Launcher	*Polyspace_Common*\RemoteLauncher	*Polyspace_Install*\polyspace\bin
Report Generator	*Polyspace_Common*\ReportGenerator	*Polyspace_Install*\polyspace\bin\polyspace-report-generator

For more information, see:

- “Polyspace Software Administration”
- “Install Polyspace Plug-In for Eclipse IDE”
- “Install Polyspace C++ Add-In for Visual Studio”

Improvements to coding rules checker

Ignore header folders without source files option

You can use the `-I` option multiple times to specify folders with header and source files to include in the compilation process. The `-includes-to-ignore` option allows you to exclude some or all of these folders from coding rules checking. In R2013a, if you specify the option `-includes-to-ignore` with the new value `all-headers`, the rule checker excludes folders that contain only header files, that is, folders that contain no source files.

For more information, see "Exclude Include Folders from Rules Checking".

Summary of coding rule violations in verification log

When you run a verification, the software reports only a summary of rule violations in the verification log. The software stores details of rule violations in an XML file within the *`Results`*`/Polyspace-Doc` folder, for example, `MISRA-C-report.xml`. Previously, the software reported details in the verification log.

MISRA C rule behavior changed in R2012b

Rule	Behavior before R2012b	Current Behavior	For more information, see ...
8.9	No warning on undefined objects with `-allow-undef-variables` option.	Because the option `-allow-undef-variables` was removed, undefined variables produce the warning: "Undefined global variable XX"	"Declarations and Definitions"

Verification uses native binary file

In R2013a, the Polyspace verification is run using the native binary file for your computer architecture. Previously, you specified a 32-bit or 64-bit verification through the option `-machine-architecture` *`option_value`*. Now, if you specify `-machine-architecture`, the software ignores the option and generates the following warning:

```
Option  -machine-architecture option_value  is obsolete.
Verification run using native binary file for your
computer architecture.
```

Previously, the default option value (`auto`) specified 32-bit verification. Now, on a Linux® system, the Polyspace verification is 64-bit. You may observe an increase in verification time compared to previous releases. The increase depends on the application being verified and the architecture of your machine.

Note For Linux systems, only the Polyspace 64-bit Client and Server products are supported.

Negative pointer offset

Compatibility Considerations: Yes

Polyspace does not allow the use of pointers with negative offset values, even if the pointers point to allocated memory locations. Consider the following code.

```
typedef struct
{
  int a;
  int b;
  int c;
} s_little_t;

typedef struct
{
  s_little_t s1; // offset 0, size 12
  s_little_t s2; // offset 12, size 12
  int x;         // offset 24, size 4
} s_big_t;


void test(void)
{
  void *addr_lx;
  addr_lx = (void *) malloc (12U);
  assert(addr_lx != 0);

  {
    s_little_t *ly = (s_little_t *)addr_lx;
    ly->a = 1;
    ly->b = 2;
    ly->c = 3;

    {
       char *bz = ((char *)ly) - 12U;  // Negative offset
       s_big_t *bs = (s_big_t *)bz;
       assert(
              ((*bs).s2.c) == 3);         //Red IDP
    }
  }
```

```
}
```

Even though `bs` points to allocated memory, the negative offset of –12 bytes generates a red IDP check.

Compatibility Considerations

Previously, if you specified the options `-allow-pointer-arith-on-struct` and `-size-in-bytes`, the software might have generated a green IDP check. In R2013a, specifying these two options does not change the red IDP check.

As a result of this change, your results may change when compared to previous versions of the software. Some checks may change color, and the selectivity rate of your results may change.

Exact representation of floating point numbers

Polyspace uses the exact value of a representable floating point number during code verification. Consider the floating point value of 1.0. Previously, Polyspace represented this value as a range 0.9999 – 1.0001. Now, Polyspace uses the exact value, that is 1.0.

Option `-function-called-before-main` renamed

The option `-function-called-before-main` has been renamed `-functions-called-before-main`. However, the software continues to recognize the old option name.

Changes to verification results

See “Negative pointer offset” on page 7.

Changes to analysis options

New options

None.

Changes to existing options

Option	For more information, see ...
`-includes-to-ignores`	"Ignore header folders without source files option" on page 5
`-function-called-before-main`	"Option `-function-called-before-main` renamed" on page 10

Options removed

The `-machine-architecture` option has been removed. See “Verification uses native binary file” on page 6.

Polyspace Server for C/C++ Product

Improved Polyspace Metrics security with HTTPS

You can now configure the Polyspace Metrics Web server with a secure HTTPS protocol. This configuration enables encrypted communication between the Polyspace server and the Polyspace Metrics Web interface. See "Configure Polyspace Metrics Web Interface".

R2012b

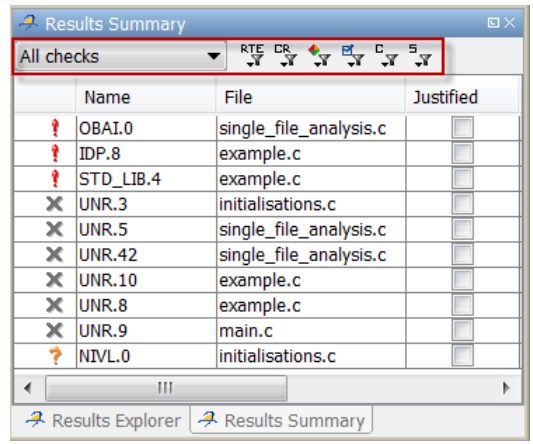
Version: 8.4
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Review of verification results improvement

Check filters in results summary view

The **Results Summary** toolbar provides check filters, which previously were available only in the **Results Explorer** view.



You can apply the check filters from either view. For example, if you filter checks by color and category in the **Results Summary** view, the software also filters out these checks from the **Results Explorer** view. For more information, see Filter Checks.

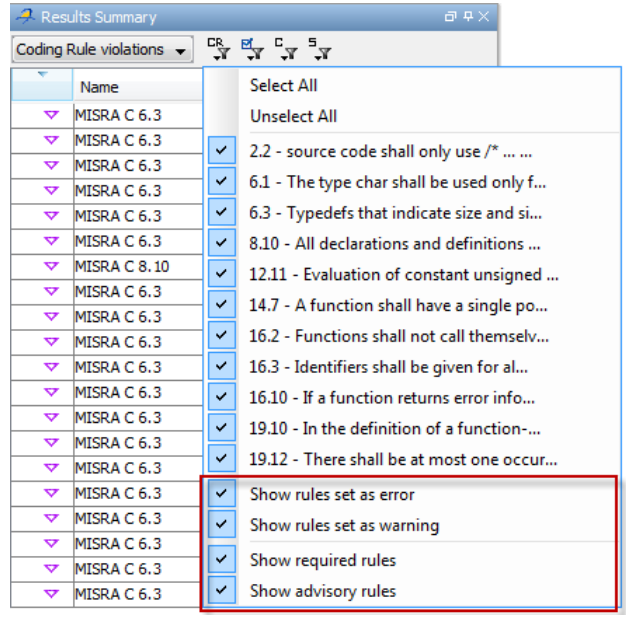
More filters for reviewing coding rule violations

On the **Results Explorer** or **Results Summary** toolbar, the following filters allow you to focus your review on specific kinds of coding rule violations.

Coding rule filters – menu items	Select to display violations of ...
Show rules set as error	Rules assigned the state Error. Any violation of these rules ends the verification after the compilation phase.
Show rules set as warning	Rules assigned the state Warning. Each violation of a coding rule generates a warning, but the verification continues.
Show required rules	Required MISRA C® or MISRA® C++ rules. For information about these rules, refer to MISRA C and MISRA C++ coding standards.
Show advisory rules	Advisory MISRA C or MISRA C++ rules. For information about these rules, refer to MISRA C and MISRA C++ coding standards.
Show shall rules	**Shall** JSF AV C++ rules. These rules are mandatory requirements. For more information about these rules, refer to Joint Strike Fighter Air Vehicle C++ coding standards.
Show will rules	**Will** JSF AV C++rules. These rules are intended to be mandatory requirements, but they do not require verification. For more information about these rules, refer to Joint Strike Fighter Air Vehicle C++ coding standards.
Show should rules	**Should** JSF AV C++ rules, which are advisory rules. For more information about these rules, refer to Joint Strike Fighter Air Vehicle C++ coding standards.

Coding rule filters – menu items	Select to display violations of ...
Show obligatory rules	Obligatory MISRA AC AGC rules. For more information about these rules, refer to *AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation.*
Show recommended rules	Recommended MISRA AC AGC rules. For more information about these rules, refer to *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation.*

To access these filters, on the **Results Explorer** or **Results Summary** toolbar, click the coding rules icon.



In addition, when you select a coding rule violation in the **Results Explorer**, **Results Summary**, or **Source** view, the **Check Details** pane displays the rule type (`req`, `adv`, `shall`, `will`, `should`, `obl`, or `rec`) in the rule description. For example:

```
MISRA C 16.2 (req) Function Recursion shall not call itself, either directly or indirectly.
MISRA C++ 5-2-9 (adv) A cast should not convert a pointer type to an integral type.
JSF C++ 98 (should) Every nonlocal name, except main(), should
be placed in some namespace.
```

For more information, see Examine Coding Rule Violations and Apply Coding Rule Violation Filters.

Justify and comment a group of checks

You can now select a group of checks in either the **Results Explorer** or **Result Summary** view, and then justify and add review information to those checks. For more information, see Review and Comment Checks.

Navigation improvements

In the **Source** view of the Results Manager perspective, if you right-click a local variable, the context menu provides the following features:

- **Search For All References** — Lists all references to the local variable in the **Search** pane. The software also supports this feature for global variables, functions, types, and classes.
- **Go To Definition** — Takes you to the line of code that contains the declaration of the local variable. The software also supports this feature for functions, types, and classes. Previously, this feature was available only for global variables.

For more information, see Exploring Results Manager Perspective.

Variable values in tooltips

The display of variable values in **Source** view tooltips is improved, providing information that narrows the range of the variable. For example, the tooltip might indicate whether the variable values are multiples of a number. The following table has examples of how tooltips display variable range values.

Previously	R2012b
`0 or 2 or 4 or [6 .. 2147483642 (0x7FFFFFFA)]`	`even values in [0 .. 2147483642 (0x7FFFFFFA)]`
`[-56 ..110] or [112 .. 166]`	`even values in [-56 .. 166]`
`[-1265 .. -46] or -23 or 0`	`multiples of 23 in [-1265 .. 0]`
`[0 .. 22] or 44`	`0 or 22 or 44`

You might also see other new types of tooltips, for example:

- For variables:
  - `1008 or 4800 or 14400 or 23040 (values are multiples of 48)`
- For variable pointers:
  - `points to 2 bytes at offset multiple of 8 in [0 .. 64]`
  - `points to 4 bytes at an even offset in [0 .. 20]`
  - `points to 4 bytes at offset 0 or 8 or 24 or 72 (offset is multiple of 8)`

This enhancement might affect the contents of the text file *Your_Project*`_Variable_View.txt` file, which your verification generates in the `\results\Polyspace-Doc` folder. Instead of only an interval list for a variable, you might see, for example, the following conventions in the file:

- multiples of (a constant) in (the interval list)
- even values in (the interval list)
- odd values in (the interval list)

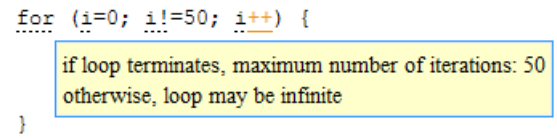
Loop information in tooltips

In the Results Manager **Source** view, the software provides a tooltip for the loop tokens `for` and `while`. The tooltip indicates whether the loop terminates. Consider the following code:

```
int foo1(int random) {
   int i;

   for (i=0; i!=50; i++) {
      if (random)
      i=51;
   }
   return i;
}
```

If you place your cursor over `for`, you see the following tooltip:



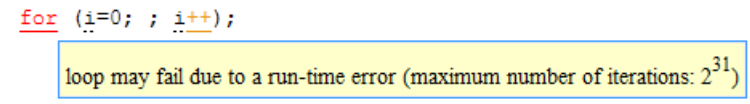
Consider another example:

```
void foo2(void) {
   int i;

   for (i=0; ; i++);

}
```

In this case, the tooltip indicates that the code might produce a run-time error.



Previously, the software could provide information about the maximum number of iterations in a loop through the loop counter tooltip. But this information did not necessarily indicate whether the loop would terminate.

Accuracy improvements for MISRA rules checking
Compatibility Considerations: Yes

For rule 5.7, the rule checker does not report a violation when:

- Different functions have parameters with the same name.
- Different functions have local variables with the same name.
- A function has a local variable that has the same name as a parameter of another function.

Consider the following code:

```
1   int foo(int param) {
2      int result;
3      return result;
4   }
5
6   short bar (float param)
7      short result;
8      return result;
}
```

In the function `bar`, `param` and `result` (line 7) do not generate violations of rule 5.7.

For rules 10.1 and 10.2, the rule checker does not report violations when an implicit conversion is valid and does not change the representation of the value. Consider the following code:

```
1    typedef unsigned short uint16;
2    typedef unsigned int uint32;
3
4    uint32 var = 0;
5
6    uint32 foo(void) {
7      uint16 result;
8      result = result + 1;
9      return result;
10  }
```

Lines 4, 8, and 9 do not generate violations of rule 10.1.

For rules 9.1 and 21.1, the software performs additional post-compilation processing for rule checking.

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations might change when compared to previous versions of the software.

Definition of custom coding rules

You can now define custom coding rules. You can verify that your code complies with these rules by using the Polyspace coding rules checker.

You can check names or text patterns in your source code with reference to custom rules in a text file. You create the text file manually or through the Polyspace verification environment.

Each rule in the text file specifies a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated. For each violation, the software generates a warning or error message in the verification log.

For more information, see Create a Custom Coding Rules File and Activate Custom Rules Checker.

Configuration of C/C++ rule checking

R2012b provides:

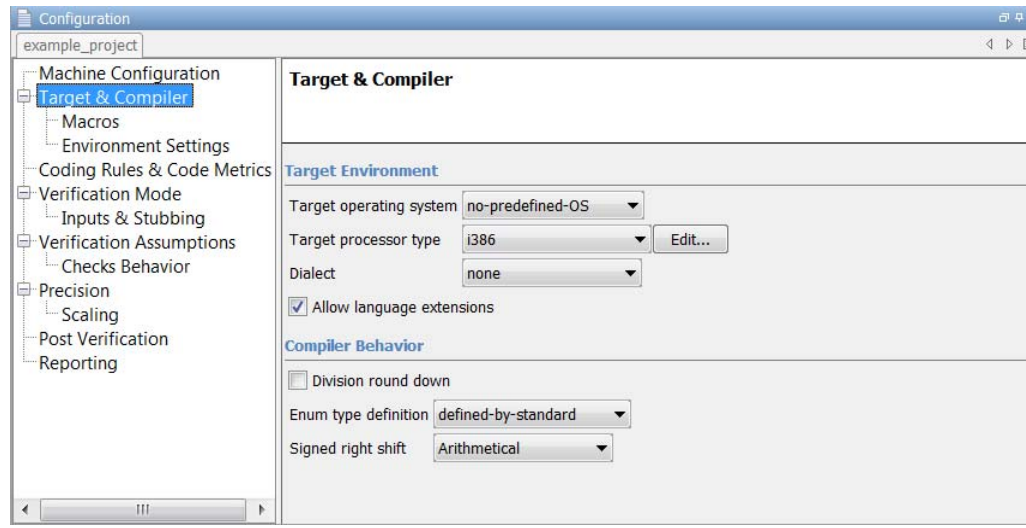
- A new option, `-misra-ac-agc`, with values `OBL-rules`, `OBL-REC-rules`, `all-rules`, `SQO-subset1`, `SQO-subset2`, and `custom`. The first three values allow you to specify checking of obligatory, obligatory and recommended, and all rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*. The other values allow you to check MISRA AC AGC rules that affect selectivity and custom rules that you define.
- The updated option `-misra2`, which allows you to specify checking of required MISRA C coding rules. The option value `AC-AGC-OBL-subset` is no longer supported.
- The updated option `-misra-cpp`, with values `required-rules`, `all-rules`, `SQO-subset1`, `SQO-subset2`, and `custom`. These values allow you to specify checking of required MISRA C++ rules, all MISRA rules, MISRA rules that affect selectivity, and custom rules that you define.
- The updated option `-jsf-coding-rules` with values `shall-rules`, `shall-will-rules`, `all-rules`, and `custom`. These values allow you to specify checking of JSF C++ **Shall**, **Will**, and **Should** rules and custom rules that you define.

For more information, see Coding Rules Setup and Coding Rules & Code Complexity Metrics.

Reorganized Configuration pane

The Project Manager perspective of the Polyspace verification environment provides a reorganized **Configuration** pane that allows improved management of verification options.

The configuration process for code verification consists of different parts, for example, specifying your target environment and compiler behavior. The **Configuration** pane contains a tree with nodes that represent different parts of the configuration process. For example, to choose your target environment and compiler, select the **Target & Compiler** node and then specify your options.



The following options have been removed.

Name in Configuration pane	Command line	What happens in R2012b
Allow non int types for bitfields	`-allow-non-int-bitfield`	If source code contains `bitfield` types that are not `int`, verification does not stop.
Allow undefined global variables	`-allow-undef-variables`	If linkage errors due to undefined global variables occur, verification does not stop.
Allow anonymous unions/structure fields	`-allow-unnamed-fields`	If source code contains unnamed fields within structures and unions, verification does not stop.
Ignore missing header files	`-ignore-missing-headers`	If some include header files are missing, Polyspace generates a warning but tries to continue compiling.
Stub complex functions	`-permissive-stubber`	If Polyspace cannot stub complex functions with parameters and return types that are function pointers, verification does not stop.
Ignore assembly code	`-discard-asm`	If source code contains assembler code, verification does not stop.
Accept integral type conflicts	`-permissive-link`	Verification allows integer type conflicts between declarations and definitions for arguments and return values of functions.

In addition, you can specify the following options from the command line or through the **Configuration > Machine Configuration > Non-official options** field.

Available from command line	Removed from Configuration pane
`-less-range-information`	**Less range information**
`-no-pointer-information`	**No pointer information**
`-keep-all-files`	**Keep all preliminary results files**
`-known-NTC`	**Functions known to cause NTC**
`-asm-begin -asm-end`	**Handle #pragma asm/endasm directives**
`-strict`	**Strict**
`-permissive`	**Permissive**
`-Wall`	**Give all warnings**

Code verification for very large applications

If the source code within your project represents a single application, you might want to verify all the code together. However, if the application is extremely large, the verification might take days.

Polyspace now provides a feature that allows you to:

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

For more information, see Code Verification for Large Applications.

Report content filtering

Previously, you used the MATLAB® Report Generator™ software to apply filters to report templates through only the **Run-time Check Details Ordered by Color/File** component. The software provides a new component, **Report Customization (Filtering)**, which allows you to apply filters from any point of the report hierarchy.

For more information, see Customize Verification Reports.

Parent folder for verification results

You can now specify a parent folder for your verification results through the **Project and result folder** tab of the Polyspace Preferences dialog box. If you do not specify a parent folder, the software uses the active module folder as the parent folder. For more information, see Specify Parent Folder for Results.

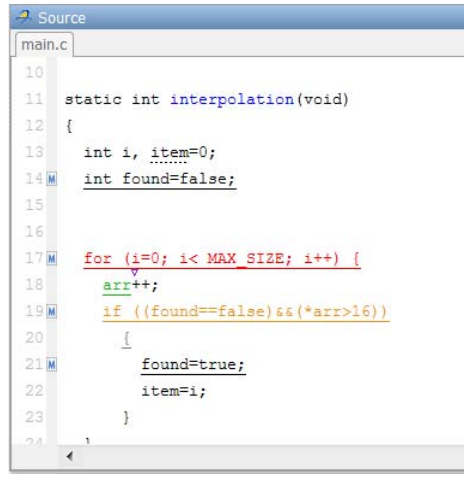
Support for relative paths

The Polyspace Project Manager now supports relative paths outside the project hierarchy. For example, paths for source files in a folder above the project location. Previously, relative paths were supported only for subfolders of the project.

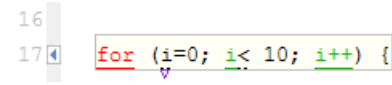
Absolute paths are still supported.

Macro expansion in source code view

You can now view the contents of source code macros in the source code view. A new code information bar displays `M` icons that identify source code lines with macros.



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



If you click the line away from the shaded region, the software displays the normal source code again.

In the **Results Explorer** or **Results Summary** views, if you select a check, the software displays the expanded macro in the source code view.

The expanded macro supports tooltips and checks like any other source code line.

If you right-click any point within the source code view, the context menu has options to display or hide the content of all macros.

Note Previously, you viewed the contents of a macro in the **Expanded Source Code** view. This view is no longer available.

Modifying or removing generic targets

In addition to creating new generic targets, you can now use the Generic target options dialog box to view, modify, or delete generic targets. For more information, see Set Up a Target.

Previously, you modified or deleted generic targets through the **Generic targets** tab in the Polyspace Preferences dialog box. This tab is no longer available.

Improved COR check for function pointer

Compatibility Considerations: Yes

The correctness COR checks that the software performs on function pointers is improved. The following table describes what the software now generates for these checks.

Function pointer scenario	Color of check	Information displayed in Results Manager perspective

Points to valid function	Green	`Function pointer points to a valid function`
		`pointer is not null`
		`function called: foo`

`NULL` function pointer	Red	`Function pointer does not point to a valid function` `pointer is null`

Points to no function	Red	`Function pointer does not point to a valid function`
		`pointer is not null`
		`pointer does not point to any function`

Points to badly typed function (number or types of arguments in function call are not compatible with function definition)	Red	`Function pointer does not point to a valid function`
		`pointer is not null`
		`pointer points to badly-typed function: foo`
		`- error when calling function foo: wrong number of arguments (call has 1 but function expects 0)`

May be `NULL`	Orange	`Function pointer may not point to a valid function`
		`pointer may be null`
		`if pointer is not null, function called: foo`

Function pointer scenario	Color of check	Information displayed in Results Manager perspective
May not point to valid function. Polyspace suggests well-typed functions that may be called through pointer.	Orange	Function pointer may not point to a valid function pointer is not null functions that may be called: {*foo*, *bar*} pointer may have become invalid and point to no valid function
May point to badly typed functions	Orange	Function pointer may not point to a valid function pointer is not null functions that may be called: {*foo*, *bar*} pointer may point to badly-typed function: {*f1*, *f2*, *f3*} - error if function *f1* is called: wrong number of arguments (call has *1* but function expects *0*) - error if function *f2* is called: wrong type of argument (argument *2* of call has type *int* but function expects type *float*) - error if function *f3* is called: wrong type of returned value (function returns type *int* but call expects type *float*)
May point to many badly typed functions	Orange	Function pointer may not point to a valid function pointer is not null functions that may be called: {*foo*, *bar*} pointer may point to badly-typed function: {*f1*, *f2*, *f3*, ...} - error if function *f1* is called: wrong number of arguments (call has *1* but function expects *0*) - error if function *f2* is called: wrong type of argument (argument *2* of call has type *int* but function expects type *float*) - error if function *f3* is called: wrong type of returned value (function returns type *int* but call expects type *float*)

Compatibility Considerations

Previously, a function pointer might generate multiple orange COR checks, for example, a check for wrong type for argument, a check for wrong number of arguments, and a check for wrong return type. Now, the software generates a single check but provides the same information in the **Check Details** pane. As a result of this improvement, you might observe fewer orange checks compared to previous versions of the software.

Permissive function pointer calls
Compatibility Considerations: Yes

By default, Polyspace allows a function pointer to call a function only if both the function pointer and function types are identical. For example, a function with type

```
int f(int*)
```

cannot not be called by a function pointer of type

```
int fptr(void*)
```

To allow calls where the function pointer and function types are not identical, you must specify the new option `-permissive-function-pointer`.

Note With applications that use function pointers extensively, this option can cause a significant loss in performance and a higher number of orange checks as Polyspace has to consider more execution paths.

Previously, the software was permissive by default, allowing calls where function pointer and function types were not identical.

Compatibility Considerations

The default behaviour has changed. If your code contains function pointers that call functions with types that are not identical to those of the function pointers, you might see more red COR checks compared to previous versions of the software. In the verification log, you might see the following message:

```
Warning: -permissive-function-pointer option may be useful
```

If you select one of the new red COR checks, in the **Check Details** pane, you might see information like the following:

```
pointer points to badly typed function: foo
    - error when calling function foo: wrong type of argument (argument n
      of call has type pointer to void but function expects type pointer to data type)
      To work around function pointer incompatibility, rerun verification with
```

```
option -permissive-function-pointer.
```

To return to previous behavior, specify the new option `-permissive-function-pointer` and rerun your verification.

Enhanced stub generation for Standard Library functions (C)

Previously, if there was a mismatch between your declaration of a standard function and the actual standard function, and you did not use the `-D__polyspace_no_function_name` to resolve the mismatch, then verification generated a compilation error. For example, if you declared:

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

instead of:

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

the conflict between your declaration and the Polyspace function definition produced a compilation error.

For this example, if your **Include** folders contain the standard header files, `stdio.h` and `string.h`, the verification recognizes your declaration and does not produce a compilation error. To suppress this feature, specify the option `-D__polyspace_static_types_for_stubs`.

If your **Include** folders do not contain the files `stdio.h` and `string.h`, you can activate the feature with the option `-D __polyspace_adapt_types_for_stubs`.

For more information, see Stubs of libc Functions.

Intermediate verification level support

From the Polyspace verification environment, you can no longer specify the following values for the option `-to`:

- `c-to-il` or `C to Intermediate Language` — If this value is specified in a project configuration file (`.cfg`), the software replaces the value with `c-compile` or `C Source Compliance Checking`.
- `cpp-normalize` or `C++ source normalization` — If this value is specified in a project configuration file (`.cfg`), the software replaces the value with `cpp-compliance` or `C++ source compliance checking`.
- `cpp-link` or `C++ Link` — If this value is specified in a project configuration file (`.cfg`), the software replaces the value with `cpp-compliance` or `C++ source compliance checking`.
- `cpp-to-il` or `C++ to Intermediate Language` — If this value is specified in a project configuration file (`.cfg`), the software replaces the value with `cpp-compliance` or `C++ source compliance checking`.

From the command line, you can still specify these values (`C to Intermediate Language`, `C++ source normalization`, `C++ Link`, and `C++ to Intermediate Language`) for the option `-to`.

Analysis of public methods called by generated main

Previously, when you specified the values `all`, `inherited-all`, `unused`, and `inherited-unused` for the option `-class-analyzer-calls`, the software analyzed both public and protected methods called by the generated main. Now, this option supports four additional values that restrict the analysis to public methods:

- `all-public` — The generated main calls all public methods of the specified classes but does not call protected methods.
- `inherited-all-public` — The generated main calls all public methods of the specified classes and the parents of these classes.
- `unused-public` — With the exception of protected methods, the generated main calls all methods that are not called within the specified classes.
- `inherited-unused-public` — The generated main calls all public methods that:
  - Belong to the specified classes and their parents.
  - Are not called by another method.

For more information, see Methods to call within the specified classes.

DRS file generation for unit-by-unit verification

Previously, you could not automatically generate a single data range specification (DRS) file if your project was configured for a `-unit-by-unit` verification. Now, if the option `-unit-by-unit` is enabled, you can generate a single DRS file containing values that represent the union of DRS values generated for each unit.

Note The DRS file generation functionality is not supported for C++.

For more information, see Data Range Configuration.

Comments for generated DRS files

When you use the Polyspace DRS Configuration dialog box to edit a DRS configuration file, you can enter comments. For example, you can enter comments to justify the values that you specify.

For more information, see Data Range Configuration.

Automatic import of comments and justifications

You can specify the batch option `-import-comments` *`polyspace_results_folder_path`* to automatically import comments and justifications from a previous verification. For more information, see Batch Options.

Storage of temporary files

If you specify the new option `-tmp-dir-in-results-dir`, Polyspace does not use the standard `/tmp` or `C:\Temp` folder to store temporary files. Instead, Polyspace uses a subfolder of the results folder. If the results folder is mounted on a network drive, this action might affect processing speed. Use this option only when the temporary folder partition is not large enough and troubleshooting is required.

For more information, see Batch Options

Changes to verification results
Compatibility Considerations: Yes

Compatibility Considerations

Verification results might change when compared to previous versions of the software. Some checks might change color, and the Selectivity rate of your results might change.

COR check

See:

- “Improved COR check for function pointer” on page 39
- “Permissive function pointer calls” on page 42

Changes to coding rules checker results
Compatibility Considerations: Yes

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations might change when compared to previous versions of the software.

MISRA-C rule improvements

R2012b provides enhanced support, which includes bug fixes, for the following rules:

- Rules 5.5 and 5.7
- Rules 6.1, 6.4, and 6.5
- Rules 8.3 and 8.4
- Rule 9.1
- Rules 10.1 and 10.2
- Rule 11.1
- Rule 12.12
- Rule 16.9
- Rules 17.1, 17.2, and 17.3
- Rule 19.9
- Rule 21.1

See “Accuracy improvements for MISRA rules checking” on page 25.

Removal of Polyspace in One Click

The Polyspace in One Click feature will be removed in a future release.

Changes to analysis options

New options

Option	For more information, see ...
`-import-comments`	"Automatic import of comments and justifications" on page 49
`-misra-ac-agc`	"Configuration of C/C++ rule checking" on page 28
`-permissive-function-pointer`	"Permissive function pointer calls" on page 42
`-tmp-dir-in-results-dir`	"Storage of temporary files" on page 50

Changes to existing options

Analysis options have been reorganized in the Project Manager. See "Reorganized Configuration pane" on page 29.

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-automatic-orange-testers-test-timeout	**Maximum test time**	Unchanged	**Post Verification**	C	
-add-to-results-repository	**Add automatically to results repository**	**Add to results repository**	**Machine Configuration**	C/C++	
-align [8|16|32]			Generic target options dialog box, through **Target & Compiler**	C/C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-allowed-pragma	**Allowed pragmas**	Unchanged	**Coding Rules & Code Metrics**	C	
-allow-language-extensions	**Allow language extensions**	Unchanged	**Target & Compiler**	C	
-allow-negative-operand-in-shift	**Allow negative operand for left shifts**	Unchanged	**Verification Assumptions > Checks Behavior**	C/C++	
-allow-ptr-arith-on-struct	**Enable pointer arithmetic out of bounds of fields**	Unchanged	**Verification Assumptions > Checks Behavior**	C	
-asm-begin -asm-end	**Handle #pragma asm/endasm directives**	Command line only			
-automatic-orange-tester	**Automatic orange tester**	Unchanged	**Post Verification**	C	
-automatic-orange-testers-loop-max-iteration	**Maximum loop iterations**	Unchanged	**Post Verification**	C	
-automatic-orange-testers-tests-number	**Number of automatic tests**	Unchanged	**Post Verification**	C	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-big-endian			Generic target options dialog box, through **Target & Compiler**	C/C++	
-boolean-types	**Effective Boolean types**	Unchanged	**Coding Rules & Code Metrics**	C	
-char-is-16bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-class-analyzer	**Class name**	Unchanged	**Verification Mode**	C++	
-class-analyzer-calls	**Select methods called by the generated main**	**Methods to call within the specified classes**	**Verification Mode**	C++	"Analysis of public methods called by generated main" on page 46
-class-only	**Analyze the class content only**	Unchanged	**Verification Mode**	C++	
-code-metrics	**Calculate code metrics**	**Calculate code complexity metrics**	**Coding Rules & Code Metrics**	C/C++	
-context-sensitivity	**Sensitivity context**	Unchanged	**Precision**	C/C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-context-sensitivity-auto	**Automatic selection for sensitivity context**	**Sensitivity context**	**Precision**	C/C++	
-continue-with-compile-error	**Continue with compile error**	Unchanged	**Target & Compiler > Environment Settings**	C/C++	
-critical-section-begin/end	**Critical section details**	Unchanged	**Verification Mode**	C/C++	
-custom-rules		**Check custom rules**	**Coding Rules & Code Metrics**	C/C++	
-D	**Defined Preprocessor Macros**	**Preprocessor definitions**	**Targets & Compler > Macros**	C/C++	
-data-range-specifications	**Variable/ function range setup**	Unchanged	**Verification Mode > Inputs & Stubbing**	C/C++	
-default-sign-of-char		**Signed**	Generic target options dialog box, through **Target & Compiler**	C/C++	
-dialect		**Dialect**	**Target & Compiler**	C/C++	
-div-round-down	**Division round down**	Unchanged	**Target & Compiler**	C	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-dos	**Disk operating system**	**Code from DOS or Windows file system**	**Target & Compiler > Environment Settings**	C/C++	
-double-is-64bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-entry-points	**Entry points or interruption**	**Entry points**	**Verification Mode**	C/C++	
-enum-type-definition	**Enum type definition**	Unchanged	**Target & Compiler**	C/C++	
-extra-flags	**Other**	**Non-official options**	**Machine Configuration**	C/C++	
-for-loop-index-scope	**Management of scope of 'for loop' variable index**	Unchanged	**Target & Compiler**	C++	
-functions-called-after-loop	**Functions called after loop**	**Termination functions** — Polyspace Model Link™ SL only	**Verification Mode**	C	
-functions-called-before-loop	**Functions called before loop**	**Initialization functions** — Polyspace Model Link SL only	**Verification Mode**	C	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-functions-called-before-main	**First functions called**	**Initialization functions/ methods**	**Verification Mode**	C/C++	
-functions-called-in-loop	**Functions called in loop**	**Cyclic functions** — Polyspace Model Link SL only	**Verification Mode**	C	
-functions-to-stub	**Functions to stub**	Unchanged	**Verification Mode > Inputs & Stubbing**	C/C++	
-green-absolute-address-checks	**Green absolute address checks**	Unchanged	**Verification Assumptions**	C/C++	
-I		**Include folders**	**Target & Compiler > Environment Settings**	C/C++	
-ignore-constant-overflows	**Ignore overflowing computations on constants**	Unchanged	**Verification Assumptions > Checks Behavior**	C/C++	
-ignore-float-rounding	**Ignore float rounding**	Unchanged	**Verification Assumptions**	C/C++	
-ignore-pragma-pack	**Ignore pragma pack directives**	Unchanged	**Target & Compiler**	C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-import-dir	**Import folder**	Unchanged	**Target & Compiler**	C++	
-include	**Include**	Unchanged	**Target & Compiler > Environment Settings**	C/C++	
-includes-to-ignore		**Files and folders to ignore**	**Coding Rules & Code Metrics**	C/C++	
-inline	**Inline**	Unchanged	**Precision > Scaling**	C/C++	
-int-is-32bits			Generic target options dialog box, through **Target & Compile**r	C/C++	
-jsf-coding-rules	**JSF C++ rules configuration**	**Check JSF C++ rules**	**Coding Rules & Code Metrics**	C++	
-k-limiting	**Depth of verification inside structures**	Unchanged	**Precision > Scaling**	C/C++	
-keep-all-files	**Keep all preliminary results files**	Command line only			
-known-NTC	**Functions known to cause NTC**	Command line only			

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-less-range-information	**Less range information**	Command line only			
-lightweight-thread-model	**Reduce task compexity**	Unchanged	**Precision > Scaling**	C	
-little-endian	**Endianness**	Unchanged	**Target & Compiler**	C/C++	
-logical-signed-right-shift		**Signed right shift**	**Target & Compiler**	C	
-long-is-32bits			Generic target options dialog box, through **Target & Compiler**		
-long-is-64bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-long-long-is-64bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-machine-architecture	**Run verification in 32 or 64-bit mode**	Unchanged	**Machine Configuration**	C/C++	
-main	**Visual Studio compliant main**	**Main entry point**	**Verification Mode**	C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-main-generator	**Generate a main**	**Verify module**	**Verification Mode**	C/C++	
-main-generator-calls	**Functions called**	**Functions to call**	**Verification Mode**	C/C++	
-main-generator-writes-variables	**Write accesses to global variables**	**Variables to initialize**	**Verification Mode**	C/C++	
-max-processes	**Number of processes for multiple core system**	Unchanged	**Machine Configuration**	C/C++	
-misra2	**MISRA C rules configuration**	**Check MISRA C rules**	**Coding Rules & Code Metrics**	C	"Configuration of C/C++ rule checking" on page 28
-misra-cpp	**MISRA C++ rules configuration**	**Check MISRA C++ rules**	**Coding Rules & Code Metrics**	C++	"Configuration of C/C++ rule checking" on page 28
-modules-precision	**Specific precision**	Unchanged	**Precision**	C	
-no-automatic-stubbing	**No automatic stubbing**	Unchanged	**Verification Mode > Inputs & Stubbing**	C++	
-no-constructors-init-check	**Don't check members initialization**	**Skip members initialization**	**Verification Mode**	C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-no-def-init-glob	**Do not consider all global variables to be initialized**	**Ignore default initialization of global variables**	**Verification Mode > Inputs & Stubbing**	C	
-no-extern-C	**Overcome link error**	Unchanged	**Target & Compiler > Environment Settings**	C++	
-no-fold	**Optimize huge static initializers**	Unchanged	**Precision > Scaling**	C	
-no-pointer-information	**No pointer information**	Command line only		C/C++	
-no-stl-stubs	**No STL stubs**	Unchanged	**Verification Mode > Inputs & Stubbing**	C++	
-O	**Precision Level**	**Precision level**	**Precision**	C/C++	
-OS-target	**Target operating system**	Unchanged	**Target & Compiler**	C/C++	
-pack-alignment-value	**Pack alignment value**	Unchanged	**Target & Compiler**	C++	
-path-sensitivity-delta	**Improve Precision of interprocedural analysis**	Unchanged	**Precision**	C/C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-permissive	**Permissive**	Command line only			
-pointer-is-24bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-pointer-is-32bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-post-analysis-command	**Command to apply after the end of the verification**	Unchanged	**Post Verification**	C/C++	
-post-preprocessing-command	**Command/ script to apply to preprocessed files**	Unchanged	**Target & Compiler > Environment Settings**	C/C++	
-report-output-format	**Report output format**	**Output format**	**Reporting**	C/C++	
-report-template	**Report template**	**Report template name**	**Reporting**	C/C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-respect-types-in-fields		**Respect types in fields**	**Verification Assumptions**	C/C++	
-respect-types-in-globals		**Respect types in global variables**	**Verification Assumptions**	C/C++	
-retype-int-pointer	**Retype symbols of integer types**	Unchanged	**Precision**	C	
-retype-pointer	**Retype variables of pointer types**	Unchanged	**Precision**	C	
-scalar-overflows-behavior	**Overflows computation mode**	Unchanged	**Verification Assumptions > Checks Behavior**	C/C++	
-scalar-overflows-check	**Detect overflows on**	Unchanged	**Verification Assumptions > Checks Behavior**	C/C++	
-server	**Send to Polyspace Server**	Unchanged	**Machine Configuration**	C/C++	
-sfr-types	**Sfr type support**	Unchanged	**Target & Compiler**	C	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-short-is-8bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-short-long-is-24bits			Generic target options dialog box, through **Target & Compiler**	C/C++	
-size-in-bytes	**Allow incomplete or partial allocation of structures**	Unchanged	**Verification Assumptions > Checks Behavior**	C	
-size-t-is-unsigned-long	**Set size_t to unsigned long**	Unchanged	**Target & Compiler**	C++	
-strict	**Strict**	Command line only			
-support-FX-option-results	**Support managed extensions**	Unchanged	**Target & Compiler**	C++	
-target	**Target processor type**	Unchanged	**Target & Compiler**	C/C++	
-temporal-exclusions-file	**Temporal exclusion point**	**Temporally exclusive tasks**	**Verification Mode**	C/C++	
-timeout	**Verification time limit**	Unchanged	**Precision**	C/C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-to	**To end of**	**Verification level**	**Precision**	C/C++	"Intermediate verification level support" on page 45
-U	**Undefined Preprocessor Macros**	**Undefine preprocessor definitions**	**Targets & Compler > Macros**	C/C++	
-unit-by-unit	**Run a verification unit by unit**	**Run unit by unit verification**	**Verification Mode**	C/C++	
-unit-by-unit-common-source	**Unit by unit common source files**	Unchanged	**Verification Mode**	C/C++	
-variables-written-before-loop	**Variables written before loop**	**Calibration variables** — Polyspace Model Link SL only	**Verification Mode**	C	
-variables-written-in-loop	**Variables written in loop**	**Input variables** — Polyspace Model Link SL only	**Verification Mode**	C	
-Wall	**Give all warnings**	Command line only		C/C++	

Option	Project Manager perspective			Languages Supported	See also
	Previous label	R2012b label	Location on R2012b Configuration pane		
-wchar-t-is	**Management of w_char_t**	Unchanged	**Target & Compiler**	C++	
-wchar-t-is-unsigned-long	**Set wchar_t to unsigned long**	Unchanged	**Target & Compiler**	C++	

Options removed

The following options have been removed:

- `-allow-non-int-bitfield`
- `-allow-undef-variables`
- `-allow-unnamed-fields`
- `-ignore-missing-headers`
- `-permissive-stubber`
- `-discard-asm`
- `-permissive-link`

For more information, see “Reorganized Configuration pane” on page 29.

Polyspace Server for C/C++ Product

Password-protected access to projects in Polyspace Metrics

You can now restrict access to a project by specifying a password:

- When you run a verification with Polyspace Metrics enabled or upload results to the Polyspace Metrics repository.
- After the creation of a project.

For more information, see Protect Access to Project Metrics.

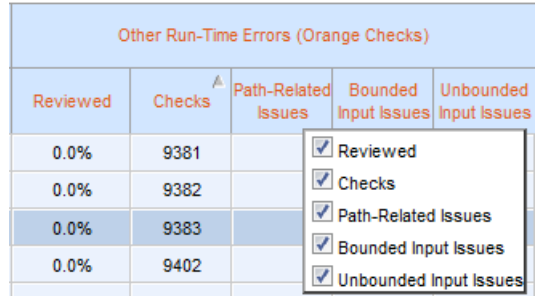
Metrics for level 0 potential errors

Polyspace Metrics now provides metrics for level 0 orange checks, which are potential errors. On the **Run-Time Checks** tab, under **Other Run-Time Errors (Orange Checks)**, the software displays values in the following new columns:

- **Path-Related Issues** — Potential errors that are path-related and not dependent on input values.
- **Bounded Input Issues** — Potential errors that are related to input values bounded by data range specifications (DRS).
- **Unbounded Input Issues** — Potential errors that are related to unbounded input values.

You can now hide or display individual columns. For example:

1. Right-click the column heading **Path-Related Issues**.



2. Clear the check boxes for the columns that you do not want to display. The software hides these columns.

Note This feature supports verification results produced by R2012a. However, if you uploaded your results using Polyspace Metrics R2012a and then upgraded to R2012b, the new columns do not display values. To see values in the new columns, you must upload the verification results again.

R2012a

Version: 8.3
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Single Perspective for Coding Rule Violations and Run-Time Checks

The software now provides, by default, a single Results Manager perspective for coding rule violations and run-time checks.

This single perspective provides the following benefits:

- Easier review of coding rule violations due to better navigation and display functionality, which previously were available for only run-time checks.
- Viewing of coding rules violations with run-time checks, which facilitates analysis of some run-time checks.

You can revert to the previous display format with separate perspectives for Coding Rules and Run-Time Checks:

1. In the Polyspace verification environment, select the **Options > Preferences > Miscellaneous** tab.
2. Select the **Show coding rule violations and run-time checks as separate perspectives** check box.
3. Click **OK**. The software displays a message asking you to restart Polyspace for the change to take effect.

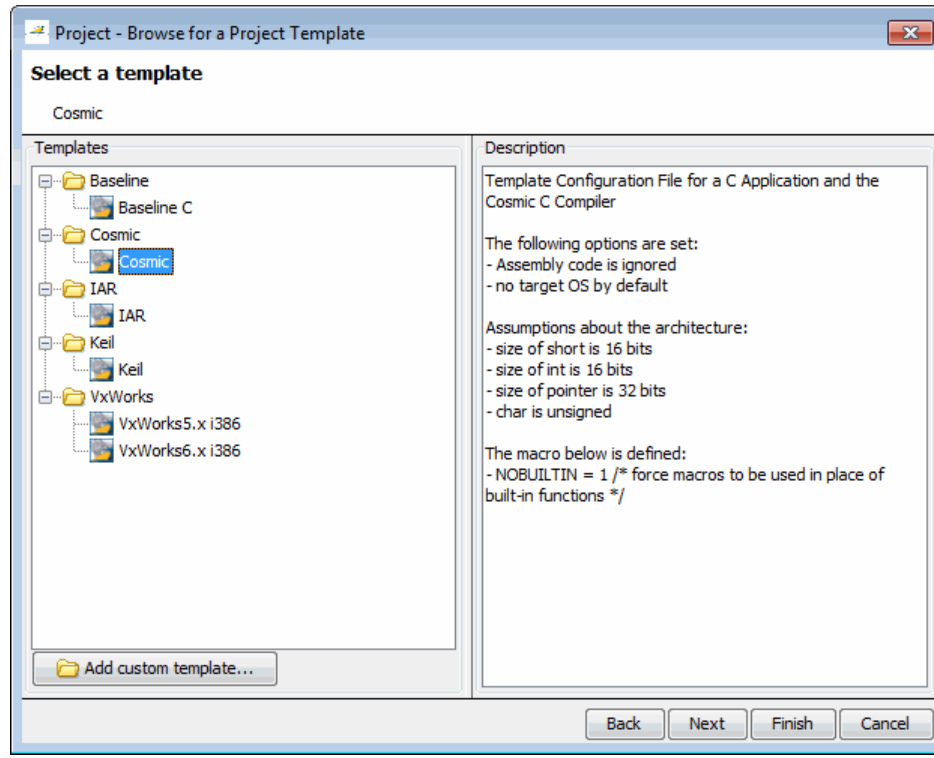
For more information, see Examining Rule Violations.

Compilation Environment Templates

Polyspace software now provides predefined compilation environment templates to help you configure verification projects.

These templates automatically set analysis options for the selected compiler, and help you locate the required include folders.

When creating a new project, you can select a template for your compiler.



For more information, see Creating a Project.

Predefined Templates

Predefined C templates are available for:

- **Baseline C** – Generic C application targeting the i386 architecture
- **Cosmic** – Cosmic C compiler
- **IAR** – IAR compiler
- **Keil** – Keil compiler
- **VxWorks5.x i386** – VxWorks 5.x compiler targeting an i386 architecture
- **VxWorks6.x i386** – VxWorks 6.x compiler targeting an i386 architecture

Predefined C++ templates are available for:

- **Baseline C++** – Generic C++ application targeting the i386 architecture
- **Visual8.0 i386** – C++ Visual 8.0 application and the i386 target
- **Visual8.0 x86_64** – C++ Visual 8.0 application and the x86_64 target
- **Visual9.0 i386** – C++ Visual 9.0 application and the i386 target
- **Visual9.0 x86_64** – C++ Visual 9.0 application and the x86_64 target

Custom Templates

You can also create custom templates from existing Project configurations, and use them to configure future projects.

For more information, see Creating Custom Project Templates.

Suppression of NTC, NTL and UNR Checks Caused by Red Checks

Compatibility Considerations: Yes

Previously, the software would generate red NTC and NTL checks that were a consequence of other red checks. Now, the software does not generate these red checks. However, the software still gives the information that these red checks provided. The software highlights the corresponding call or loop identifier by applying a dashed underline to the identifier.

If the cause of the problem is known, the software provides this information in a tooltip for the underlined call or loop identifier. In addition, when you right-click the identifier, the context menu provides a **Go to Cause** item. Selecting this item takes you to the red check that is the cause.

The software:

- Does not generate gray UNR checks if the cause is a red check
- Still generates red NTC, NTL, and K-NTC checks for a call or loop identifier if the corresponding code contains orange checks.
- Does not generate NTC checks for the functions `exit()` and `abort()`, but provides tooltips for these functions. For example, `exit()`, which does not correspond to an error, terminates the program.

Consider the following code.

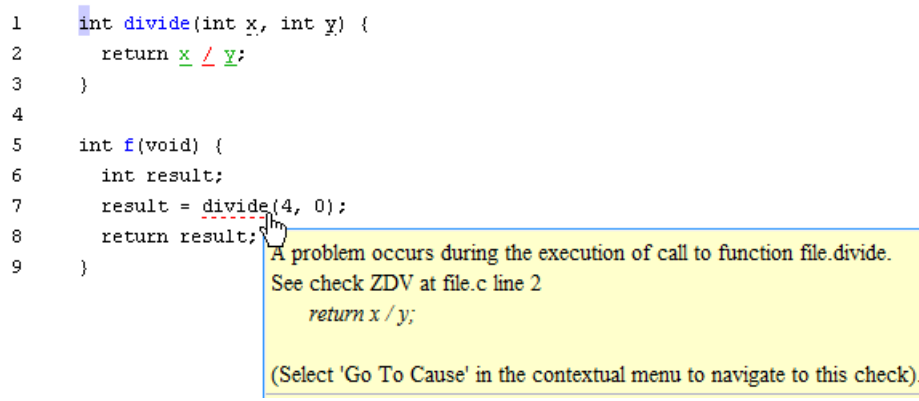
```
1     int divide(int x, int y) {
2       return x / y;
3     }
4
5     int f(void) {
6       int result;
7       result = divide(4, 0);
8       return result;
9     }
```

In R2011b, verification of this code produced the following:

```
1     int divide(int x, int y) {
2       return x / y;
3     }
4
5     int f(void) {
6       int result;
7       result = divide(4, 0);
8       return result;
9     }
```

The red NTC check for the call of the function `divide` (line 7) was a consequence of the red ZDV check for the division operator `/` (line2).

In R2012a, verification of the same code produces the following:



The software does not generate an NTC check, but underlines `divide`. In addition, the tooltip provides information about the problem. If you right-click `divide` and select **Go to Cause** from the context menu, the software takes you to the division operator. This operator has a red ZDV check.

```
1     int divide(int x, int y) {
2       return x / y;
```

Compatibility Considerations

As a result of this new feature, you might observe a significant reduction in the total number of red and gray checks when compared to previous versions of the software.

Probable Cause Information About Red and Orange Checks

With some red and orange checks, the software now provides information about the probable cause of the check. At a certain point in the code, a value is generated that results in a red or orange check at another point in the code. For example:

```
int f() {
  if (..)
    return 0 ;      // This should be the cause of the orange division by zero
  return 1 ;
}
void g() {
  int x = f() ;
  int y = 360 / x ; // Orange check for division by zero
}
```

For this check, in the **Check Details** view, the software lists the probable cause, intermediate events, and the orange check.

For more information, see Viewing Probable Cause Information.

Enhanced MISRA-C Coding Rules Checker

The following improvements have been made:

- Support for the OBL (obligatory) category specified by MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation:
  - Includes new support for rules 3.4, 12.11, 16.10, and 17.1.
  - To check for compliance with the OBL category, specify option `-misra2` with the new value `AC-AGC-OBL-subset`. See MISRA C rules configuration (-misra2).
  - Rule 3.4 requires checking that all pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives by specifying `-misra2` with the new option `-allowed-pragmas`. For example:

    ```
    polyspace-c -misra2 AC-AGC-OBL-subset -allowed-pragmas pragma1,pragma2,pragma3
    ```

    See MISRA C rules configuration (-misra2).
  - With coding rule 16.10, the software does not indicate a violation when the function is `memcpy`, `memmove`, `memset`, `strcpy`, `strncpy`, `strcat`, or `strncat`.
- Enhanced support for rules 16.7 and 19.10:
  - Rule 16.7 — The software generates a warning if a non-`const` pointer parameter is passed to a call of a function that is declared with a `const` pointer parameter.
  - Rule 19.10 — The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter `x`. The software does not generate a warning if `x` appears as `(x)` or `(x,` or `,x)` or `,x,`.
- Enhanced format for XML report files (`MISRA-C-report.xml`, `MISRA-CPP-report.xml`, and `JSF-report.xml`).

For more information, see MISRA C Rules Supported.

Integrated Compilation Assistant

The Compilation Assistant is now enabled by default. During a verification, if the Compilation Assistant detects compilation errors, the verification stops and the software displays errors and possible solutions on the **Output Summary tab**.

To disable the Compilation Assistant, select **Options > Preferences**, which opens the Polyspace Preferences dialog box. Then, on the **Project and result folder** tab, clear the **Compilation Assistant** check box and click **OK**.

The **Configuration** pane has a new tab, **Compiler Settings**, which replaces the **Compilation Assistant** view.

For more information, see Checking for Compilation Problems.

Data Range Specification Enhancements

Previously, a wizard was provided for data range specification (DRS). Now, you can specify data ranges through the Polyspace DRS configuration dialog box, which provides toolbar buttons to:

- Generate a DRS configuration file.
- Update an existing file.
- Convert a DRS text file to an XML file.

For more information, see Specifying Data Ranges for Variables and Functions (Contextual Verification).

Redefinition of Successful Verification

Previously, if a Polyspace verification failed, for example, during `pass1` (`Software Safety Analysis level 1`), the software communicated the failure through log messages and the **Project Browser**. However, if you clicked the *xx*`_LAST_RESULTS.exe` file within the **Project Browser**, the software displayed any results (colored checks) that had been generated by the verification. Now, the software deems a verification successful provided some results have been generated.

Polyspace Report Generator Enhancements

You can:

- Generate multiple reports in the Results Manager perspective. See Generating Verification Reports.
- Customize report templates with MATLAB Report Generator software, which allows you to filter results by:
  - Justification status — Display all, justified, or unjustified checks.
  - Type — Display only listed types of run-time checks.
  - Function — Display only run-time checks from specified functions.

  For more information, see Customizing Verification Reports.

Polyspace In One Click (POC) Enhancement

The POC software has been rewritten. The software that replaces the previous **Send To** functionality now runs verifications without requesting additional settings. See Using Polyspace In One Click.

Note Support for the **Send To** feature will be removed in a future release.

Absolute Addresses

Compatibility Considerations: Yes

Polyspace supports a new check category, ABS_ADDR. The software generates an orange ABS_ADDR check when an absolute address is assigned to a pointer. The software has no information about the absolute address and therefore cannot verify, for example, the address, availability of memory, and initialization of memory.

After generating the orange ABS_ADDR check for the first assignment operation, the software permits memory access to the absolute address. This new behavior produces fewer orange checks in code that contains absolute addresses. After the first assignment operation, IDP and NIV checks for memory access operations are now green. Previously these checks were orange.

A new option, `-green-absolute-address-checks`, is also available. If you know that the absolute addresses in your code are valid, you can specify this option which makes all ABS_ADDR checks green.

For more information, see Absolute Addresses: ABS_ADDR and Green absolute address checks (-green-absolute-address-checks).

Compatibility Considerations

Because of this new check, verification results might change when compared to previous versions of the software. The total number of checks might change as the software now generates an ABS_ADDR check for each conversion of an integer to a pointer.

If you previously created a comment for an orange IDP or NIV check (for example, to explain the check), the comment continues to appear although the check may now be green. In addition, the new ABS_ADDR check does not have a comment. In Polyspace Metrics, information about justifications for the previously orange IDP or NIV checks is lost.

Header Files Without Run-Time Checks and Coding Rule Violations

It is quite common for code to contain header files with library inline functions that are never called. Previously, these files were listed in the **Results Explorer** view, which could slow down your review of results. Now, if header files do not contain run-time checks or coding rule violations, the software does not list these header files in the **Results Explorer** view.

Improved Access to Polyspace Demos

In the Polyspace verification environment, you can now open supplied `Demo` projects through the **Help** menu.

Changes to Verification Results

Compatibility Considerations: Yes

Compatibility Considerations

Verification results might change when compared to previous versions of the software. Some checks might change color, and the Selectivity rate of your results might change.

NTC and NTL Checks

See "Suppression of NTC, NTL and UNR Checks Caused by Red Checks" on page 78.

ABS_ADDR Check

See new "Absolute Addresses" on page 88.

Changes to Coding Rules Checker Results

Compatibility Considerations: Yes

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations might change when compared to previous versions of the software.

New MISRA-C Rules Supported

The coding rules checker now supports the following MISRA-C Rules:

- Rule 3.4
- Rule 12.11
- Rule 16.10
- Rule 17.1

See "Enhanced MISRA-C Coding Rules Checker" on page 82.

MISRA-C: Rule Improvements

Support for the following rules is enhanced:

- Rule 16.7
- Rule 19.10

See "Enhanced MISRA-C Coding Rules Checker" on page 82.

Changes to Analysis Options

New Options

Option	For more information
Allowed pragmas (`-allowed-pragmas`)	“Enhanced MISRA-C Coding Rules Checker” on page 82
Green absolute address checks (`-green-absolute-address-checks`)	“Absolute Addresses” on page 88

Changes to Existing Options

No name changes to existing options.

Options Removed

None

Polyspace Server for C/C++ Product

Enhanced Polyspace Metrics Project Index

The enhanced project index enables you to display projects as categories, which is useful when you have a large number of projects to manage. Now, you can:

- Create multiple-level project categories.
- Move projects between categories by dragging and dropping projects.
- Rename and remove categories. You can remove categories without deleting the projects within the categories. The software moves these projects back to the root level.

For more information, see Organizing Polyspace Metrics Projects.

Redefinition of Successful Verification

Previously, if a Polyspace verification failed, for example, during `pass1` (`Software Safety Analysis level 1`), the software communicated the failure through log messages and the **Project Browser**. However, if you clicked the *xx*`_LAST_RESULTS.exe` file within the **Project Browser**, the software displayed any results (colored checks) that had been generated by the verification. Now, the software deems a verification successful provided some results have been generated.

R2011b

Version: 8.2
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

STD_LIB Check

Compatibility Considerations: Yes

Previously, if the arguments of a function that belonged to the C standard library were not valid, the software would generate a check within the corresponding stub in `__polyspace_stdstubs.c`. In addition, the check category (visible in the procedural entities view) did not indicate a link to the standard library.

Now, Polyspace supports a new check category `STD_LIB`, which allows easier review of run-time errors arising from standard library calls. For example, if a standard library function call does not contain valid arguments, the software generates a red `STD_LIB` check at the function call in your code. The check does not appear in `__polyspace_stdstubs.c`.

For more information, see Stubbing Standard Library Functions.

Compatibility Considerations

Due to the introduction of the `STD_LIB` check, verification results may change when compared to previous versions of the software.

In addition, since the `STD_LIB` check has a different location and aggregates information from multiple checks, you cannot import review comments on standard library checks from previous releases.

For example, if you commented a check in the standard stubs using R2011a results, that comment is lost when you import comments from the R2011a results into R2011b results.

Enhanced MISRA-C Coding Rules Checker

Compatibility Considerations: Yes

The following improvements have been made:

- Compliance with MISRA-C:2004 Technical Corrigendum 1 — For rules 4.1, 5.1, 5.3, 6.1, 6.3, 7.1, 9.2, 10.5, 12.6, 13.5
- New support for rules 6.2, 14.1, and 17.2
- New option `-boolean-types`, which supports rules 12.6, 13.2, and 15.4
- Enhanced support for rules 1.1, 2.3, 5.2, 5.4, 5.5, 5.6, 5.7, 6.4, 8.1, 8.5, 11.1, 11.2, 11.4, 12.3, 12.4, 13.1, 13.7, 15.2, 16.8, 17.3, 17.6, 19.4, 19.15, and 20.1

For more information, see Checking Coding Rules.

Compatibility Considerations

Due to the improvements to the MISRA C coding rules checker, verification results may change when compared to previous versions of the software.

Review Orange Checks that are Potential Run-Time Errors

Previously, there were two modes in which you could review verification results — manual and assistant. For the manual mode, you set the Assistant slider to `Off` and the software displayed all orange checks (in addition to the red and green checks) . With the assistant mode, there were three levels of review — corresponding to settings `1`, `2`, and `3` of the Assistant slider. You could specify the number of orange checks to display through the **Assistant Configuration** tab in the Polyspace Preference dialog box.

Now, Polyspace allows you to review results at five different levels. You can set the Review slider to `0`, `1`, `2`, `3`, or `All`:

- `0` — Display red and gray checks. In addition, display orange checks that are potential run-time errors. On the **Polyspace Preference > Review Configuration** tab, you can specify the type of potential run-time errors that you are interested in. You have the option of not displaying any orange checks.
- `1`, `2`, and `3` — This functionality is unchanged. Display red, gray, and green checks. In addition, display orange checks according to values specified on the **Polyspace Preference > Review Configuration** tab.
- `All` — Display red, gray, green, and all orange checks.

The **Assistant Configuration** tab is renamed the **Review Configuration** tab.

For more information, see Reviewing Results Systematically and Reviewing All Checks.

Display Sources of Orange Checks

The software identifies, where possible, code that is the source of orange checks and provides information about this code on the new **Orange Sources** tab. You can display this tab in the Run-Time Checks perspective, and see the following columns of information:

- **Type** —Type of code element that causes orange check
- **Name** — Name of code element
- **File** — Name of source file
- **Line** — Line number in source file
- **Max Oranges** — Maximum number of orange checks arising from code element
- **Suggestion** — How you can resolve the orange check

For more information, see Viewing Sources of Orange Checks.

With some orange checks, through this new tab, you can add or modify data range specifications to resolve the checks. See Refining Data Range Specifications.

Enhanced Automatic Orange Tester

Previously, you had to run the Automatic Orange Tester manually after the completion of a verification. Now, when you select the Automatic Orange Tester option

- You specify the new option `-automatic-orange-tester`. Polyspace still supports the previous option `-prepare-automatic-tests` in R2011b. However, `-prepare-automatic-tests` will be removed from a future release.
- The softwares runs dynamic tests on the orange code automatically at the end of the verification.
- You can specify test parameters when you configure your verification. If you do not specify test parameters, the software uses default test parameter values.
- If you run a server verification, the software will run the dynamic tests on the server.

The Automatic Orange Tester now also supports the following options:

- `-ignore-float-rounding`
- `-respect-types-in-globals`
- `-respect-types-in-fields`
- `-entry-points`

For more information, see Automatically Testing Orange Code.

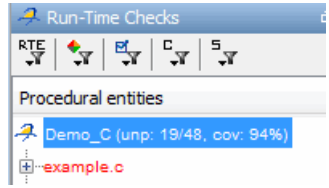
No Gray Checks in Unreachable Code

Compatibility Considerations: Yes

The only gray checks that Polyspace generates now are UNR checks for unreachable branches of code. In addition, Polyspace generates the UNR check only at the highest level of a branch. You no longer see nested UNR checks , that is, UNR checks in sub-branches.

In addition, the software displays two new metrics for the project in the procedural entities view:

- unp — Number of unreachable procedures (functions) as a fraction of the total number of procedures (functions)
- cov — Percentage of elementary operations in executable procedures (functions) covered by verification



These metrics provide:

- A measure of the code coverage achieved by the Polyspace verification.
- Indicators about the validity of your Polyspace configuration. For example, a large unp value and a low cov value may indicate an early red check or missing function call.

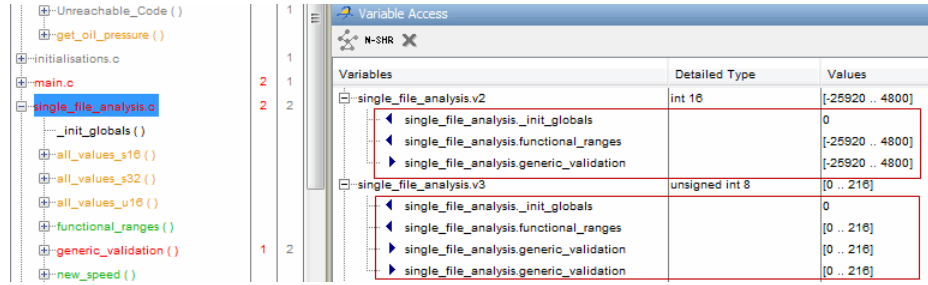
See Results Explorer Tab.

Compatibility Considerations

Due to the removal of non-UNR gray checks and nested UNR checks, verification results may change when compared to previous versions of the software.

Global Variable Range Information

In the **Variable Access** pane, Polyspace displays range information for read and write access operations on global variables within each source file. Previously, the displayed value was the union of all access operations on the global variable within a file. The software did not display range information for individual operations. Now, for global variables that are integers (signed and unsigned) or floating point variables (`float` and `double`), Polyspace also provides range information for the individual access operations from which the union value is obtained.



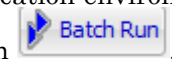
See Variable Access Pane.

Read and Write Access in Dead Code

If a read or write access operation on a global variable lies within dead code, then Polyspace colors the operation gray in the **Variable Access** pane. When you examine verification results, you can hide these operations by clicking the new filter button  . See Variable Access Pane.

Run All Verifications in Project

You can have many verifications within a project, each verification being associated with an active configuration. Previously, you could only run one verification at a time from the Polyspace verification environment (PVE).

Now, if you select a project and click the button 

, Polyspace will run all verifications in the project. See Running a Verification.

Specifying Functions Not Called by Generated Main

You can now specify source files in your project that the main generator will ignore. Functions defined in these source files are not called by the automatically generated main.

Use this option for files containing function bodies, so that the verification looks for the function body only when the function is called by a primary source file and no body is found.

For more information, see Verifying a C Application Without a “Main” in the .Polyspace Products for C/C++ User’s Guide.

Stubbing Specific Functions

You can now specify specific functions that you want the software to stub using the option **Functions to stub** (`-functions-to-stub`).

For more information, see Stubbing in the .Polyspace Products for C/C++ User's Guide.

Changes to Verification Results
Compatibility Considerations: Yes

Compatibility Considerations

Verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Refer to the following sections for information on the specific changes.

Cross-block Critical Sections

In previous releases, the beginning and the end of a critical section must be in same block.

In R2011b, every kind of cross block critical section is supported. However, some constructs may raise a warning.

For example:

```
void foo(void) {
    BEGIN_CS();

    for (;;) {
       END_CS();
    }
}

Warning: Ending critical section cs in a loop in test.foo
```

Function Pointers in `extern const` Structure Stubbed

In previous releases, function pointers in `extern const` structures were not always stubbed, and could result in a COR error.

For example:

```
typedef struct {
int a;
void (*foo)(void);
} AA;

extern const AA aaa;
extern  AA aaaa;

extern const a;
const b;

void foo(void)
{
int bb;
bb = a;
bb = b;

bb = aaa.a;
aaaa.foo();
aaa.foo();   //COR Error in previous releases
}
```

In R2011b, every extern variable will be stubbed regardless of its qualifiers.

Pointers point to the Beginning of Allocated Objects

In previous releases, a pointer in the generated main points anywhere in the allocated buffer, which could cause problems when using DRS.

Allocated objects now point at the beginning of the object.

Changes to Coding Rules Checker Results

Compatibility Considerations: Yes

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations may change when compared to previous versions of the software.

Refer to the following sections for information on the specific changes.

New MISRA-C Rules Supported

The coding rules checker now supports the following MISRA-C Rules:

- Rule 1.2
- Rule 3.1
- Rule 3.4
- Rule 6.2
- Rule 12.11
- Rule 14.1
- Rule 16.10

MISRA-C: Rule 1.1 Messages

Message reported for violations of MISRA-C: Rule 1.1 has been improved.

MISRA-C: Rule 6.3 Improvements

Enforcement of MISRA-C Rule 6.3 has been improved:

- no more violations when the plain char is used
- no more violations when basic types are used for bitfields declarations

In previous releases, the coding rules checker reported a violation on the following code:

```
typedef struct TestData_tag {

unsigned int  IsOK              :1;
unsigned int  IsCounterOK       :1;
unsigned int  IsNew             :1;
unsigned int  UnusedBytes       :13;

} TestData;

void main(void) {

TestData c;

c.IsOK = 1;
}
```

In R2011b, this syntax is allowed.

MISRA-C: Rule 17.6 Improvements

Enforcement of MISRA-C: Rule 17.6 has been improved.

If the address of an object is assigned to another object that may persist after the first object has ceased to exist, a runtime error may occur.

In previous releases, the coding rules checker did not detect a violation in the following example:

```
extern int *vg;
void provide( short int a )
{
   int v1;
   v1 = a;
   vg = &v1;
```

```
}
```

Changes to Analysis Options

New Options

Option	For more information
Functions to stub (`-functions-to-stub`)	"Stubbing Specific Functions" on page 111
`-main-generator-files-to-ignore`	"Specifying Functions Not Called by Generated Main" on page 110
Maximum test time `-dynamic-execution-test-timeout`	"Enhanced Automatic Orange Tester" on page 105
Maximum loop iterations `-dynamic-execution-loop-max-iteration`	"Enhanced Automatic Orange Tester" on page 105
Number of automatic tests `-dynamic-execution-tests-number`	"Enhanced Automatic Orange Tester" on page 105

Changes to Existing Options

The following options have been renamed in R2011a.

New Name (R2011b)	Previous Name (R2011a)	Change
Stub complex functions	**Stub all functions**	GUI name only
Dialect support	**Keil/IAR support**	GUI name only
`-automatic-orange-tester`	`-prepare-automatic-test`	Command-line name and enhanced functionality

Deprecated Options

- **Launch code verification from beginning of** (`-from`)

> **Note** The `-from` option is still accepted when launching a verification in batch mode.

Polyspace Server for C/C++ Product

Running Multiple Verifications Simultaneously
Compatibility Considerations: Yes

If you purchase more than one license for a Polyspace server, you can now configure the server to run multiple verifications at the same time. This can improve the performance of server verifications.

To configure your server to run multiple verifications, open the Remote Launcher Manager, then set the **Number of Polyspace verifications that can run simultaneously on this server** to the number of licenses you have activated for your server.

For more information, see Configuring Polyspace Server Software in the Polyspace Installation Guide.

Compatibility Considerations

If you configure your server to run more than one verification simultaneously, the server will not be able to run verifications using older versions of the software.

For example, if your server has both R2011a and R2011b software installed, you cannot run a verification using the R2011a software.

Polyspace Metrics

Review Changes between Results of Successive Verifications

You can specify a version of a project and review only the differences between verification results of the specified version and the previous verification. See Review New Findings.

File Modules with Quality Levels

If you have projects with two or more file modules in the Polyspace verification environment, by default Polyspace Metrics displays verification results using the same module structure. However, Polyspace Metrics also allows you to create or delete file modules. You can group files into a module and specify a quality level for the module, which applies to all files within the module. This feature allows you to specify different quality levels for your files in the review of verification results. See Creating a File Module and Specifying Quality Level.

Enhanced Graphs and Charts

Polyspace Metrics displays enhanced graphs and charts.

If you specify a range of project versions:

- On the **Summary** tab, **Run-Time Defects** are plotted as separate categories, `High`, `Medium`, and `Low`.
- On the Run-Time Checks tab:
  - Under **Confirmed Defects**, you see separate plots for the categories, `High`, `Medium`, and `Low`.
  - Under **Run-Time Findings**, you see separate plots for red checks, NTC checks, and gray checks.

If you specify a single version of a project, Polyspace Metrics displays file defect information, ordering the files according to the number of defects in each file. Use the new **# items** field to specify the maximum number of files for which information is displayed. See Displaying Metrics for Single Project Version.

R2011a

Version: 8.1
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Code Metrics (New for C++)

Code metric support, including cyclomatic number and other HIS metrics.

Polyspace verification can now generate metrics about code complexity, which are based on the Hersteller Initiative Software (HIS) standard.

These metrics include:

- **Project metrics** – including number of recursions, number of include headers, and number of files.
- **File metrics** – including comment density, and number of lines.
- **Function metrics** – including cyclomatic number, number of static paths, number of calls, and Language scope.

When you run a verification with the `-code-metrics` option enabled, you can view software quality metrics data in the Polyspace Metrics Web interface (**Code Metrics** view), or by running a Software Quality Objectives report from the Polyspace verification environment.

Verification	Project Metrics				File Metrics			Function Metrics										Software Quality Objectives	
	Files	Header Files	Recursion	Direct Recursion	Lines	Lines without Comments	Comment Density	Cyclomatic Complexity	Language Scope	Paths	Calling Functions	Called Functions	Instructions	Call Levels	Function Parameters	Goto Statements	Return Points	Quality Status	Level
V4	6	7	1	1	755	463	FAIL	PASS	PASS	112	PASS	PASS	186	PASS	PASS	0	PASS	FAIL	SQO-1
__polysp																			SQO-1
example					248	136	16.0%	PASS	PASS	45	PASS	PASS	61	PASS	PASS	0	PASS	FAIL	SQO-1
initialisat					108	71	4.0%	PASS	PASS	13	PASS	PASS	20	PASS	PASS	0	PASS	FAIL	SQO-1
main.c					58	45	4.0%	PASS	PASS	6	PASS	PASS	22	PASS	PASS	0	PASS	FAIL	SQO-1
single_fil					140	80	19.0%	PASS	PASS	23	PASS	PASS	36	PASS	PASS	0	PASS	FAIL	SQO-1
tasks1.c					117	82	7.0%	PASS	PASS	13	PASS	PASS	32	PASS	PASS	0	PASS	FAIL	SQO-1
tasks2.c					84	49	11.0%	PASS	PASS	12	PASS	PASS	15	PASS	PASS	0	PASS	FAIL	SQO-1

The software generates numeric values or pass/fail results for various metrics.

For more information, see Software Quality with Polyspace Metrics in the Polyspace Products for C/C++ User's Guide.

Saving Polyspace Metrics Review

Previously, when you saved your project (**Ctrl+S**) after a review of results from Polyspace Metrics, the software would save your comments and justifications both locally and in the Polyspace Metrics repository.

Now, if you save your project (**Ctrl+S**), the software saves your review to a local folder only. A new button  is available on the Run-Time Checks toolbar. If you click this button, the software saves your comments and justifications to a local folder *and* the Polyspace Metrics repository.

This feature allows you to upload your review to the repository only when you are satisfied that your review is, for example, correct and complete.

You can still configure your software to display the previous behavior.

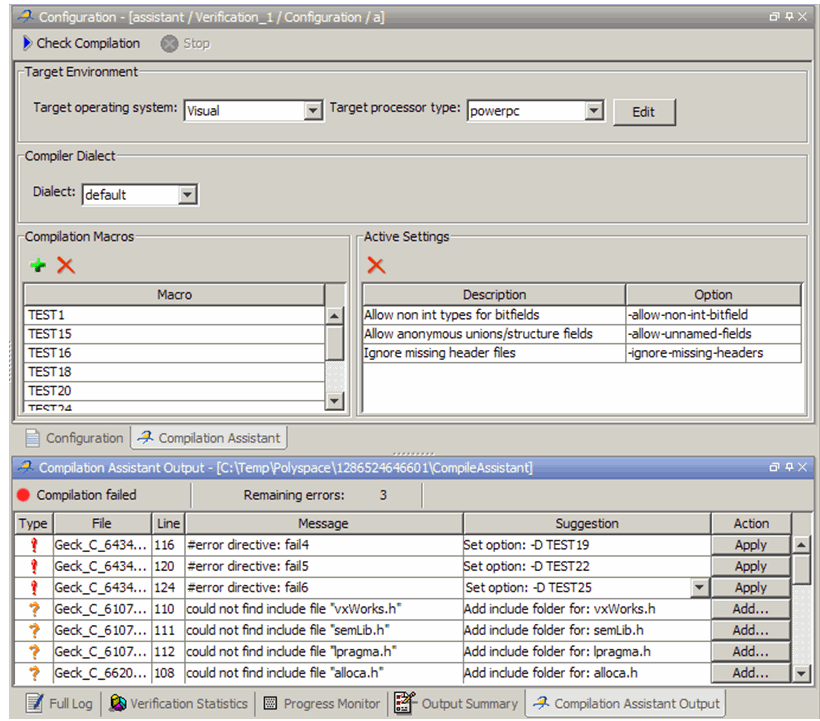
For more information, see Saving Review Comments and Justifications in the Polyspace Products for C/C++ User's Guide.

Compilation Assistant

New Compilation Assistant to ease project configuration (cross-compiler settings).

The Compilation Assistant allows you to check your project for compilation problems before launching a verification. The Compilation Assistant then:

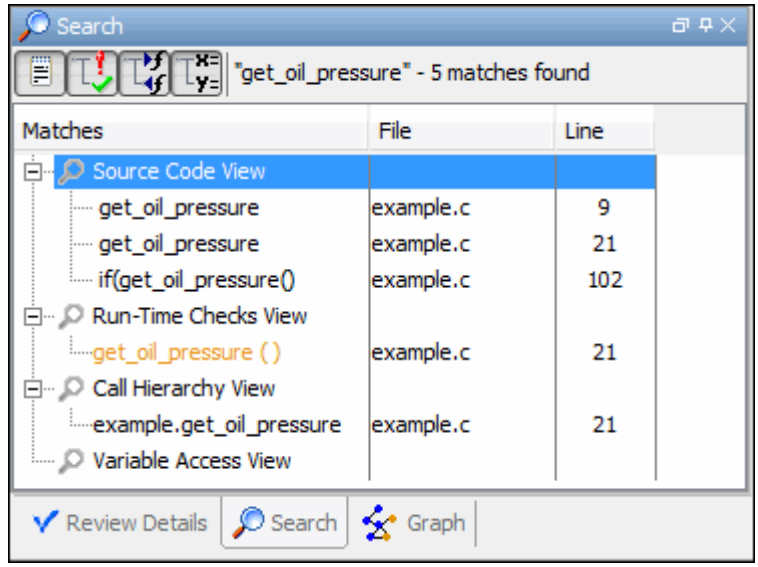
- Automatically detects pre-processing, compilation, and dialect options required for your code (for example, `-I` and `-D`).
- Provides suggestions to solve compilation problems.



For more information, see Checking for Compilation Problems in the Polyspace Products for C/C++ User's Guide.

Improved Search Function

Enhanced search functionality in the Run-Time Checks perspective allows you to perform a search in several views at once (call hierarchy, variable access, run-time checks and source code), and provides search results in a single “Search” view.



For more information, see Searching Results in Results Manager Perspective in the Polyspace Products for C/C++ User’s Guide.

Back to Source Function in Run-Time Checks Perspective

Improved navigation from the Run-Time Checks perspective to the source code containing a check.

You can now right-click a check in your verification results, and open the source file containing that check.

You can configure the software to open source files in either a text editor, or your IDE.

For more information, see Configuring Text and XML Editors in the Polyspace Products for C/C++ User's Guide.

Structure Fields in Data Dictionary

Distinction of variable fields in the Data Dictionary provides a more accurate Data Dictionary.

The enhanced Data Dictionary:

- Helps locate specific field accesses.
- Provides more information on fields (number of read/write accesses, field type).
- Provides a hierarchical view of structured variables.

For more information, see Variable Access Pane in the Polyspace Products for C/C++ User's Guide.

Overflow Check Customization
Compatibility Considerations: Yes

New options allow you to customize how OVFL checks are handled during verification. You can customize computation through overflow constructions, control the presence of overflow checks, and the dynamic behavior in case of a run-time error.

These options allow you to:

- Not generate OVFL checks on all computations (values are computed the same way processors do).
- Not truncate the value after an OVFL check, and carry on with wrapped values (OVFL check does not impact values during verification).

For more information, see Detect overflows on (-scalar-overflows-checks) and Overflows computation mode (-scalar-overflows-behavior) in the Polyspace Products for C/C++ Reference.

Compatibility Considerations

The option `-detect-unsigned-overflows` (available in previous releases) has been renamed. To achieve the same behavior as the previous option, specify the new option `-scalar-overflows-checks signed-and-unsigned`.

When using the new options, your verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Main Generator Improvements

Compatibility Considerations: Yes

Enhanced main-generator to improve verification results for generated code.

The new main-generator allows you greater control over the behavior of the generated main. New options allow you to generate a main specifically designed for cyclic programs, to support generated code and Model-Based Design. This improves verification results at the subsystem level.

The generated main now has the following behavior:

1. It initializes any variables identified by the option `-variables written-before-loop`.
2. It calls any functions specified by the option `-functions-called-before-loop`. This could be considered an initialization function.
3. It initializes any variables identified by the option `-variables written-in-loop`.
4. It calls any functions specified by the option `-functions-called-in-loop`.
5. It calls any functions specified by the option `-functions-called-after-loop`. This could be a terminate function for a cyclic program.

For more information, see Automatically Generating a Main in the Polyspace Products for C/C++ User's Guide.

Compatibility Considerations

Due to precision improvements, verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

In addition, several Analysis options have been renamed to support the new main generator.

Previous Name (R2010b)	New Name (R2011a)
`-main-generator-writes-variables`	`-variables-written-before-loop`
`-function-called-before-main`	`-functions-called-before-loop`
`-main-generator-calls`	`-functions-called-in-loop`

If you have any scripts that use the old options, update them to reflect the new names.

Verification Time Limit

You can now specify a time limit for verifications using the `-timeout` option. If the verification does not complete within the specified time, the verification fails.

For more information, see Verification time limit (-timeout) in the Polyspace Products for C/C++ Reference.

Continue Verification with Compile Errors

You can now specify that a verification continues even if some source files do not compile, using the option `-continue-with-compile-error`.

Functions that are used but not specified are stubbed automatically.

If a source file contains global variables, you may also need to select the option `-allow-undef-variables` to enable verification.

For more information, see Continue with compile error (-continue-with-compile-error) in the Polyspace Products for C/C++ Reference.

Precision Improvements
Compatibility Considerations: Yes

Improved precision on arrays and functions, resulting in less orange checks.

The precision improvements effect:

- `NIV`, `NIVL`, `NIP`, and `IRV` checks
- array cells
- boolean decision graphs
- various other constructs

Compatibility Considerations

Verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Permissive Mode Set By Default

Compatibility Considerations: Yes

Permissive verification mode is now set by default for new projects. This reduces the number of compilation errors for verifications launched with default settings.

The following options are now set by default:

- `-discard-asm`
- `-allow-non-in-bitfields`
- `-permissive-link`
- `-allow-undef-variables`
- `-allow-unnamed-fields`
- `-allow-negative-operand-in-shift`
- `-allow-language-extensions`

If you want to use stricter compilation settings, you can select them in the project configuration.

Compatibility Considerations

When using the default options, your results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Default Project Location

On Windows systems, the default project location has changed.

The default project location is now in `My Documents`. Previously, the default location was defined in the user profile.

Variable Range Inconsistency between Variable Access Pane and Tooltips

The range given for a variable in the Variable Access Pane (Variables View) can differ from the range given by tooltips on the reads of a variable in the Source code view. The range provided by the tooltip will be wider than the range given in the Variables View.

This difference is due to imprecision in the tooltip. Use the variable range that the Variables view provides.

For example:

- Variables View states that variable *X* is in range `[0..4000]`
- Tooltip on a read of *X* states that the range is `[0,7000]`.

In this case, you should accept `[0..4000]` as the range. The tooltip range is caused by imprecision that may be fixed in future releases.

Visual Studio Integration

New Visual Studio import tool allows you to automatically extract some Polyspace settings from a Visual Studio project file.

This tool can help you:

- Locate source files, include folders and preprocessing directives
- Set some Polyspace Visual Studio specific options

For more information, see *Importing Visual Studio Project Information into Polyspace Project* in the *PolySpace® Products for C++ User's Guide*.

Product Name Change in Files and Folders

Compatibility Considerations: Yes

The Polyspace product name has changed from "PolySpace" to "Polyspace" in R2011a. This change impacts the name of all files and folders created by the software.

For example:

- `PolySpace-Doc` folder has changed to `Polyspace-Doc`
- `PolySpace_xxxx.log` file has changed to `Polyspace_xxxx.log`

Compatibility Considerations

If you have existing folders that use the previous product name (for example, `PolySpace/PolySpace_Common`) the R2011a installation will continue to use these existing folders. However, any files or folders created during or after installation will use the new name.

If you have any shortcuts or scripts that are case-sensitive, you should update them to use the right name.

Visual Studio Support

Added support for Microsoft® Visual Studio 2010.

For more information, see the Polyspace Installation Guide.

Eclipse IDE Support

Added support for Version 3.6 of the Eclipse IDE.

For more information, see the Polyspace Installation Guide.

License Manager Support

The License Manager for Polyspace products has been upgraded to FlexNet® 11.9.

You may need to upgrade your FlexNet server and daemon.

For more information, see Polyspace License Installation in the Polyspace Installation Guide.

Changes to Verification Results

Compatibility Considerations: Yes

Compatibility Considerations

Verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Refer to the following sections for information on the specific changes.

Certain COR Checks Changing to OVFL

In previous releases, certain types of overflow errors were reported as COR checks instead of OVFL checks. For example:

```
typedef long int32;
extern int32 random(void);


int32 func(int32 a, int32 b)
{
int32 res = 0;
if (random()) {
res =  a/b;    // COR changing to OVFL in R2011a
}
return res;
}
```

These checks are now reported as OVFL, which will impact check statistics when compared to previous releases.

COR Checks on Function Pointer

In previous releases, verification reported a COR check on function pointer when the parameter type of function pointer is `void*`. For example:

```
typedef void (*func)(void*);

void foo(int *p) { *p=1; }

void bar(void)
{
int a;
func A = foo;
A(&a);
}
```

In R2011a, verification considers that type `void *` is compatible with all other pointer types.

This may result in changes to the color of `COR` checks. The Call graph may also been impacted. It can also have an impact on performance and precision (more calls considered).

NIV Check on Local Volatile Variables

The behavior of NIV checks for local volatile variables has changed.

In previous releases, the NIV for a local volatile variable was always orange. In R2011a, verification allows local volatile variables to behave just like other variables – if they are initialized in the code, the NIV is green.

Polyspace verification considers that the hardware can bring a value (so NIV cannot be red) but will not de-initialize. Therefore, if the variable is initialized by the code, it is green.

OVFL Checks on Assignment

By default, verification no longer reports `OVFL` checks on assignment, for example:

```
uc = ~uc;
```

The total number of checks in your results may change when compared to previous releases.

If you want the verification to report these types of checks, you can use the option `-detect-overflows-on-operator-not` to retain the previous behavior.

Precision Improvements for NIV Checks

Improved precision on `NIV`, `NIVL`, `NIP`, and `IRV` checks.

Precision Improvements on Arrays and Functions

Improved precision on arrays and functions.

Compilation Errors for Classes without Constructors

In previous releases, a compilation error occurs when you use the options `-unit-by-unit` or `-class-analyzer all` on source code containing classes with no user defined or compiler generated constructor.

In R2011a, this behavior changes as follows:

- No compilation error occurs.
- When using the options `-unit-by-unit` or `-class-analyzer all`, if a class has no constructor, all of its members are randomly initialized to the full range.
- When using the option `-class-analyzer custom-class-list`, if a class among the custom-class-list has no constructor, the verification does not initialize the class members in order to highlight NIV/NIP on accessing the class members (which means that this class instance cannot be constructed).
- A warning is displayed in the log file.

Changes to Coding Rules Checker Results

Compatibility Considerations: Yes

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations may change when compared to previous versions of the software.

Refer to the following sections for information on the specific changes.

MISRA C Rule 12.1 – Parentheses for Operand of Unary Operator.

In previous releases, the coding rules checker could incorrectly report a violation of MIRSA C Rule 12.1 for the operand of a unary operator. For example:

```
Y1 =  (U1 * U2) -0.366;       // Passes 12.1
Y2 =  (-1 * (0.366)) + (U1 * U2);   // Fails 12.1
Y3 =  -0.366 + (U1 * U2);      // Fails 12.1
4 = 0.366 + (U1 * U2);       // Passes 12.1
```

The MISRA rule states that parentheses are not required for the operand of a unary operator.

The number of violations of Rule 12.1 may decrease when compared to previous releases.

Single Rule Violation Reported Multiple Times

In previous releases, Polyspace Metrics could report more than one violation of a single coding rule in the same location. This occurred when the message of a rule violation was modified, and the same results were uploaded to the Metrics database multiple times.

In R2011a, messages for rule violations that have the same ID and the same location are merged into a single message of only one rule violation

Therefore, the total number of rule violations may be lower in R2011a than in previous releases.

Changes to Analysis Options

New Options

Option	For more information
Variables written in loop (`-variables-written-in-loop`)	"Main Generator Improvements" on page 132
Functions called after loop (`-functions-called-after-loop`)	"Main Generator Improvements" on page 132
Overflow computation mode (`-scalar overflows-behavior`)	"Overflow Check Customization" on page 131
Continue with compile error (`-continue-with-compile-error`)	"Continue Verification with Compile Errors" on page 135
Verification time limit (`-timeout`)	"Verification Time Limit" on page 134

Changes to Existing Options

The following options have been renamed in R2011a.

New Name (R2011a)	Previous Name (R2010b)	Change
Target operating system	**Operating system target for PolySpace stubs**	GUI name only
Ignore assembly code	**Discard Assembly code**	GUI name only
Allow non int types for bitfields	**Allow non-ANSI/ISO C-90 types of bitfields**	GUI name only
Allow undefined global variables	**Continue even with undefined global variables**	GUI name only
Ignore overflowing computations on constants	**Permits overflowing computations on constants**	GUI name only
Allow anonymous unions/structure fields	**Allow un-named Unions/Structures**	GUI name only
Allow negative operand for left shifts	**Do not check the sign of operand in left shifts**	GUI name only

New Name (R2011a)	Previous Name (R2010b)	Change
Ignore missing header files	**No error on missing header file**	GUI name only
Variables written before loop (`-variables-written-before-loop`)	**Write accesses to global variables** (`-main-generator-writes-variables`)	GUI and command-line name See "Main Generator Improvements" on page 132
Functions called before loop (`-functions-called-before-loop`)	**First functions to call** (`-function-called-before-main`)	GUI and command-line name See "Main Generator Improvements" on page 132
Functions called in loop (`-functions-called-in-loop`)	**Function calls** (`-main-generator-calls`)	GUI and command-line name See "Main Generator Improvements" on page 132
Detect overflows on (`-scalar-overflows-checks`)	**Detect overflows on unsigned integers** (`-detect-unsigned-overflows`)	Functionality change GUI and command line name See "Overflow Check Customization" on page 131

In addition, the default settings for some **Permissive** options have changed.

Deprecated Options

None.

Polyspace Server for C/C++ Product

Code Metrics (New for C++)

Code metric support, including cyclomatic number and other HIS metrics.

Polyspace verification can now generate metrics about code complexity, which are based on the Hersteller Initiative Software (HIS) standard.

These metrics include:

- **Project metrics** – including number of recursions, number of include headers, and number of files.
- **File metrics** – including comment density, and number of lines.
- **Function metrics** – including cyclomatic number, number of static paths, number of calls, and Language scope.

When you run a verification with the `-code-metrics` option enabled, you can view software quality metrics data in the Polyspace Metrics Web interface (**Code Metrics** view), or by running a Software Quality Objectives report from the Polyspace verification environment.

Verification	Project Metrics				File Metrics			Function Metrics										Software Quality Objectives	
	Files	Header Files	Recursion	Direct Recursion	Lines	Lines without Comments	Comment Density	Cyclomatic Complexity	Language Scope	Paths	Calling Functions	Called Functions	Instructions	Call Levels	Function Parameters	Goto Statements	Return Points	Quality Status	Level
V4	6	7	1	1	755	463	FAIL	PASS	PASS	112	PASS	PASS	186	PASS	PASS	0	PASS	FAIL	SQO-1
__polysp																			SQO-1
example					248	136	16.0%	PASS	PASS	45	PASS	PASS	61	PASS	PASS	0	PASS	FAIL	SQO-1
initialisat					108	71	4.0%	PASS	PASS	13	PASS	PASS	20	PASS	PASS	0	PASS	FAIL	SQO-1
main.c					58	45	4.0%	PASS	PASS	6	PASS	PASS	22	PASS	PASS	0	PASS	FAIL	SQO-1
single_fil					140	80	19.0%	PASS	PASS	23	PASS	PASS	36	PASS	PASS	0	PASS	FAIL	SQO-1
tasks1.c					117	82	7.0%	PASS	PASS	13	PASS	PASS	32	PASS	PASS	0	PASS	FAIL	SQO-1
tasks2.c					84	49	11.0%	PASS	PASS	12	PASS	PASS	15	PASS	PASS	0	PASS	FAIL	SQO-1

The software generates numeric values or pass/fail results for various metrics.

For more information, see Software Quality with Polyspace Metrics in the *PolySpace Products for C++ User's Guide*.

Saving Polyspace Metrics Review

Previously, when you saved your project (**Ctrl+S**) after a review of results from Polyspace Metrics, the software would save your comments and justifications both locally and in the Polyspace Metrics repository.

Now, if you save your project (**Ctrl+S**), the software saves your review to a local folder only. A new button  is available on the Run-Time Checks toolbar. If you click this button, the software saves your comments and justifications to a local folder *and* the Polyspace Metrics repository.

This feature allows you to upload your review to the repository only when you are satisfied that your review is, for example, correct and complete.

You can still configure your software to display the previous behavior.

For more information, see Saving Review Comments and Justifications in the Polyspace Products for C/C++ User's Guide.

Automatic Comment Import for Server Verifications

When you download results from the Polyspace server, the software now automatically imports any comments from results in the destination folder into the downloaded results (except for verifications using the option `-add-to-results-repository`).

As a result of this change, you can now download intermediate results for a verification running on the Polyspace server, and add or edit comments on those results. When you later download the final results, your comments are preserved.

You can also download and comment on a single unit of a unit-by-unit verification, even if other units are still pending in the server queue. When you download the final results (which overwrites the earlier results), your comments are preserved.

License Manager Support

The License Manager for Polyspace products has been upgraded to FlexNet 11.9.

You may need to upgrade your FlexNet server and daemon.

For more information, see Polyspace License Installation in the Polyspace Installation Guide.

R2010b

Version: 8.0
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Polyspace Graphical User Interface

Redesigned Polyspace graphical user interface replaces the Launcher and Viewer modules with a single, unified interface called the Polyspace verification environment (PVE).

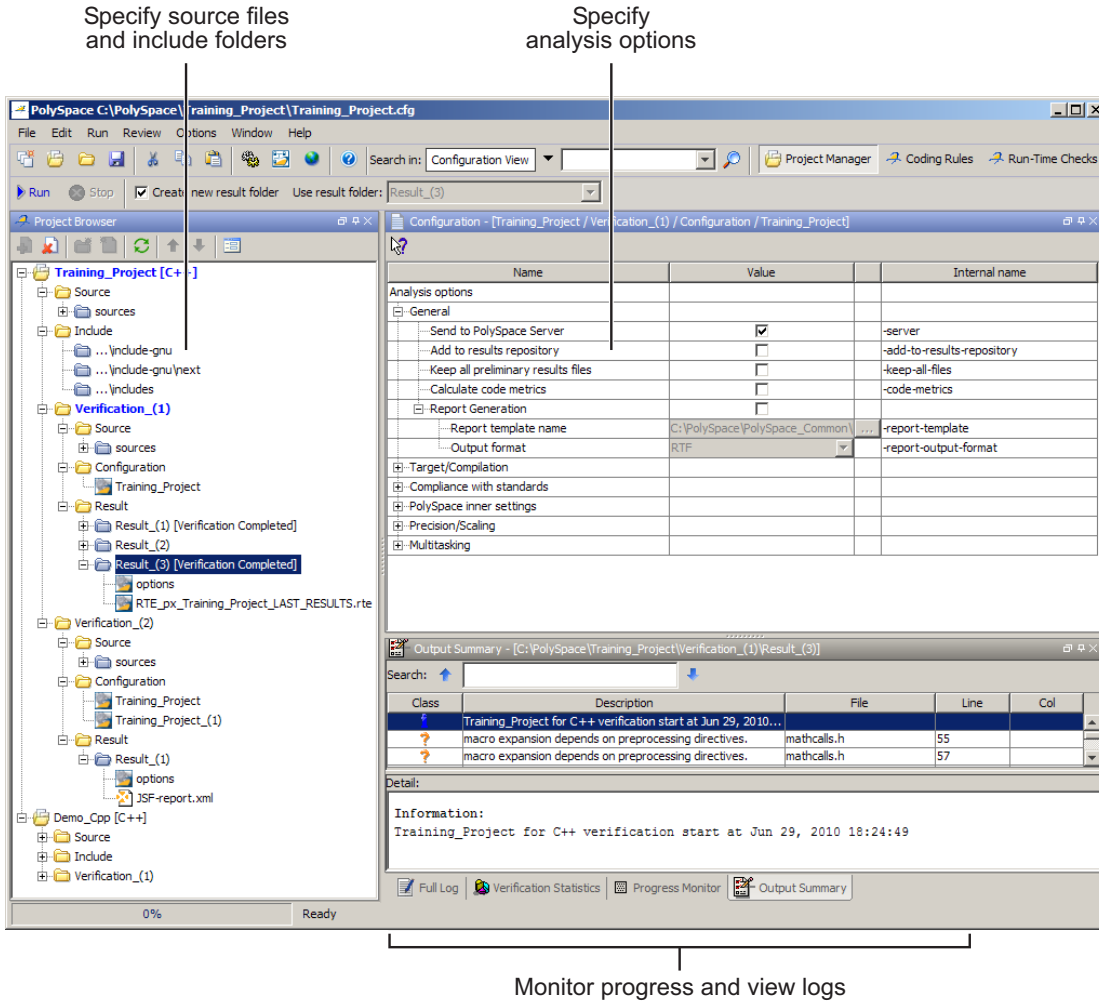
You use the Polyspace verification environment to create Polyspace projects, launch verifications, and review verification results. The new interface also enables you to provide comments in the source code or in the results.

The Polyspace verification environment consists of three perspectives:

- "Project Manager Perspective" on page 160
- "Coding Rules Perspective" on page 161
- "Run-Time Checks Perspective" on page 162

Project Manager Perspective

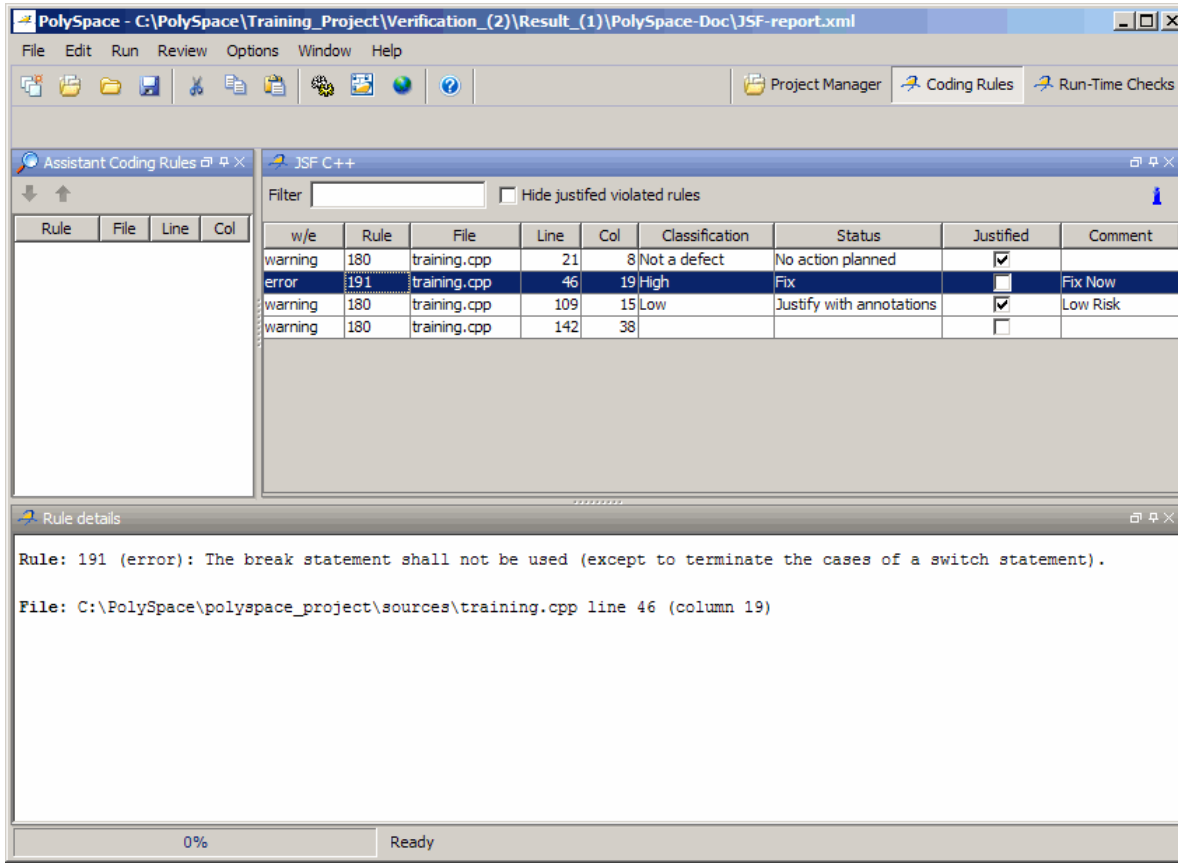
The Project Manager perspective allows you to create projects, set verification parameters, and launch verifications.



For information on using the Project Manager perspective, see Setting Up a Verification Project in the <trademark class="registered">PolySpace</trademark> Products for C User's Guide or Polyspace Products for C++ User's Guide.

Coding Rules Perspective

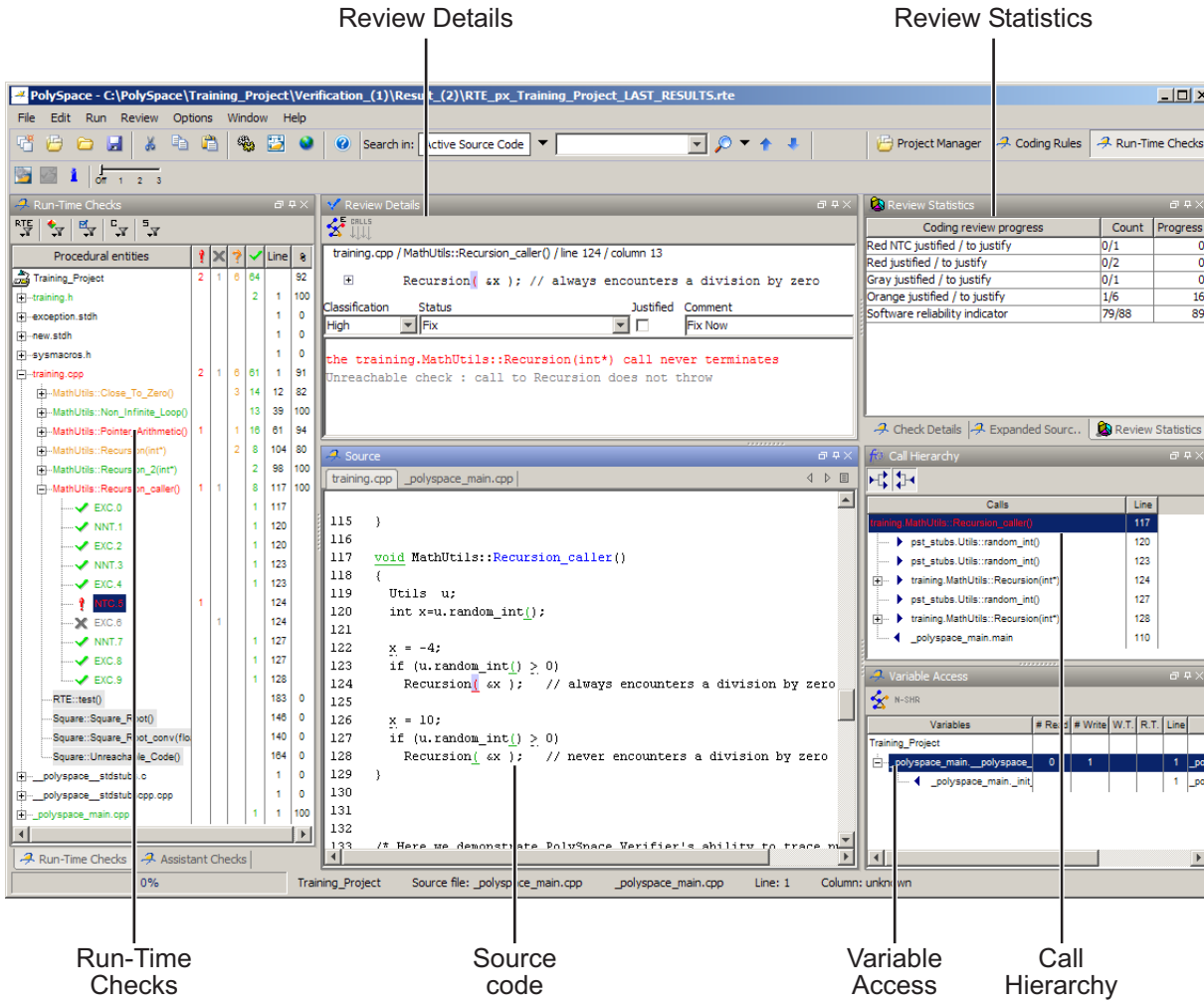
The Coding Rules perspective allows you to review results from the Polyspace coding rules checker, to check compliance with established coding standards.



For information on using the Coding Rules perspective, see Checking Coding Rules in the Polyspace Products for C/C++ User's Guide or Checking Coding Rules in the Polyspace Products for C++ User's Guide.

Run-Time Checks Perspective

The Run-Time Checks perspective allows you to review verification results, comment individual checks, and track review progress.



For information on using the Run-Time Checks perspective, see Reviewing Verification Results in the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Permissiveness on File and Folder Names

Polyspace software now allows space characters in the names of Projects, source files, and folders, as well as in option arguments.

In addition, multiple source files with the same name are now allowed.

Note Non-ASCII characters in file names are not supported.

MISRA C++ Coding Rules Support

Enhanced MISRA C++ checker supports all statically enforceable MISRA-C++ coding rules.

Polyspace software can now check all possible C++ programming rules defined by the MISRA C++ coding standard. The Polyspace MISRA C++ checker provides messages when MISRA C++ rules are not respected. Most messages are reported during the compile phase of a verification.

Note The Polyspace MISRA C++ checker is based on MISRA C++:2008 – "Guidelines for the use of the C++ language in critical systems." For more information on these coding standards, see http://www.misra-cpp.com.

For more information, see Checking Coding Rules, in the Polyspace Products for C++ User's Guide.

Coding Rules Checker Enhancements

The coding rules checker for MISRA C, MISRA C++, and JSF C++ coding standards has been enhanced as follows:

- You can now set all supported coding rules to any state: `Error`, `Warning`, or `Off`.
- The **Files and Folders to ignore** (`-includes-to-ignore`) option now supports the keyword "`all`," allowing you to exclude all included files from coding rules checking.
- The new Coding Rules perspective allows you to review and categorize coding rule violations, and provide comments in the results to justify violations.
- The MISRA C checker now allows you to automatically select two recommended subsets of coding rules (`SQO-subset1`, and `SQO-subset2`), in addition to creating a custom subset.

For more information, see Checking Coding Rules in the Polyspace Products for C/C++ User's Guide or Checking Coding Rules, in the Polyspace Products for C++ User's Guide.

Code Metrics (for C)

Code metric support, including cyclomatic number and other HIS metrics.

Polyspace verification can now generate metrics about code complexity, which are based on the Hersteller Initiative Software (HIS) standard.

These metrics include:

- **Project metrics** – including number of recursions, number of include headers, and number of files.
- **File metrics** – including comment density, and number of lines.
- **Function metrics** – including cyclomatic number, number of static paths, number of calls, and Language scope.

When you run a verification with the `-calculate-code-metrics` option enabled, you can view software quality metrics data in the Polyspace Metrics Web interface (**Code Metrics** view), or by running a Software Quality Objectives report from the Polyspace verification environment.

Verification	Project Metrics				File Metrics			Function Metrics										Software Quality Objectives	
	Files	Header Files	Recursion	Direct Recursion	Lines	Lines without Comments	Comment Density	Cyclomatic Complexity	Language Scope	Paths	Calling Functions	Called Functions	Instructions	Call Levels	Function Parameters	Goto Statements	Return Points	Quality Status	Level
V4	6	7	1	1	755	463	FAIL	PASS	PASS	112	PASS	PASS	186	PASS	PASS	0	PASS	FAIL	SQO-1
__polysp																			SQO-1
example					248	136	16.0%	PASS	PASS	45	PASS	PASS	61	PASS	PASS	0	PASS	FAIL	SQO-1
initialisat					108	71	4.0%	PASS	PASS	13	PASS	PASS	20	PASS	PASS	0	PASS	FAIL	SQO-1
main.c					58	45	4.0%	PASS	PASS	6	PASS	PASS	22	PASS	PASS	0	PASS	FAIL	SQO-1
single_fil					140	80	19.0%	PASS	PASS	23	PASS	PASS	36	PASS	PASS	0	PASS	FAIL	SQO-1
tasks1.c					117	82	7.0%	PASS	PASS	13	PASS	PASS	32	PASS	PASS	0	PASS	FAIL	SQO-1
tasks2.c					84	49	11.0%	PASS	PASS	12	PASS	PASS	15	PASS	PASS	0	PASS	FAIL	SQO-1

The software generates numeric values or pass/fail results for various metrics.

For more information, see Software Quality with Polyspace Metricsin the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Filtering Orange Checks Caused by Input Data (New for C++)

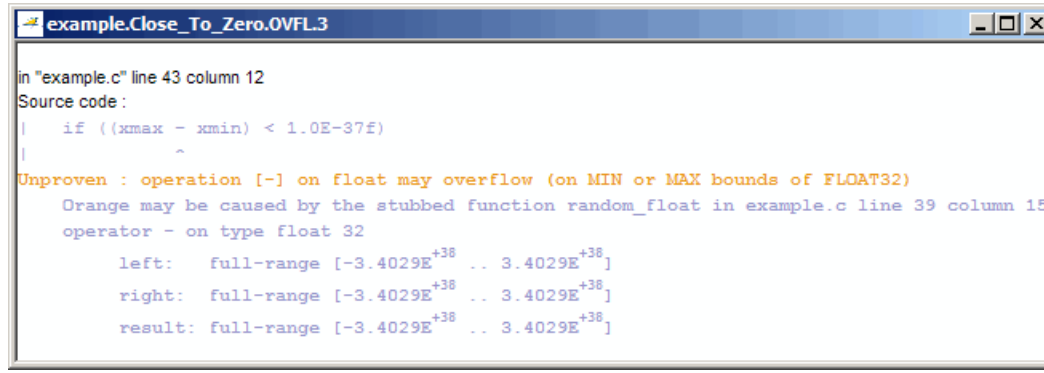
Polyspace verification now identifies orange checks caused by input data for C++ code, in addition to C code. The software provides additional information on these orange checks, and allows you to hide them in the Run-Time Checks perspective.

Note Although this type of orange check could reveal a bug, they usually do not.

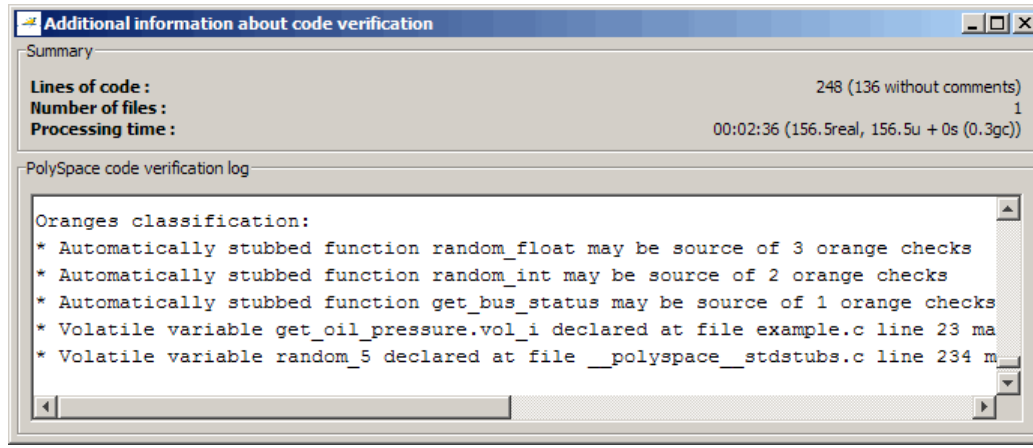
Verification can identify orange checks caused by:

- Stubs
- Main-generator calls
- Volatile variables
- Extern variables
- Absolute address

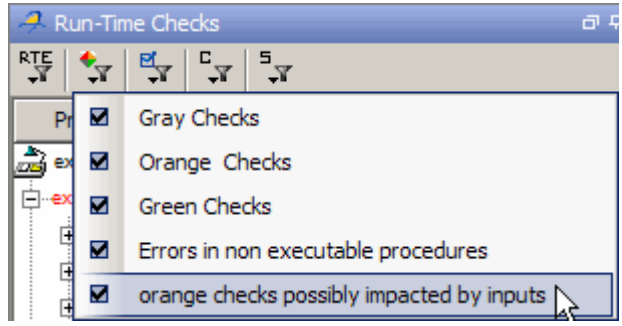
When the software identifies this type of orange check, the Run-Time Checks perspective provides information on its cause.



The Polyspace code verification log file also lists possible sources of imprecision for orange checks.



In addition, you can now hide these types of orange checks in the Run-Time Checks perspective. When using Expert mode, click the **Color filter** icon, then clear the **Orange checks possibly impacted by inputs** option.



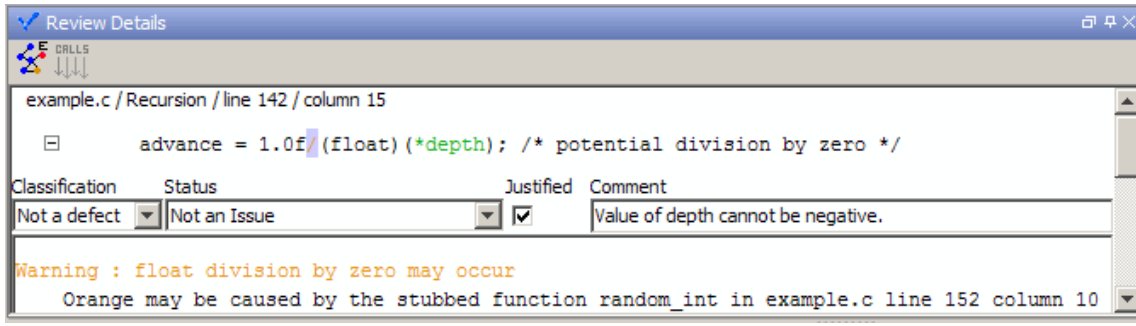
The software hides orange checks impacted by inputs.

For more information, see Working with Orange Checks Caused by Input Data in the Polyspace Products for C++ User's Guide.

New Options to Classify Run-Time Checks and Coding Rules Violations

Compatibility Considerations: Yes

When reviewing results in the Run-Time Checks perspective or the Coding Rules perspective, the software now provides additional options for classifying checks



After you review the check, you can specify the following:

- **Classification** – Select an option to describe the seriousness of the issue.
- **Status** – Select an option to describe how you intend to address the issue.
- **Justified** – Select the check box to indicate that you have justified this check or rule violation.
- **Comment** – Enter additional information about the check

The software provides predefined values for Classification and Status. You can also define your own statuses.

In addition to reviewing checks through the user interface, you can place comments in your code that highlight and categorize checks identified in previous verifications. The software displays the information that you provide within your code comments, and marks the checks as **Justified**.

For more information, see Reviewing and Commenting Checks in the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Compatibility Considerations

The syntax for code comments has changed to reflect the new options for categorizing checks.

The syntax for run-time checks is now:

```
/* polyspace<RTE:RTE1 : [Classification] : [Status] > [Comment] */
```

The syntax for coding-rule violations is now:

```
/* polyspace<JSF:Rule1 : [Classification] : [Status] > [Comment] */
```

If you placed comments in your code using the previous syntax, the comments will still appear in your results, but the text may be displayed in different columns.

For more information on code comments, including full syntax, see Highlighting Known Coding Rule Violations and Run-Time Errors in the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Japanese and Korean Text in Comments

Japanese and Korean characters are now supported for comments in results review.

For more information, see Reviewing Checks Progressivelyin the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Pointer Information in the Run-Time Checks Perspective

Enhanced ToolTip messages on pointers to improve understanding of problems with the pointer.

For example, messages on offset in the allocated buffer now indicate if the pointer is inside its bounds, in addition to giving raw numbers.

For more information, see Using Pointer Information in Results Manager Perspective in the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Main Generation in C++
Compatibility Considerations: Yes

Enhanced main generation options in C++ allow you to use both main generator and class analyzer modes at the same time (the options `-class-analyzer` and `-main-generator-calls` can be used simultaneously).

In addition, the options **Select methods called by the generated main** (`-class-analyzer-calls`), and **Function calls** (`-main-generator-calls`) are enhanced to provide more flexibility in configuring what functions are called. by the generated main.

The default behavior of the main generator is now as follows:

- If you set the **Class name** (`-class-analyzer`) option to `all` or `custom`, and set `-class-analyzer-calls`, then the option `-main-generator-calls` is automatically set to `unused`, unless you explicitly set another value for `-main-generator-calls`.
- Setting the **Function calls** (`-main-generator-calls`) option to `unused`, `all`, or `custom` automatically sets `-class-analyzer` to `none`, unless you explicitly set the `-class-analyzer` option.

For more information, see Generate a main (-main-generator)in the Polyspace Products for C++ Reference.

Compatibility Considerations

If you use scripts that specify a value for the option `-class-analyzer-calls`, you may need to update your scripts to reflect the new option arguments. The new syntax is:

```
-class-analyzer-calls [ all | unused | inherited_all |
  inherited_unused | custom ]
```

Where:

- `all` corresponds to the previous argument "`default`."
- `inherited_unused` corresponds to previous argument "`inherited`."

- `inherited_all` means every inherited methods will be called by the generated main.

Multiple Functions Called Before Main

The option **First functions to call** (`-function-called before main`) now accepts a list of multiple functions, instead of just a single function.

For more information, seeFunctions called after loop (-function-called-after-loop) in the Polyspace Products for C/C++ Reference or Generate a main (-main-generator)in the Polyspace Products for C++ Reference.

Support for C99 Extensions (C)

Partial support of C99 extensions.

A new option, `-allow-language-extensions`, enables verification to accept a subset of common C language constructs and extended keywords, as defined by the C99 standard or supported by many compilers.

When you select this option, the following constructs are supported:

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

In addition, when you use this option, the software ignores the following extended keywords:

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

For more information, see Allow language extensions (-allow-language-extensions)in the Polyspace Products for C/C++ Reference.

New Target Processor Support (C)

Added support for 64-bit target.

The Target processor type (`-target`) option now supports the target x86_64, allowing the verification to emulate 64–bit processors.

For more information, see Predefined Target Processor Specifications in the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Default Target Processor
Compatibility Considerations: Yes

The default setting of the Target processor type (`-target-processor`) option has changed from `SPARC` to `i386`.

Compatibility Considerations

If you launch verifications without specifying a value for this option, the default value has changed. Therefore, your verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Default Operating System Target
Compatibility Considerations: Yes

The default setting of the Operating system target for Polyspace stubs (`-OS-target`) option has changed from `Solaris` to `Linux`.

Compatibility Considerations

If you launch verifications without specifying a value for this option, the default value has changed. Therefore, your verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Include Folders Added to Verification by Default

Compatibility Considerations: Yes

Polyspace software now automatically adds the following standard include folders after any includes you specify:

- *PolySpace_Install*`/Verifier/include/include-gnu`
- *PolySpace_Install*`/Verifier/include/include-gnu/next`

The path to these folders will be printed in the log file at the beginning of the compilation.

Compatibility Considerations

The total number of checks in your verification may change when compared to previous releases, if you did not previously include these folders.

Operating System Support

Added support for the Windows® 7 operating system.

Solaris™ operating system is no longer supported for new installations.

For more information, see the Polyspace Installation Guide.

Changes to Verification Results
Compatibility Considerations: Yes

Compatibility Considerations

Verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Refer to the following sections for information on the specific changes.

New NIP Check on Pointer to Member Function

New NIP check introduced on variables corresponding to a pointer to member function when verifying the pointer.

Previously, pointers to member functions were translated into a structure composed of 4 fields. In this release, these 4 fields are checked and the information is merged into a NIP.

For example (in R2010a and earlier):

```
struct A {
  virtual void f() { } ;
  void g() {} ;
};

int main()
{
```

```
  A a ;
  void (A::*pmf)() ;
  volatile int alea ;

  if (alea)
    assert(pmf != 0) ; // RED NIV located on '(' => expected a NIP

  if (alea)
    (a.*pmf)();   // no red, only grey

  if (alea)
    pmf = &A::f ;
  else
    pmf = &A::g ;

  assert(pmf != 0) ; // spurious info on '(', green NIV instead of NIP

  (a.*pmf)();  //


}
```

In R2010b, some NIV checks may change to NIP checks (on pointer to member function) The Selectivity rate of your results may change when compared to previous versions of the software.

Generated Main Calls in the Main Loop and init Function

The call of a function given to the option `-function-called-before-main` is now removed from the main-generator loop.

In previous releases, when an "init" function was called before the main loop, it was also called in the main loop. For example:

```
void main(void)
{
    /* ****************************************** *
    * Initialization of global variables with random *
    * ****************************************** */
```

```
    /* ****************************** *
    * Call of initialization function *
    * ****************************** */
    {

        /* call it */
        init();
    }
    while (PST_TRUE())
    {
        /* ***************** *
        * Call of functions *
        * ***************** */
        if (PST_TRUE())
        {

            /* call it */
            init();
        }
        if (PST_TRUE())
        {

            /* call it */
            foo();
        }
    }
}
```

This init function is now removed from the main-generator loop.

The Selectivity rate of your results may change when compared to previous versions of the software.

INF Checks Replaced by Value on Range (C++)

When transforming C checks into C++ checks, the software changes INF checks into a new value on range category (VOBJ) and displays them like other value on range (VOR) information in the Run-Time Checks perspective.

The number of checks in your results may decrease when compared to previous releases.

Value on Range (VOR) Values in `pass0` Results

Verification results now give value on range values (intervals) computed during `pass0`.

In previous releases, value on range values in `pass0` could only be constants or the type `full-range`.

When reviewing `pass0` results, value on range tooltips will now contain more information than in previous releases.

Changes in Behavior of Inline and Sensitivity Context Options

Verification now displays a warning of you specify a nonexistent function as an argument of the options **Inline** (`-inline`) or **Sensitivity context** (`-context-sensitivity`). The option is ignored, and verification continues.

In previous releases, specifying a nonexistent function caused the verification to stop.

Permissiveness on Delete of Pointer to Incomplete Class

Polyspace verification now gives a warning when it detects a delete on a pointer with incomplete class, unless you set the Dialect (`-dialect`) option to `iso`. If you specify the iso dialect, the verification will raise a compilation error.

In previous releases, a delete on a pointer with incomplete class implied a crash, and produced an error. For example:

```
#include <memory>

typedef class BaseClass;
typedef class Container
{
private:
    std::auto_ptr<BaseClass> data;
```

```
public:
    Container(std::auto_ptr<BaseClass> p) : data(p) {};
};
```

In R2010b, this code will be accepted with a warning, except in iso mode, where it will raise a compilation error.

Changes to Coding Rules Checker Results

Compatibility Considerations: Yes

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations may change when compared to previous versions of the software.

Refer to the following sections for information on the specific changes.

MISRA and JSF Violations No Longer Reported on Internal Include Folders

The coding rules checker now ignores the Include folders provided with the product (`include-gnu/` and `include-linux/`).

No violations are reported for identifiers appearing in hidden files, even if these files are hidden in a hard-coded way.

The total number of violations reported by the coding rules checker may decrease when compared to previous releases, since any violations within the include files are no longer reported.

MISRA-C++ Rule 2-10-2 Violations on Type Hidden by Using Directive

The MISRA-C++ checker is more precise on violations of rule 2-10-2, "Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope," when the type is hidden by a using directive on the same type.

For example:

```
#include "misra.h"

namespace ns1 {
  class A    // No Violation since the type A is declared only here.
  {
    A & operator= ( A const & rhs );
  public:
    A ( );
    virtual void bar( ) = 0;

  };

}
using ns1::A;
namespace ns2 {
  class D : public A
  {
  public:
    virtual void foo( ) = 0;
    D ( ) : A()
    {
    }
  };
}
```

In previous releases, the MISRA-C++ checker incorrectly reported a violation on the type `A`.

You may see fewer violations of rule 2-10-2 in MISRA C++ reports, when compared with previous releases.

MISRA-C++ Rules 2-10-4 and 2-10-6 Violations on Templates

The coding rules checker no longer reports violations of MISRA-C++ Rules 2-10-4 “A class, union or enum name (including qualification, if any) shall be a unique identifier,” and 2-10-6 “If an identifier refers to a type, it shall not also refer to an object or a function in the same scope” when the template class is present in the code. A violation is reported only for explicit specialization (which has its own declaration).

You may see fewer violations of rules of 2-10-4 and 2-10-6 in MISRA C++ reports, when compared with previous releases.

MISRA-C++ Rule 3-1-1 Duplicate Violations

The coding rules checker no longer reports duplicate violations of MISRA-C++ Rule 3-1-1 ”It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.”

In previous releases, the coding rules checker sometimes incorrectly reported this violation multiple times on the same function.

You may see fewer violations of rule 3-1-1 in MISRA C++ reports, when compared with previous releases.

MISRA-C++ Rule 3-4-1 Violations on Local Variables

The MISRA-C++ coding rules checker is more precise on violations of Rule 3-4-1, “An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.”

For example, the coding rules checker no longer reports violations of rule 3-4-1 for the following code:

```
volatile int32_t rd;
if (rd != 0) {
  int32_t i;
  {
    int32_t j;
    {
      goto L1;
    }
```

```
    rd = j;
  }
  rd = i;
}
```

In previous releases, the coding rules checker incorrectly reported a violation of rule 3-4-1 for the variable `rd`.

You may see fewer violations of rule 3-4-1 in MISRA C++ reports.

MISRA-C++ Rule 7-4-3 Violations on Assembly Language

The MISRA-C++ checker no longer reports errors for rule 7-4-3, "Assembly language shall be encapsulated and isolated," for certain compliant constructions. For example:

```
void Delay_a ( void )
{
   asm ( "NOP" );   // Compliant
}
```

In previous releases, the MISRA-C++ checker incorrectly reported a violation of rule 7-4-3 for this code.

You may see fewer violations of rule 7-4-3 in MISRA C++ reports.

MISRA-C++ Rule 12-1-1, 12-1-2, and 12-8-2 Violations

The MISRA-C++ checker is more precise on violations of rule 12-1-1, "An object's dynamic type shall not be used from the body of its constructor or destructor," rule 12-1-2 "All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes," and rule 12-8-2 "The copy assignment operator shall be declared protected or private in an abstract class."

Violations of rule 12-1-1 are now reported on destructors. For example:

```
class C2
  {
  public:
    ~C2 ( )
```

```
    {
      typeid ( C2 ); // New 12-1-1 violation reported here
      C2::foo ( );
      foo ( );
      dynamic_cast< C2* > ( this );
    }
    virtual void foo ( );
    C2 ( )
    {
      typeid ( C2 ); // 12-1-1 violation reported
      C2::foo ( );
      foo ( );
      dynamic_cast< C2* > ( this );
    }
  };
```

In addition, violations of these rules are now reported in the following cases:

- On `typeid` on any class with virtual function in itself or in one of its base.
- On `typeid` on pointer this or conversion of pointer this.
- On `dynamic_cast` on pointer this or conversion of pointer this.

For example, in the following code violations are now reported on typeid if the type is struct:

```
struct S2
  {
    ~S2 ( )
    {
      typeid ( S2 ); // New violation reported here
      S2::foo ( );
      foo ( );
      dynamic_cast< S2* > ( this );
    }
    virtual void foo ( );
    S2 ( )
    {
      typeid ( S2 ); // New violation reported here
      S2::foo ( );
```

```
    foo ( );
    dynamic_cast< S2* > ( this );
  }
};
```

In previous releases, the MISRA-C++ checker did not report these violations.

You may see additional violations of rule 12-1-1, 12-1-2, and 12-8-2 in MISRA C++ reports, when compared with previous releases.

JSF Rule AV-136 Violations on Local Variables

The JSF C++ coding rules checker is more precise on violations of Rule 136, "Declarations should be at the smallest feasible scope."

For example, the coding rules checker no longer reports violations of rule 136 for the following code:

```
volatile int32_t rd;
if (rd != 0) {
  int32_t i;
  {
    int32_t j;
    {
      goto L1;
    }
    rd = j;
  }
  rd = i;
}
```

In previous releases, the coding rules checker incorrectly reported a violation of rule 136 for the variable `rd`.

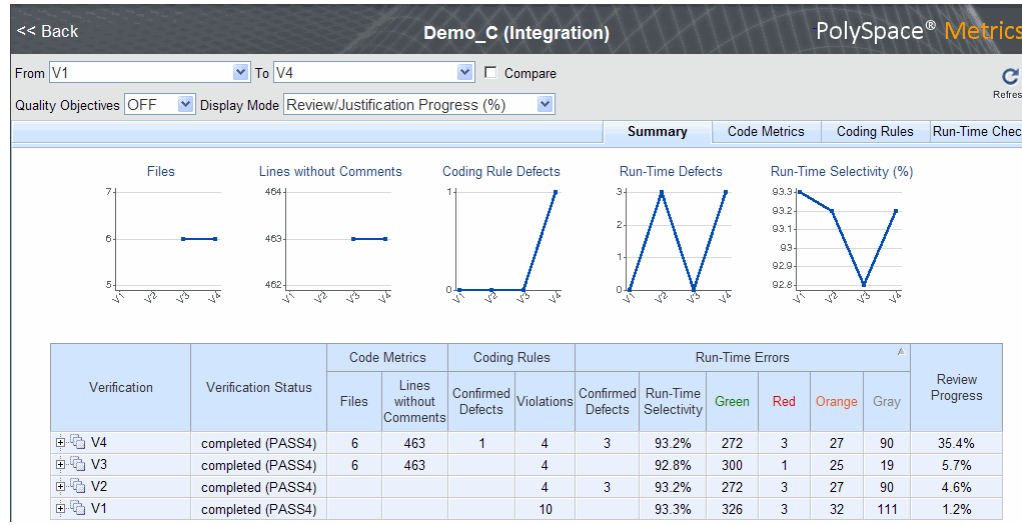
You may see fewer violations of rule 136 in JSF C++ reports.

Polyspace Server for C/C++ Product

Polyspace Metrics Web Interface

A web-based tool for software development managers, quality assurance engineers, and software developers, which allows you to do the following in software projects:

- Evaluate software quality metrics
- Monitor the variation of code metrics, coding rule violations, and run-time checks through the lifecycle of a project
- View defect numbers, run-time reliability of the software, review progress, and the status of the code with respect to software quality objectives.



Verification	Verification Status	Code Metrics		Coding Rules		Run-Time Errors						Review Progress
		Files	Lines without Comments	Confirmed Defects	Violations	Confirmed Defects	Run-Time Selectivity	Green	Red	Orange	Gray	
V4	completed (PASS4)	6	463	1	4	3	93.2%	272	3	27	90	35.4%
V3	completed (PASS4)	6	463		4		92.8%	300	1	25	19	5.7%
V2	completed (PASS4)				4	3	93.2%	272	3	27	90	4.6%
V1	completed (PASS4)				10		93.3%	326	3	32	111	1.2%

In addition, if you have the Polyspace Client™ for C/C++ product installed on your computer, you can drill down to coding rule violations and run-time checks in the Polyspace verification environment. This allows you to:

- Review coding rule violations
- Review run-time checks and, if required, classify these checks as defects

For more information, see Software Quality with Polyspace Metrics in the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Automatic Verification

Configure verifications to start automatically and periodically, for example, at a specific time every night. At the end of each verification, the software stores results in a results repository and updates the metrics for your software project. You can also configure the software to send you an email at the end of the verification. This email contains links to results, compilation errors, run-time errors, or processing errors.

For more information, see Specifying Automatic Verification in the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Operating System Support

Added support for the Windows 7 operating system.

Solaris operating system is no longer supported for new installations.

For more information, see the Polyspace Installation Guide.

R2010a

Version: 7.2
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

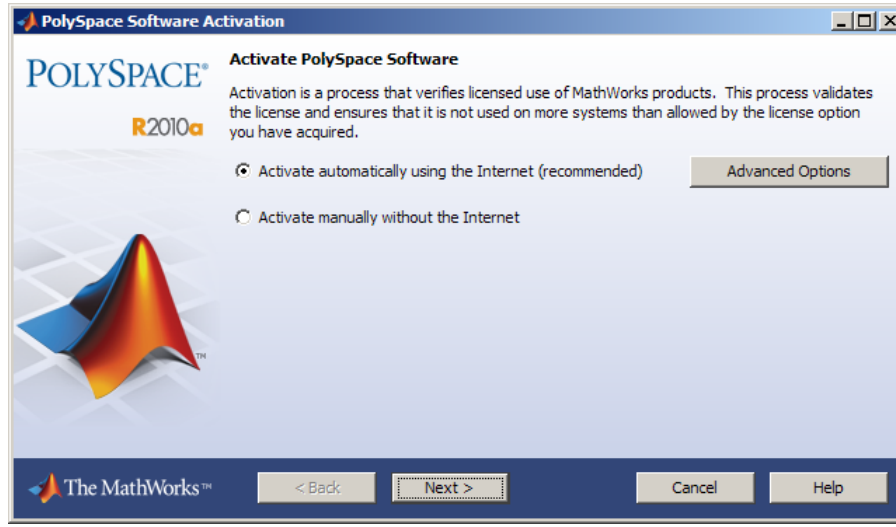
License Activation

Polyspace products now support the MathWorks software activation mechanism.

Activation is a process that verifies licensed use of MathWorks® products. The process validates your product licenses. You must complete the activation process before you can use Polyspace software.

Note If you are using Designated Computer (Individual) licenses, you must activate the license for each Polyspace system individually. However, if you are using Concurrent licenses for multiple Polyspace systems, you do not need to activate each Polyspace system. You activate the license once (for the FLEXnet license server), then provide license files for each Polyspace system.

The easiest way to activate the software is to log in to your MathWorks Account during installation. At the end of the installation process, the Polyspace Software Activation dialog box opens.



Follow the prompts in the Polyspace Software Activation dialog box to complete the activation process.

If you do not have a MathWorks account, you can create one during the activation process. To create an account, you must have an Activation Key, which identifies the license you want to install and activate.

If your Polyspace system is not connected to the internet, you can access the MathWorks License Center on a computer with internet access, activate your license, and download a license file for transfer to your Polyspace system. If you do not have access to a computer with an Internet connection, contact Customer Support.

For more information on how to activate your software, see Activating Polyspace Softwarein the Polyspace Installation Guide.

For more information on software activation, including frequently asked questions, refer to the MathWorks Web site:
www.mathworks.com/support/activation/polyspace.html

MISRA C++ Checker

Polyspace software can now analyze your C++ code to check compliance with the MISRA C++ coding standard.

The Polyspace MISRA C++ checker provides messages when MISRA C++ rules are not respected. Most messages are reported during the compile phase of a verification.

The MISRA C++ checker can check 167 of the 183 statically enforceable MISRA C++ coding rules.

Note The Polyspace MISRA C++ checker is based on MISRA C++:2008 – "Guidelines for the use of the C++ language in critical systems." For more information on these coding standards, see http://www.misra-cpp.com.

For more information, see *Checking Coding Rules*, in the *PolySpace Client/Server for C++ User Guide*.

Source Code Comments

Polyspace software now allows you to place comments in your code that provide information about known coding rule violations and run-time errors. You can use these comments to

- Hide or highlight known coding rule violations.
- Highlight and categorize previously identified run-time errors.

This information can then make the review process quicker and easier by allowing you to focus on new coding rule violations and run-time errors. .

When you review verification results, the Viewer displays comments on individual checks. You can then skip these commented checks, or simply use them as additional information during your review.

The coding rules log in the Launcher displays comments regarding coding rules. You can use these comments to filter out commented violations from the results, or simply to provide additional information on specific violations.

For more information, see Highlighting Known Coding Rule Violations and Run-Time Errors in the *PolySpace Products for C User's Guide*.

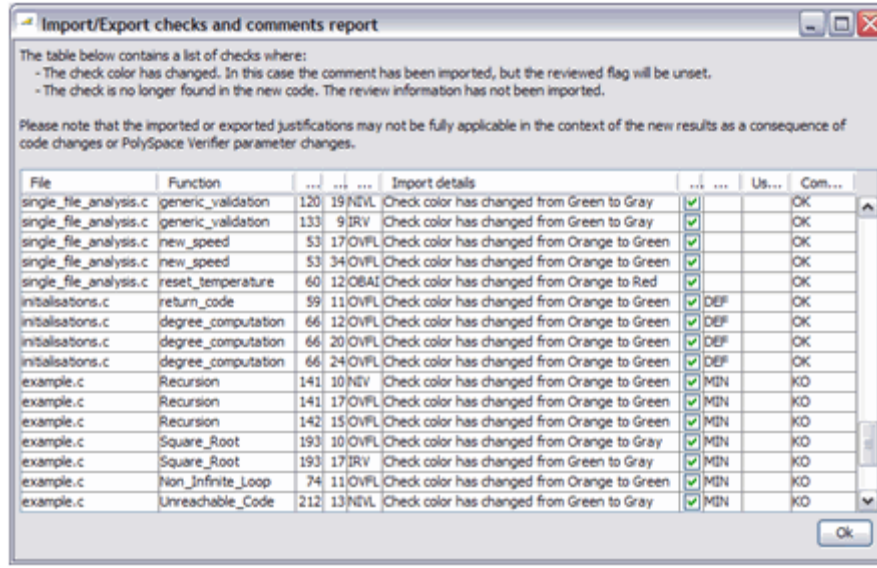
Importing Review Comments

Compatibility Considerations: Yes

New Import/Export checks and comments report allows you to you to compare the source code and verification results from a previous verification to the current verification, and highlights differences in the results.

Importing review comments from a previous verification can be extremely useful, since it allows you to avoid reviewing checks twice, and to compare verification results over time. However, if your code has changed since the previous verification, or if you have upgraded to a new version of the software, the imported comments may not be applicable to your current results. For example, the color of a check may have changed, or the justification for an orange check may no longer be relevant to the current code.

Use the Import/Export checks and comments report to highlight these differences, and focus on unreviewed results.



File	Function	...	...	...	Import details	...	...	Us...	Com...
single_file_analysis.c	generic_validation	120	19	NIVL	Check color has changed from Green to Gray	✓			OK
single_file_analysis.c	generic_validation	133	9	IRV	Check color has changed from Green to Gray	✓			OK
single_file_analysis.c	new_speed	53	17	OVFL	Check color has changed from Orange to Green	✓			OK
single_file_analysis.c	new_speed	53	34	OVFL	Check color has changed from Orange to Green	✓			OK
single_file_analysis.c	reset_temperature	60	12	OBAI	Check color has changed from Orange to Red	✓			OK
initialisations.c	return_code	59	11	OVFL	Check color has changed from Orange to Green	✓	DEF		OK
initialisations.c	degree_computation	66	12	OVFL	Check color has changed from Orange to Green	✓	DEF		OK
initialisations.c	degree_computation	66	20	OVFL	Check color has changed from Orange to Green	✓	DEF		OK
initialisations.c	degree_computation	66	24	OVFL	Check color has changed from Orange to Green	✓	DEF		OK
example.c	Recursion	141	10	NIV	Check color has changed from Orange to Green	✓	MIN		KO
example.c	Recursion	141	17	OVFL	Check color has changed from Orange to Green	✓	MIN		KO
example.c	Recursion	142	15	OVFL	Check color has changed from Orange to Green	✓	MIN		KO
example.c	Square_Root	193	10	OVFL	Check color has changed from Orange to Gray	✓	MIN		KO
example.c	Square_Root	193	17	IRV	Check color has changed from Green to Gray	✓	MIN		KO
example.c	Non_Infinite_Loop	74	11	OVFL	Check color has changed from Orange to Green	✓	MIN		KO
example.c	Unreachable_Code	212	13	NIVL	Check color has changed from Green to Gray	✓	MIN		KO

For more information, see Importing and Exporting Review Comments in the Polyspace Products for C/C++ User's Guide.

Compatibility Considerations

In previous releases, when you specified the option `-keep-all-files`, it was possible to add comments to the results for a specific verification level (for example, `pass2`) , and then import them into another set of results (for example `pass4`) in the same results folder.

This is no longer possible in R2010a.

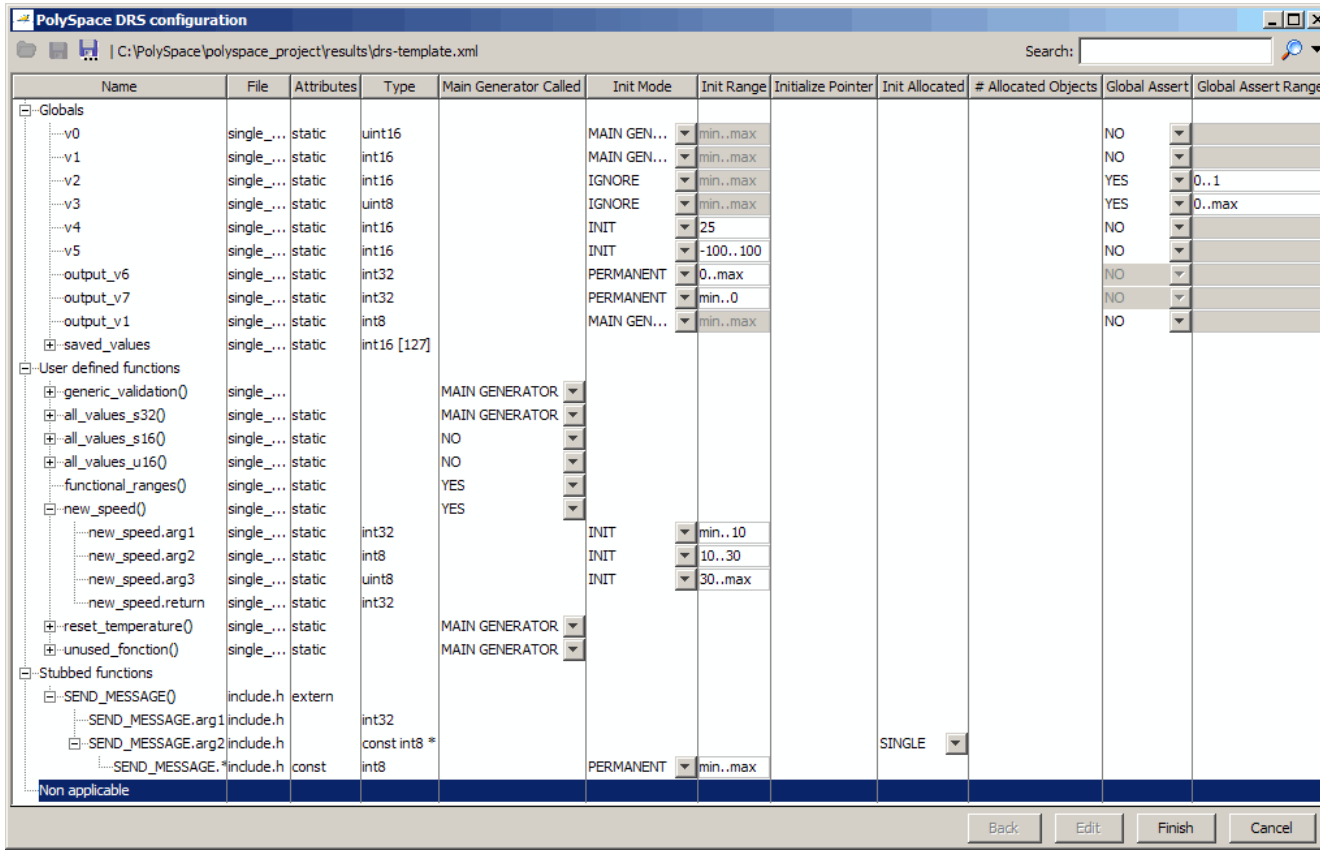
Data Range Specifications (DRS) Enhancements

Compatibility Considerations: Yes

Enhanced Data Range Specifications, including new format and workflow.

The Polyspace Data Range Specifications (DRS) feature now allows you to set constraints on data ranges using a new graphical user interface. When you enable the DRS feature, Polyspace software analyzes the files in your project, and generate a DRS template containing all the global variables, user defined functions, and stub functions for which you can specify data ranges.

To specify data ranges, you then edit this template using the Polyspace DRS configuration interface.



In addition, the DRS feature now allows you to specify constraints for additional types of data, including:

- Input parameters for user-defined functions called by the main generator
- Static variables
- Pointers (C only)

For more information, see Specifying Data Ranges for Variables and Functions (Contextual Verification) in the *PolySpace Products for C User's Guide*.

Compatibility Considerations

Symbols ranged by DRS (`init`, `permanent` or `globalassert` mode) are no longer ignored by the main-generator. This can lead to differences in values and colors, for example full range instead of 0, or orange instead of green.

Pointer Information in the Viewer

Enhanced ToolTips in the Viewer now display pointer information, in addition to data ranges.

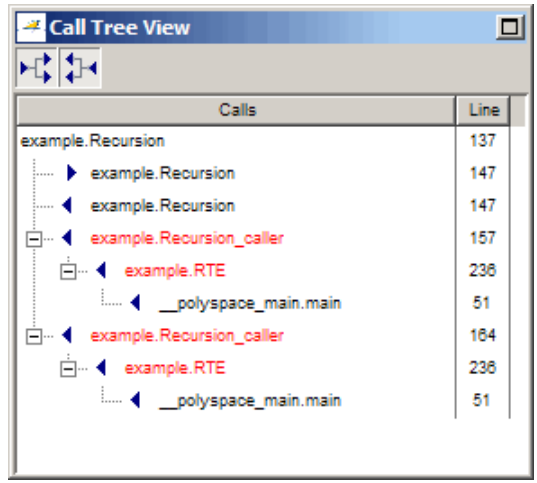
The software now provides, through tooltip messages, useful information about pointers to variables or functions. You see this information in the source code view when you place your cursor over a pointer, dereference character, function call, or function declaration. In addition, if you click a pointer check, dereference character, function call, or function declaration, the software displays pointer information in the selected check view.

For more information, see Using Pointer Information in Results Manager Perspective in the *PolySpace Products for C User's Guide*.

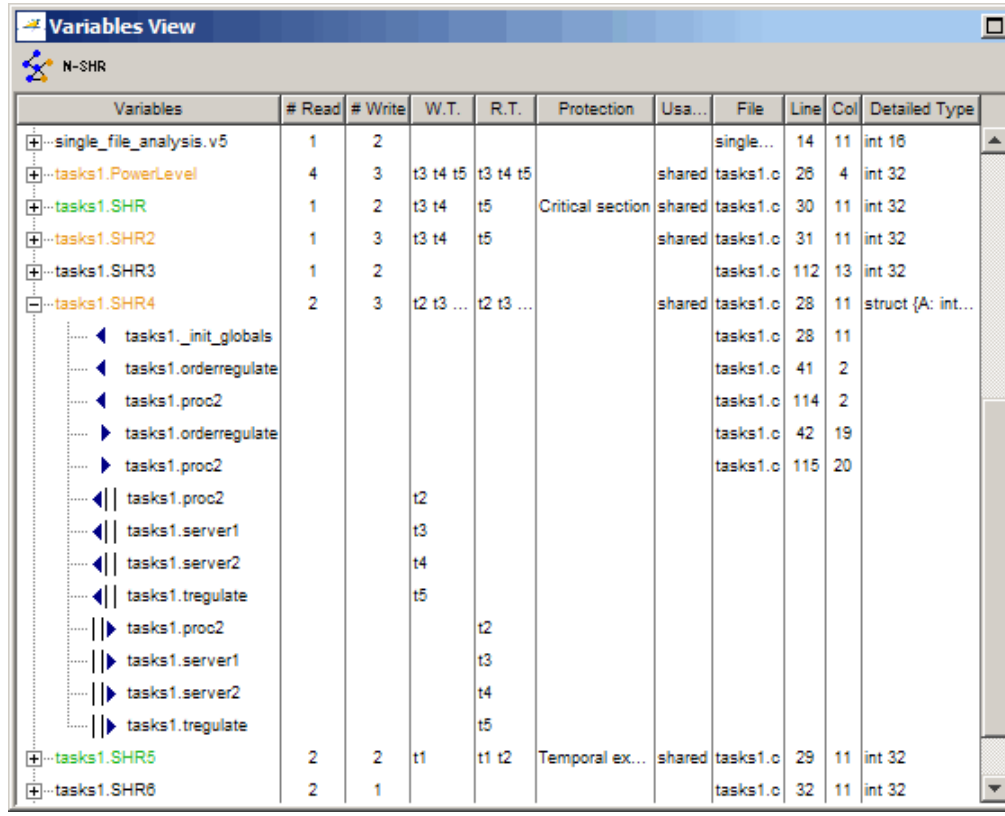
Enhanced Call Tree View and Variables View (Data Dictionary)

Enhanced user interface of the Call Tree View and Variables View improves navigation and usability.

In the Call Tree View, you can now double click any function call to go directly to the function definition.



In the Variables View, you can now right-click a variable to show legend information, and can open the concurrent access graph for a variable directly from the Variables View.

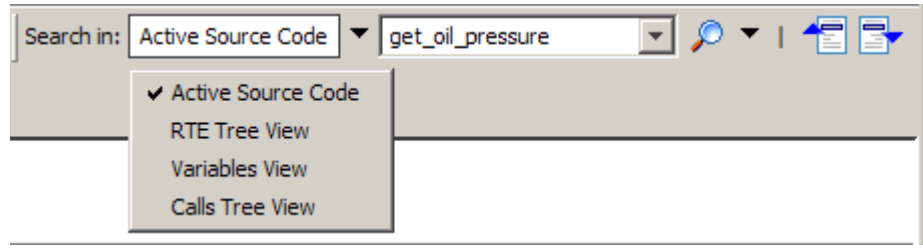


For more information, see Exploring the Results Manager Perspective in the *PolySpace Products for C User's Guide*.

Enhanced Search Function in Viewer

Enhanced Search feature in the Viewer improves navigation in your results.

The Viewer toolbar now contains a Search interface. This allows you to quickly enter search terms, specify search options, and set the scope for your search.



For more information, see Exploring the Results Manager Perspective in the *PolySpace Products for C User's Guide*.

Filtering Orange Checks in Viewer (C only)

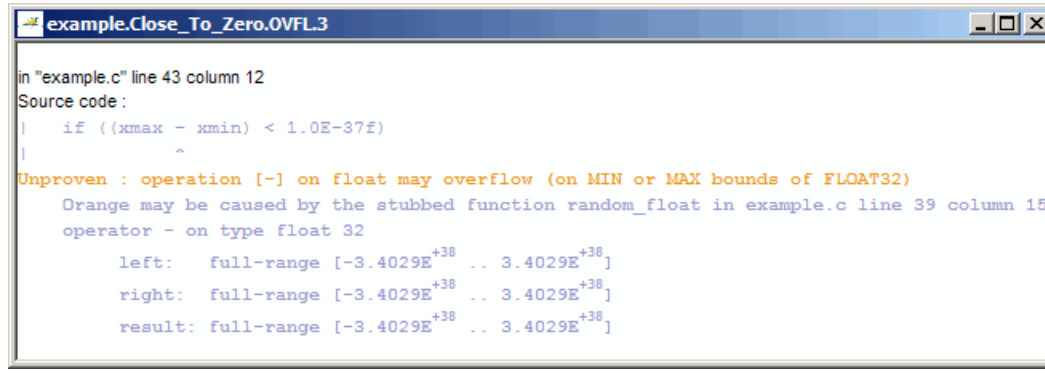
Polyspace verification now identifies orange checks caused by input data. The software provides additional information on these orange checks, and allows you to hide them in the Viewer.

Note Although this type of orange check could reveal a bug, they usually do not.

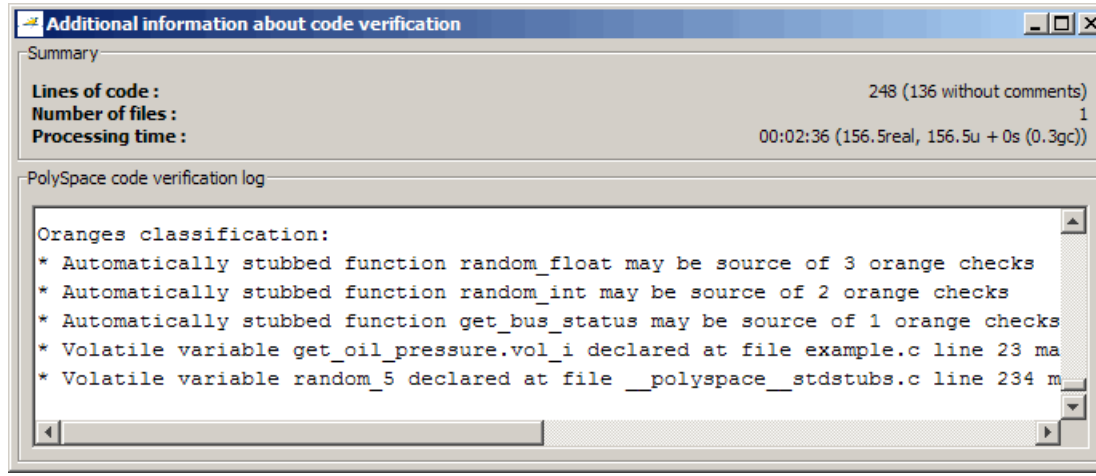
Verification can identify orange checks caused by:

- Stubs
- Main-generator calls
- Volatile variables
- Extern variables
- Absolute address

When the software identifies this type of orange check, the Viewer provides information on its cause.



The Polyspace code verification log file also lists possible sources of imprecision for orange checks.



In addition, you can now hide these types of orange checks in the Viewer. When using Expert mode, click the filter button to hide oranges impacted by input data.

For more information, see Working with Orange Checks Caused by Input Data in the *PolySpace Products for C User's Guide*.

Methodological Assistant Enhancements

Compatibility Considerations: Yes

Enhanced Methodological Assistant in the Viewer.

The Methodological Assistant now allows you to define either a minimum percentage of orange checks to review, or a specific number of orange checks to review. This makes it easier to set specific quality criteria for your code at each level of review.

In addition, the Methodological Assistant now presents checks in a more logical order. Checks that are most likely to reveal bugs appear first, while non-useful checks no longer appear.

The new order of checks is:

1. All red checks (an error always occurs)
2. Orange checks known to produce errors in some situations (dark orange). For example, red for one call to a procedure and green for another.
3. Some gray checks (UNR checks)
4. Other orange checks (depending on the methodology and criterion level)

Most gray checks no longer appear in the Methodological Assistant, since reviewing many gray checks that occur after a red check is not useful. Only UNR checks that are not nested within dead code blocks appear in assistant mode.

Compatibility Considerations

The number of checks presented for review in Assistant mode is different than in previous releases, since most gray checks no longer appear. In addition, the order in which you review checks is different.

Class Analyzer Enhancements for C++

Compatibility Considerations: Yes

Enhanced class analyzer can analyze a file with more than one class.

Unit-by-unit verifications can now verify files containing more than one class. Every class and function out of class contained in such files is now verified.

For more information, see *PolySpace Class Analyzer* in the *PolySpace Client/Server for C++ User Guide*.

Compatibility Considerations

In `-unit-by-unit` mode, files that previously were not verified because they contained more than one class are now verified.

Change to Time Format in Log File

Compatibility Considerations: Yes

The time format reported in the log file has been updated to provide more information.

Example of new line (R2010a and later):

```
User time for polyspace-c:  00:02:24 (144.6real, 144.6u + 0s
(0.3gc))
```

Example of old line (R2009b and earlier):

```
User time for rte-kernel:  4684.4real, 4319.2u + 324.6s (0.3gc)
```

Compatibility Considerations

The new time format can impact some scripts that summarize information from the log file.

Merging of OVFL and UNFL Checks

Compatibility Considerations: Yes

Overflow (OVFL) and underflow (UNFL) checks have been merged into a single OVFL check. This reduces the number of orange checks you need to review, while continuing to provide the same information.

For red and orange checks, the check message provides the bounds that cause the overflow.

Compatibility Considerations

The Selectivity rate of your results may change when compared to previous versions of the software. Underflows and overflows are now identified as a single check, so the Selectivity will decrease if the checks were green (2 green checks become 1 green), but will increase if the checks were both orange (2 orange checks become 1 orange).

Improved UNR Checks

Compatibility Considerations: Yes

Enhanced unreachable code (UNR) checks now provide additional information to help you understand the results. UNR checks now include information on:

- Localization of condition
- Type of condition
- End of block localization

For example:

```
// UNR (unreachable code) => UNR (unreachable code)   \
(end of block at line YYY)

// UNR (unreachable code) => UNR (unreachable code)   \
(condition at line XXX, column AAA) ?
```

In addition, verification now reports new `UNR` checks on:

- unreachable statements after `return`, `break`, `goto`, and `continue` statements.
- `if` statements when the `if` condition is always true and if there is no `else` statement.

For more information on these new checks, see "Changes to Verification Results" on page 221.

Compatibility Considerations

The number of checks in your verification results may change due to the new `UNR` checks.

Changes to Verification Results
Compatibility Considerations: Yes

Compatibility Considerations

Verification results may change when compared to previous versions of the software. Some checks may change color, and the Selectivity rate of your results may change.

Refer to the following sections for information on the specific changes.

Merging of OVFL and UNFL Checks

Overflow (OVFL) and underflow (UNFL) checks have been merged into a single OVFL check. This reduces the number of orange checks you need to review, while continuing to provide the same information.

For red and orange checks, the check message provides the bounds that cause the overflow.

The Selectivity rate of your results may change when compared to previous versions of the software.

New Gray (`UNR`) Checks on `return`, `break`, `goto`, and `continue` Statements

Verification now reports gray `UNR` checks on unreachable statements after `return`, `break`, `goto`, and `continue` statements.

For example:

```
67 switch (counter) {
68  case 0:
69   counter = 0;
70   break;
71 case 1:
72  counter = 2;
73  break;
74   counter = 2; /* unreachable code ! */
75   break;
```

The number of checks in your verification results may increase due to these new UNR checks.

New Gray (`UNR`) Check on If Statement Without Else

Verification now reports a gray `UNR` check on an `if` statement when the `if` condition is always true and if there is no `else` statement.

This allows you to find `if` branches that are always reachable, even when there is no `else`.

For example:

```
if (true())  // UNR if-condition always evaluates to true
 {
  // ...
 }
```

The number of checks in your verification results may increase due to new UNR checks.

Nested Gray (UNR) Checks No Longer Appear in Reports

Nested UNR checks in unreachable blocks no longer appear in the Methodological Assistant, or in generated reports.

The number of checks in generated reports may decrease due to elimination of these checks..

Dead Code on Else Branch

Verification now reports gray UNR checks on empty branches.

For example:

```
void fct (void)
{
 int a = 1;
 if (a){
  a++;
 }
 else // ==> Now gray UNR
 {
 // dummy
 }
}
```

The number of checks in your verification results may increase due to the new UNR check on empty branches.

Data Ranges for Fields of Structures (C)

Symbols ranged by DRS (`init`, `permanent` or `globalassert` mode) are now considered by the main-generator.

In previous releases, if DRS provided ranges for some fields of a structure, the other fields (not ranged by DRS) were not initialized by the main-generator, and therefore had an initial value of 0.

For example:

```
// DRS: s.x 0 10 init

struct { int x; int y; } s;
int foo(void)
{
  return s.y; // y value: 0, full-range expected
}
```

Symbols ranged by DRS are no longer ignored by the main-generator. This can lead to differences in values and colors, for example full range instead of 0, or orange instead of green.

Functions Called Before Main in Unit-by-unit Verification (C++)

The behavior of the option `-function-called-before-main` has changed for unit-by-unit verifications of C++ code.

When you set the option `-function-called-before-main` in unit-by-unit mode:

- If the `init` function is an out of class function, it is called at the beginning of the generated main (before calls to constructors).
- If the `init` function is a method, it is called after all constructor calls of the corresponding class.

In previous releases, the `init` function was always called after constructor calls for each class.

Verification results may change when compared to previous versions of the software, due to changes in the call sequence.

Main Generator Initialization of Function Pointers

The main-generator now initializes function pointers with default-mode stubs instead of pure stubs.

In previous releases, the main-generator initialized function pointers with pointers to pure functions.

This change may lead to differences in the color of checks in your results. For example:

```
int x;
  s->fptr(&x);
  read(x); // LNIV red with 9b -> orange with 10a
```

OVFL Check on Array Index Removed

In previous releases, verification reported an overflow (OVFL) check on pointer/array dereference. However, this overflow never occurred if there was an OBAI problem first. Therefore, the check was not useful.

In R2010a, the OVFL check no longer appears on array index, the check has been merged into the OBAI check.

For example, in the following code there is no OVFL check on the array index.

```
int main(void)
{
   volatile int i,x;
   int tab[10];

   x = tab[i];

}
```

The Selectivity of your results may change when compared to previous versions of the software. The OVFL check on array access has been merged into the OBAI check, so there are fewer checks reported. Selectivity will increase if the overflow check was orange, but will decrease if the OVFL check was green.

IDP Check on Local Member Access Removed (C++)

Verification no longer reports an IDP check on local member access.

In previous releases, verification reported an IDP check. This IDP appeared on the "." when accessing the field of an object returned by copy construction.

For example:

```
struct C {
  C(const C&c1) { k =c1.k; }
  C() { k = 0 ;}
  int k ;
} ;

C g() {
  C ret ;
  ret.k = 2 ; // IDP on "." here
  return ret;
}

int main() {
  C c = g() ;
}
```

However, this check was caused by an internal pointer and was not useful.

The Selectivity of your results may change when compared to previous versions of the software.

OBAI Check on Dynamic Initialization of Array Removed (C++)

Verification no longer reports an OBAI check on dynamic initialization of array.

In previous releases, verification reported an OBAI check. The OBAI check appeared on dynamic initialization of array with an aggregate.

For example:

```
nt main(void)
```

```
{
  float tab[] = // extra green obai check
    {
      4.3,
      0.0F
    };

  return 0;
}
```

However, this check was caused by an internal translation and was not useful.

The Selectivity of your results may change when compared to previous versions of the software.

Duplicate Checks in `For/While` Loops Removed

Verification no longer reports duplicate checks in condition expression of `for` and `while` loops.

Any duplicate checks on a loop condition are now merged in a single check, except when condition expression is complex.

Due to the reduction in the number of checks, the selectivity of your results may change when compared to previous versions of the software.

`malloc(0)` Limitation Removed

Verification no longer has a limitation when `malloc(0)` returns a null pointer.

In previous releases, verification reported a green check on the following code:

```
assert(malloc(0) == NULL) ;
```

However, this construction could fail. The software now verifies this construction.

Verification results may change when compared to previous versions of the software.

Change in OOP on Deletion of Null Pointer (C++)

Verification no longer reports a red OOP check when deleting a null pointer.

In previous releases, verification reported a red OOP check on the following code:

```
struct A {
  virtual void f() { }
  ~A() { }
} ;

int main() {
  A* pa ;
  if (0) pa = (A*) 0Xfff;
  pa = 0;
  delete pa ; // red OOP
}
```

However, calling "delete" on a null pointer is allowed. The red OOP when deleting a null pointer is now gray.

Verification results may change when compared to previous versions of the software.

Change to IDP Check When Accessing a Field of an Inherited Class (C)

Verification no longer reports a red IDP check when accessing a field of an inherited class with an mcpu target.

In previous releases, verification reported a red IDP check.

For example:

```
struct Val {
 int val;
};

struct Left : virtual Val {
 int left;
```

```
 virtual int get_left() { return left; } // polymorphic:yes
};

struct Right : virtual Val {
 int right;
 virtual int get_right() { return right; }
};

struct S : Left, Right {      // multiple:yes
};

S s = S();
Left& le = s;         // intermediate:global, reference:yes
Right& re = s;         // intermediate:global, reference:yes

int main(void){
  assert(re.val == 0);    // Unexpected red IDP
}
```

However, this was not actually an error. The check is no longer red.

The color of the IDP check has changed when compared to previous versions of the software.

Changes to Coding Rules Checker Results

Compatibility Considerations: Yes

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations may change when compared to previous versions of the software.

Refer to the following sections for information on the specific changes.

MISRA-C Rule 10.1 Violations on Constant Operands

The MISRA-C checker no longer reports errors for rule 10.1, "The value of an expression of integer type shall not be implicitly converted to a different underlying type," for certain constructions. For example:

```
int i;
for (i = 0; i < 12; i++)
```

An integer constant that fits into the size of a `char` is now seen as a `signed char` whatever the sign of `char` (this depends on the selected target or is set by option).

If you use the options `-target powerpc` or `-default-sign-of-char unsigned`, the coding rules checker will report fewer violations of MISRA-C rule 10.1 on constant operands.

MISRA-C Rule 12.5 Violation Report Improved

The coding rules checker now reports a column number for violations of MISRA-C rule 12.5.

You may see more violations of rule 12.5, since two violations that occur on same line but in different columns are now identified separately.

MISRA-C Rule 7.1 Violations on File Names of Preprocessed Files

The coding rules checker no longer reports violations of MISRA-C rule 7.1 on the names of internal preprocessing files. These violations occurred in projects containing Japanese characters.

You may see fewer violations of rule 7.1 in MISRA reports.

MISRA-C Rule 5.4 Violations on Anonymous Structures and Unions

The coding rules checker no longer reports violations of MISRA-C rule 5.4 on anonymous struct/union fields.

You may see fewer violations of rule 5.4 in MISRA reports.

JSF Rule AV-151 Violations on Evaluation of Constant

The coding rules checker no longer reports violations of JSF rule AV-151 on internal evaluation of a constant value, for example when there is an expression in an enum list.

You may see fewer violations of rule AV-151 in JSF reports.

Enumerated Types Support

The option `-enum-type-definition` allows verification to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition.

When using this option, each enum type is represented by the smallest integral type that can hold all its enumeration values.

Possible values are:

- `defined-by-standard`.
- `auto-signed-first`.
- `auto-unsigned-first`

For more information, see Enum type definition (-enum-type-definition) in the *PolySpace Products for C Reference*.

New Target Processor Support

Added support for the `c18` 24-bit target processor (C only).

For more information, see Predefined Target Processor Specifications in the *PolySpace Products for C Reference*.

Operating System Support

Added support for the following Linux distributions:

- OpenSuSE 11.1
- Debian 5.x
- Ubuntu 8.04, 8.10, 9.04, and 9.10

For more information, see the Polyspace Installation Guide.

Polyspace Server for C/C++ Product

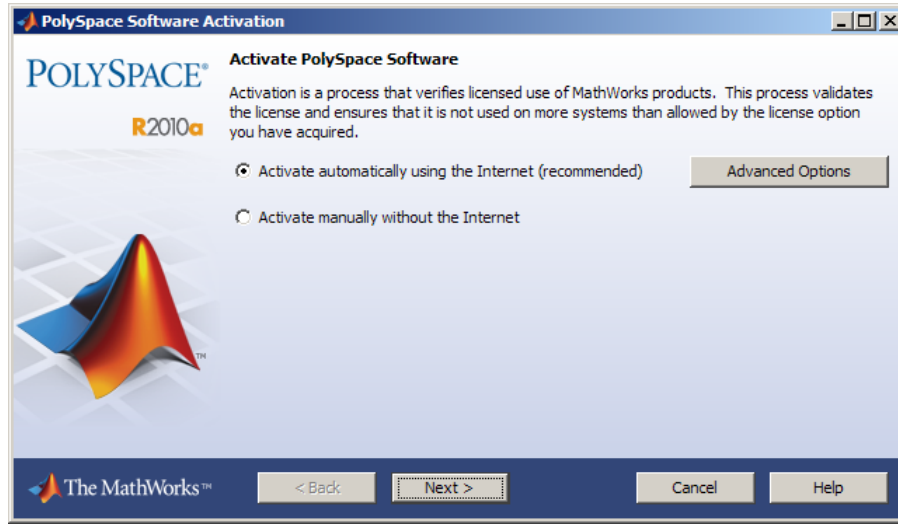
License Activation

Polyspace products now support the MathWorks software activation mechanism.

Activation is a process that verifies licensed use of MathWorks products. The process validates your product licenses. You must complete the activation process before you can use Polyspace software.

Note If you are using Designated Computer (Individual) licenses, you must activate the license for each Polyspace system individually. However, if you are using Concurrent licenses for multiple Polyspace systems, you do not need to activate each Polyspace system. You activate the license once (for the FLEXnet license server), then provide license files for each Polyspace system.

The easiest way to activate the software is to log in to your MathWorks Account during installation. At the end of the installation process, the Polyspace Software Activation dialog box opens.



Follow the prompts in the Polyspace Software Activation dialog box to complete the activation process.

If you do not have a MathWorks account, you can create one during the activation process. To create an account, you must have an Activation Key, which identifies the license you want to install and activate.

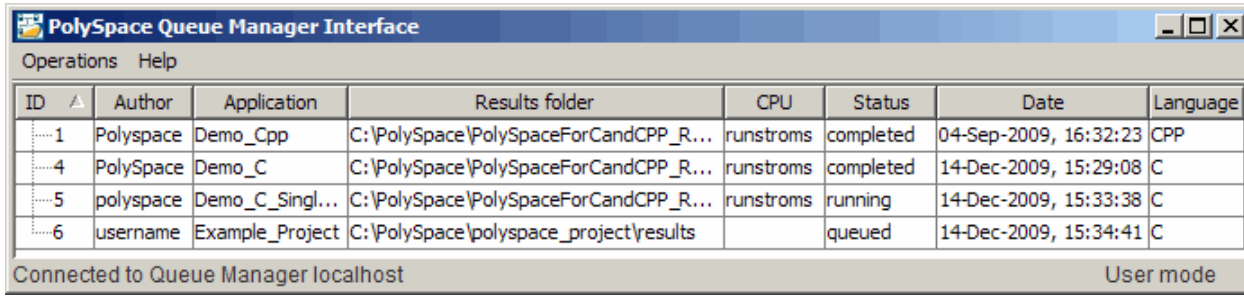
If your Polyspace system is not connected to the internet, you can access the MathWorks License Center on a computer with internet access, activate your license, and download a license file for transfer to your Polyspace system. If you do not have access to a computer with an Internet connection, contact Customer Support.

For more information on how to activate your software, see Activating Polyspace Softwarein the Polyspace Installation Guide.

For more information on software activation, including frequently asked questions, refer to the MathWorks Web site: www.mathworks.com/support/activation/polyspace.html

Queue Manager Interface

The Polyspace Queue Manager Interface (Spooler) is now available on Linux machines, providing a graphical interface for managing verification jobs on the Polyspace server.



For more information, see Managing Verification Jobs Using the Polyspace Queue Managerin the Polyspace Products for C/C++ User's Guide or Polyspace Products for C++ User's Guide.

Operating System Support

Added support for the following Linux distributions:

- OpenSuSE 11.1
- Debian 5.x
- Ubuntu 8.04, 8.10, 9.04, and 9.10

For more information, see the Polyspace Installation Guide.

R2009b

Version: 7.1
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Report Generator

New Report Generator that presents Polyspace results in PDF, HTML, and other output formats.

The Polyspace Report Generator allows you to generate reports about your verification results, using the following predefined report templates:

- **Coding Rules Report** – Provides information about compliance with MISRA-C Coding Rules, as well as Polyspace configuration settings for the verification.
- **Developer Report** – Provides information useful to developers, including summary results, detailed lists of red, orange, and gray checks, and Polyspace configuration settings for the verification.
- **Developer with Green Checks Report** – Provides the same content as the Developer Report, but also includes a detailed list of green checks.
- **Quality Report** – Provides information useful to quality engineers, including summary results, statistics about the code, graphs showing distributions of checks per file, and Polyspace configuration settings for the verification.

The Polyspace Report Generator allows you to generate verification reports in the following formats:

- HTML
- PDF
- RTF
- Microsoft Word
- XML

Note Microsoft Word format is not available on UNIX platforms. RTF format is used instead.

For more information, see Generating Reports of Verification Results in the Polyspace Products for C/C++ User's Guide.

Viewer Enhancements

Enhanced Viewer displays results with tooltips containing the values of variables, operands, function parameters, and return values.

You can see range information associated with variables and operators within the source code view.

Note The displayed range information represents a superset of dynamic values, which the software computes using static methods.

If a line of code is all the same color, selecting the line opens an Expanded Source Code window. Place your cursor over the required operator or variable in this window to view range information.

If a line of code contains different colored checks, selecting a check displays the error or warning message along with range information in the selected check view.

For more information, see Using Range Information in the Results Manager Perspective in the Polyspace Products for C/C++ User's Guide.

Global Data Graphs

New Graphs (similar to concurrent access graphs) available for all global data.

You can display the access sequence for any variable that is read or written in the code. The access graph displays the read and write access for the variable.

For more information, see Displaying the Access Graph for Variables in the Polyspace Products for C/C++ User's Guide.

Unit-by-unit Verification

New option to create a separate verification job for each source file in the project.

When you run a unit-by-unit verification, each source file is compiled, sent to the Polyspace Server, and verified individually.

The queue manager displays a job for the full verification group, as well as jobs for each unit (using a tree structure).

When verification is complete, you can download and view results for the entire project, or for individual units. When downloading a verification group, all the unit results are downloaded and a summary of the download status for each unit is displayed.

Note Unit by unit verification is available only for server verifications.

For more information, see Running Verification Unit-by-Unit in the *PolySpace Products for C Reference*.

Changes to Coding Rules Checker Results

Compatibility Considerations: Yes

Compatibility Considerations

Due to changes in the coding rules checker, the number of coding rule violations may change when compared to previous versions of the software.

Refer to the following sections for information on the specific changes.

MISRA-C Rule 5.1 Analysis Improved

The coding rules checker now applies MISRA-C rule 5.1 to all identifiers external and internal.

MISRA-C Rule 5.2 Analysis Improved

The coding rules checker now detects violations of MISRA-C Rule 5.2 when the declaration in the outer scope occurs after the declaration in the inner scope.

MISRA-C Rule 5.7 Analysis Improved

The coding rules checker now detects violations of MISRA-C Rule 5.7 in local reused identifiers.

MISRA-C Rule 8.10 Analysis Improved

Only the last declaration takes precedence for `static` or `extern`. The coding rules checker no longer reports violations of MISRA-C Rule 8.10 if the last declaration is static.

MISRA-C Rule 10.1 Analysis Relaxed

The coding rules checker has relaxed enforcement of MISRA-C Rule 10.1 for `x` in `[x]` for any type of expression `x`.

MISRA-C Rule 10.5 Analysis Improved

The coding rules checker now detects violations of MISRA-C Rule 10.5 in expressions with constants.

For example:

```
c = (uint8_t)(ui8 << ( 1U << 2U ) );
```

MISRA-C Rule 12.7 Analysis Improved

The coding rules checker now detects violations of MISRA-C Rule 12.7 in expressions with constants.

For example:

```
~(i=1);
```

MISRA-C Rule 15.0 Analysis Improved

The coding rules checker now detects violations of MISRA-C Rule 15.0 in all statements between switch and first case clause (label, harmless statement).

In addition the coding rules checker now detects jumps and label statements.

MISRA-C Rule 16.4 Analysis Improved

The coding rules checker now keeps the names of the parameters of the first declaration, and reports violations of MISRA-C Rule 16.4 for each occurrence.

Operating System Support

Added support for Windows Server® 2008.

For more information, see the Polyspace Installation Guide.

Polyspace Server for C/C++ Product

Operating System Support

Added support for Windows Server 2008.

For more information, see the Polyspace Installation Guide.

R2009a

Version: 7.0
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

JSF++ Support

Enhanced JSF C++ checker supports all checkable Joint Strike Fighter Air Vehicle C++ coding standards (JSF++:2005).

Polyspace software can now check all possible C++ programming rules defined by Lockheed Martin® for the JSF program. These coding standards are designed to improve the robustness of C++ code, and improve maintainability.

For more information, see *JSF C++ Checker*, in the *PolySpace Client/Server for C++ User Guide*.

Back to Source Link

New “back-to-source” link in the Polyspace launcher associates compile errors, MISRA-C violations, and JSF++ violations reported in the logs directly to the source file.

For more information, see Viewing Coding Rules Checker Results in the *PolySpace Products for C User's Guide* or *Examining the JSF Log*, in the *PolySpace Client/Server for C++ User Guide*.

Eclipse Integration

New Polyspace integration with the Eclipse IDE, Version 3.3.

The Polyspace Client for C/C++ product can be integrated with the Eclipse Integrated Development Environment through the Polyspace C/C++ plug-in for Eclipse IDE.

This plug-in provides Polyspace source code verification and bug detection functionality for source code developed within Eclipse IDE. Features include the following:

- A contextual menu that allows you to launch a verification of one or more files.
- Views in the Eclipse editor that allow you to set verification parameters and monitor verification progress.

For more information, see Using Polyspace Software in the Eclipse IDE in the *PolySpace Products for C User's Guide*.

Performance Improvements for Multi-Core Systems

Enhanced performance on multi-core architecture platforms, improving the speed of Polyspace code verification.

The time required to perform an average code verification has been reduced. On multi-core systems, you can now select the number of processes that can run simultaneously, further improving performance.

For more information, see Number of processes for multiple CPU core systems (-max-processes) in the *PolySpace Products for C Reference*.

Architecture Improvements

Compatibility Considerations: Yes

Several changes have been made to the Polyspace architecture to improve overall performance, as well as the precision of verification results.

During each verification phase (pass), the software now only analyzes those procedures that need to be analyzed. This means that starting with PASS1, if the verification cannot be more precise than that already completed in a previous pass, the procedure is not analyzed again. This improves the overall performance of the verification. It also means that some passes will finish more quickly than others, and some passes could be completely empty. This is normal behavior.

In addition, these architecture improvements result in the following changes:

- The `quick` precision option is now obsolete, and has been removed. `quick` mode has been replaced with verification PASS0. PASS0 takes somewhat longer to run, but the results are more complete. The limitations of `quick` mode, (no NTL or NTC checks, no float checks, no variable dictionary) no longer apply. Unlike `quick` mode, PASS0 also provides full navigation in the Viewer.
- The `voa` option is now obsolete, and has been removed. Value On Assignment checks are now provided by default. In general, this means that Polyspace results now contain many more VOA checks. For C applications, all possible VOA are given.
- The UOVFL (Float Underflows and Overflows) check no longer exists. Float underflows and overflows are now reported as two separate checks. This is similar to the way integers are handled.

  Note Since the single UOVFL check has been replaced by two checks, the total number of checks reported by Polyspace on a given file may be different in this release than with previous versions of the software.

- Messages have been improved for float arithmetic checks, making them similar to the messages for integers. For example, NIV checks on float variables now contain the type size (32 or 64).

- For IPT (Inspection Point) checks, there is now one check for each variable. Previously there was a single IPT check (on the keyword) for multiple variables.
- The log file has several additions, including the names of each PASS, the verification phases, and additional messages.

Compatibility Considerations

The verification results provided by Polyspace software may be different in R2009a than with previous releases of the software. Verification results are more precise, and the total number of checks reported on a given source file may be different. In general, the software now reports more checks, due to increased VOA checks, changes to the IPT check, and the single float UOVFL check being replaced by two checks (UNFL and OVFL).

In addition, due to the float UOVFL check being split into two checks, the selectivity (number of proven checks red+green+gray / number of total checks) of a verification may change significantly for applications using many float variables. For example, an application that had 10 orange UOVFL checks with a previous release, could now have up to 20 orange UNFL and OVFL checks on the same float variables. Although this appears to be a decrease in precision, the verification itself is not less precise.

Mathematical Functions Included in Stubs

Compatibility Considerations: Yes

Mathematical functions are now included in the standard stubs. This means:

- An IRV (Initialized Return Value) check appears on the math function call.
- The POW check no longer appears in the Viewer.
- Math functions appear in the call graph.
- The modeling of mathematical functions is visible through the stub body, instead of being handled internally.
- By default, math functions are launched with the option `-context-sensitivity` , allowing them to distinguish their calling sites.

In addition, you can provide your own math functions instead of using the standard stub provided by Polyspace software. This allows the software to verify the body of the math function, instead of using a stub for the math function.

For example, in C90, the mathematical function `fabs()` has the prototype:

```
double fabs(double) ;
```

However, on a 16-bit target, the function may have the prototype:

```
float fabs(float);
```

In this case, you would want to verify your own `fabs()` function.

To provide your own math function:

1. Create source code for the function. For example:

```
float  fabs (float var)
{
  if (var >= 0.0f)
    return var;
  return -var;
}
```

2 Provide the function to your verification using the `D` compiler flag. For example:

```
polyspace-c -D  __polyspace_no_fabs
```

Note There is a compiler flag for each standard ANSI C90 mathematical function. A complete list of flags is located in the file: `%POLYSPACE_C%\Verifier\cinclude\__polyspace__stdstubs.c`.

Compatibility Considerations

Since the POW check no longer appears in the Viewer, verification results may be different in R2009a than with previous releases of the software.

Character Encoding Options

New character encoding option allows you to view source files created on an operating system that uses different character encoding than your current system.

You specify the character encoding used by the operating system on which the source file was created using the **Character encoding** tab in the Preferences dialog box of the Polyspace Viewer.

For more information, see Setting Character Encoding Preferences in the *PolySpace Products for C User's Guide*.

Automatic Orange Tester

Compatibility Considerations: Yes

The Automatic Orange Tester (for C), dynamically stresses unproven code (orange checks) to help you identify run-time errors.

For more information, see Automatically Testing Orange Code in the Polyspace Products for C/C++ User's Guide.

Compatibility Considerations

If you open verification results created with an older version of the product in the Automatic Orange Tester, you may get a compilation error. The version of the product used to create the instrumented source code must be the same as the one used for analysis in the Automatic Orange Tester.

To avoid this problem, re-launch the code verification with the current version of the product.

Operating System Support

Added support for Windows Server 2003, Windows Vista™, and Red Hat Enterprise Linux Workstation v.5.

For more information, see the Polyspace Installation Guide.

Polyspace Server for C/C++ Product

Performance Improvements for Multi-Core Systems

Enhanced performance on multi-core architecture platforms, improving the speed of Polyspace code verification.

The time required to perform an average code verification has been reduced. On multi-core systems, you can now select the number of processes that can run simultaneously, further improving performance.

For more information, see Number of processes for multiple CPU core systems (-max-processes) in the Polyspace Products for C/C++ Referenceor Polyspace Products for C++ Reference.

Architecture Improvements

Compatibility Considerations: Yes

Several changes have been made to the Polyspace architecture to improve overall performance, as well as the precision of verification results.

During each verification phase (pass), the software now only analyzes those procedures that need to be analyzed. This means that starting with PASS1, if the verification cannot be more precise than that already completed in a previous pass, the procedure is not analyzed again. This improves the overall performance of the verification. It also means that some passes will finish more quickly than others, and some passes could be completely empty. This is normal behavior.

In addition, these architecture improvements result in the following changes:

- The `quick` precision option is now obsolete, and has been removed. `quick` mode has been replaced with verification PASS0. PASS0 takes somewhat longer to run, but the results are more complete. The limitations of `quick` mode, (no NTL or NTC checks, no float checks, no variable dictionary) no longer apply. Unlike `quick` mode, PASS0 also provides full navigation in the Viewer.
- The `voa` option is now obsolete, and has been removed. Value On Assignment checks are now provided by default. In C, all possible VOA are given.
- The UOVFL (Float Underflows and Overflows) check no longer exists. Float underflows and overflows are now reported as two separate checks. This is similar to the way integers are handled.

  Note Since the single UOVFL check has been replaced by two checks, the total number of checks reported by Polyspace on a given file may be different in this release than with previous versions of the software.

- Messages have been improved for float arithmetic checks, making them similar to the messages for integers. For example, NIV checks on float variables now contain the type size (32 or 64).

- For IPT (Inspection Point) checks, there is now one check for each variable. Previously there was a single IPT check (on the keyword) for multiple variables.
- The log file has several additions, including the names of each PASS, the verification phases, and additional messages.

Compatibility Considerations

The verification results provided by Polyspace software may be different in R2009a than with previous releases of the software. Verification results are more precise, and the total number of checks reported on a given source file may be different. In general, the software now reports more checks, due to increased VOA checks, changes to the IPT check, and the single float UOVFL check being replaced by two checks (UNFL and OVFL).

In addition, due to the float UOVFL check being split into two checks, the selectivity (number of proven checks red+green+gray / number of total checks) of a verification may change significantly for applications using many float variables. For example, an application that had 10 orange UOVFL checks with a previous release, could now have up to 20 orange UNFL and OVFL checks on the same float variables. Although this appears to be a decrease in precision, the verification itself is not less precise.

Operating System Support

Added support for Windows Server 2003, Windows Vista, and Red Hat Enterprise Linux Workstation v.5.

For more information, see the Polyspace Installation Guide.

R2008b

Version: 6.0
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Automatic Orange Tester

Automatic Orange Tester (for C), dynamically stresses unproven code (orange checks) to identify run-time errors, and provides information to help you identify the cause of these errors.

The Automatic Orange Tester complements the results review in the Viewer module of Polyspace Client for C/C++ by automatically creating test cases for all input variables in orange code, and then dynamically testing the code to find actual runtime errors. The Automatic Orange Tester also provides detailed information on why each test-case failed. You can use this information to quickly identify the cause of the error, and determine if there is an actual bug in the code.

For more information, see Automatically Testing Orange Code in the *PolySpace Client/Server for C User's Guide*.

JSF++ Support

Support for a subset of the Joint Strike Fighter Air Vehicle C++ coding standards (JSF++:2005).

Polyspace software can now check 120 of the C++ programming rules defined by Lockheed Martin for the JSF program. These coding standards are designed to improve the robustness of C++ code, and improve maintainability.

For more information, see *JSF C++ Checker*, in the *PolySpace Client/Server for C++ User Guide*.

Operating System Support

Added support for 64–bit Linux.

For more information, see the Polyspace Installation Guide.

Polyspace Server for C/C++ Product

Operating System Support

Added support for 64–bit Linux.

For more information, see the Polyspace Installation Guide.

R2008a

Version: 5.1
New Features: Yes
Bug Fixes: Yes

Polyspace Client for C/C++ Product

Removed Cygwin Software Dependency for Windows Platforms

Compatibility Considerations: Yes

Previous versions of Polyspace products used Cygwin™ emulation to run UNIX® commands on Windows systems.

In version 5.1, the Cygwin software dependency has been removed. Removing Cygwin simplifies the Polyspace product installation process while improving the performance and robustness of the Polyspace Verification process.

Compatibility Considerations

Due to the Cygwin changes, Polyspace Client for C/C++ Version 5.1 is not compatible with previous versions of Polyspace products on Windows platforms. To avoid compatibility problems on Windows platforms, you must upgrade all your Polyspace client and server products at the same time.

If your Polyspace server is running on a Windows platform, the binary files used for batch commands in previous releases will not work without Cygwin software installed. In version 5.1, the software provides new .exe files for these batch commands. However, these files are now located in a different location.

Commands	Previous Location	New Location
Standard	*PolyspaceInstallDir*\verifier\bin\	*PolyspaceInstallDir*\verifier\wbin\
Remote Launcher	*Polyspace_Common*\RemoteLauncher\bin\	*Polyspace_Common*\RemoteLauncher\wbin\
Viewer	*Polyspace_Common*\Viewer\bin\	*Polyspace_Common*\Viewer\wbin\

If you wrote scripts using batch commands in previous releases, you must modify the scripts to use the new commands.

In addition, if you used Cygwin shell scripts for postprocessing or target compilation, those scripts will no longer run on version 5.1. To support scripting, the Polyspace software now includes Perl. You can access Perl in:

PolyspaceInstallDir\verifier\tools\perl\win32\bin\perl.exe

Enhanced Installer

Version 5.1 includes an enhanced and simplified installer for all Polyspace products. The installation process is now faster and easier to complete than in previous releases.

For more information, see the Polyspace Installation Guide.

Viewer Improvements

Enhanced exploring capability in the viewer to provide more precise locations for C++ checks.

The source code view of the Polyspace viewer now displays the location of C++ checks more accurately.

One-Click Enhancements

Enhanced Polyspace-In-One-Click options, to allow switching between multiple projects using a browse history.

For more information, see Day to Day Use in the *PolySpace Client/Server for C User Guide*.

Generic Target Option for C++

New Generic Target option for C++, to allow custom target processors. The Generic Target option for C++ is similar to the previous Generic Target for C.

For more information, see *Defining Generic Targets* in the *PolySpace Client/Server for C++ User Guide*.

Class Analyzer Enhancements for C++

Enhanced class analyzer now calls all private constructors and destructors.

Previously, the sources analyzed were generally non-inherited public or protected methods of the class. In version 5.1, the functions that are analyzed include all non-inherited constructors and destructors, and all non-inherited public or protected methods of the class.

For more information, see *PolySpace Class Analyzer* in the *PolySpace Client/Server for C++ User Guide*.

GNU Compiler Support for C++

New support for the GNU® compiler (GCC 3.4) for C++.

The new GNU dialect option supports variable length arrays, anonymous structures, and other constructions allowed by GCC.

For more information, see *Dialect Issues* in the *PolySpace Client/Server for C++ User Guide*.

Polyspace C++ Add-in for Visual Studio

Simplified user interface for Polyspace C++ add-in for Microsoft Visual Studio.

The Polyspace Browser tab has been eliminated from the Visual Studio window. To perform an analysis of a file in Visual Studio, you now simply right-click on the file and select **Start Polyspace**.

For more information, see *Using PolySpace Software in Visual Studio* in the *PolySpace Client/Server for C++ User Guide*.

Operating System Support

Added support for the following operating systems:

- Solaris 2.10
- Windows XP x64 (32-bit mode)

For more information, see the Polyspace Installation Guide.

Polyspace Server for C/C++ Product

Removed Cygwin Software Dependency for Windows Platforms

Compatibility Considerations: Yes

Previous versions of Polyspace products used Cygwin emulation to run UNIX commands on Windows systems.

In version 5.1, the Cygwin software dependency has been removed. Removing Cygwin simplifies the Polyspace product installation process while improving the performance and robustness of the Polyspace Verification process.

Compatibility Considerations

Due to the Cygwin changes, Polyspace Server™ for C/C++ Version 5.1 is not compatible with previous versions of Polyspace products on Windows platforms. To avoid compatibility problems on Windows platforms, you must upgrade all your Polyspace client and server products at the same time.

If your Polyspace server is running on a Windows platform, the binary files used for batch commands in previous releases will not work without Cygwin software installed. In version 5.1, the software provides new .exe files for these batch commands. However, these files are now located in a different location.

Commands	Previous Location	New Location
Standard	*PolyspaceInstallDir*\verifier\bin\	*PolyspaceInstallDir*\verifier\wbin\
Remote Launcher	*Polyspace_Common*\RemoteLauncher\bin\	*Polyspace_Common*\RemoteLauncher\wbin\
Viewer	*Polyspace_Common*\Viewer\bin\	*Polyspace_Common*\Viewer\wbin\

If you wrote scripts using batch commands in previous releases, you must modify the scripts to use the new commands.

In addition, if you used Cygwin shell scripts for postprocessing or target compilation, those scripts will no longer run on version 5.1. To support scripting, the Polyspace software now includes Perl. You can access Perl in:

PolyspaceInstallDir\verifier\tools\perl\win32\bin\perl.exe

Enhanced Installer

Version 5.1 includes an enhanced and simplified installer for all Polyspace products. The installation process is now faster and easier to complete than in previous releases.

For more information, see the Polyspace Installation Guide.

GNU Compiler Support for C++

New support for the GNU compiler (GCC 3.4) for C++.

The new GNU dialect option supports variable length arrays, anonymous structures, and other constructions allowed by GCC.

For more information, see *Dialect Issues* in the *PolySpace Client/Server for C++ User Guide*.

Operating System Support

Added support for the following operating systems:

- Solaris 2.10
- Windows XP x64 (32-bit mode)

For more information, see the Polyspace Installation Guide.