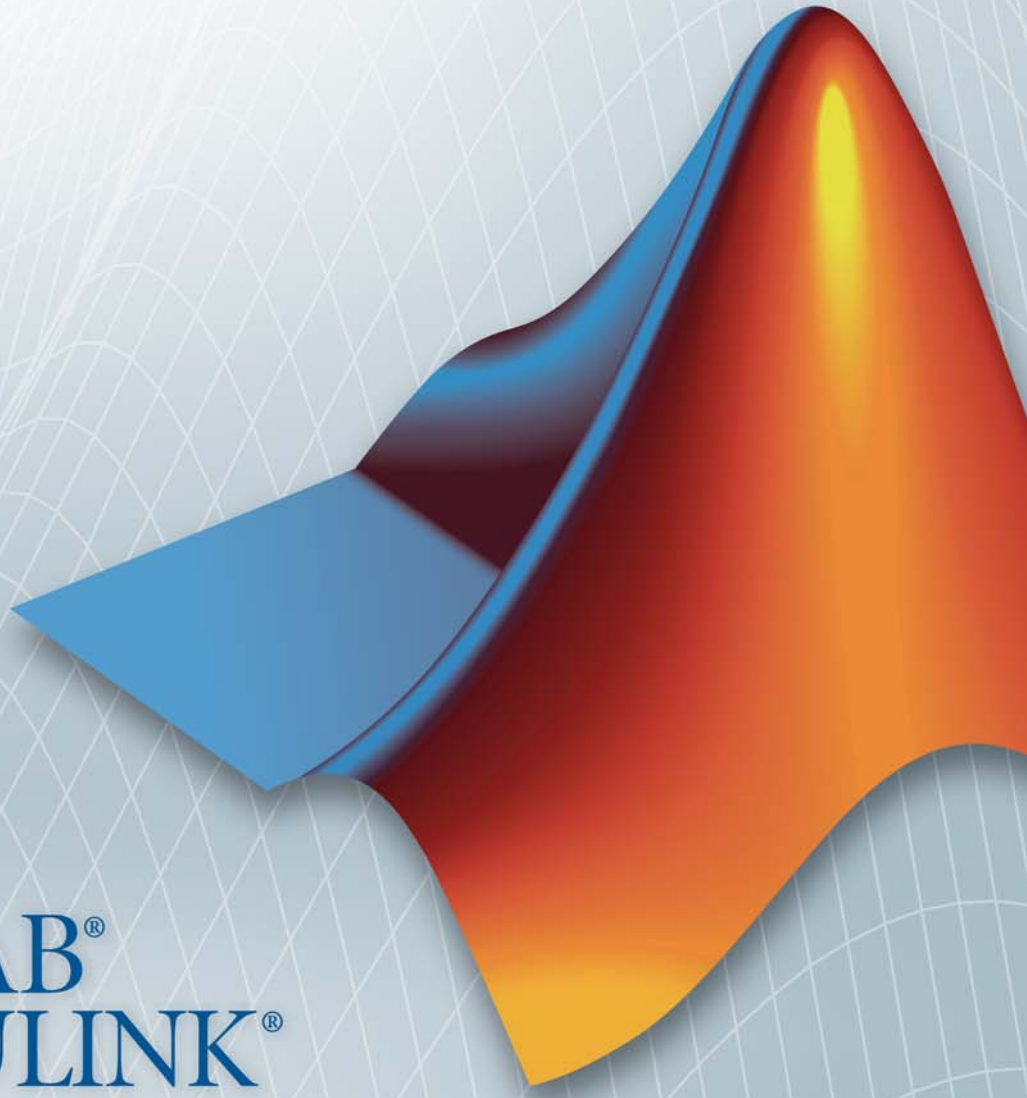
Simulink® Code Inspector™

User's Guide

R2012b



MATLAB®
&SIMULINK®

MathWorks®

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.

Simulink® Code Inspector™ User's Guide

© COPYRIGHT 2011–2012 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2011	Online only	New for Version 1.0 (Release 2011b)
March 2012	Online only	Revised for Version 1.1 (Release 2012a)
September 2012	Online only	Revised for Version 1.2 (Release 2012b)

Contents

1 Getting Started

Product Description 1-2
Key Features 1-2

Why Use This Product? 1-3

Code Inspector Capabilities 1-4
About Code Inspector Capabilities 1-4
Model Interface 1-5
Block Behavior 1-6
Block Connectivity and Execution Order 1-7
Data and File Packaging 1-8
Local Variables 1-9
Limitations 1-9

General Approach to Code Inspection 1-10
Simulink Code Inspector and DO-178 Workflows 1-10
End-to-End General Workflow 1-11

Inspect Generated Code for a Sample Model 1-14

2 Model Compatibility Checking

Model Compatibility Checking Basics 2-2

Check Model Compatibility Using the Graphical User Interface 2-5

Check Model Compatibility Using the Command-Line Interface **2-9**

Fix or Work Around Incompatibilities **2-10**
Fix or Work Around Unsupported Blocks **2-10**
Fix or Work Around Global Data Store Usage **2-10**

3 Code Inspection

Code Inspection Basics **3-2**

Inspect Code Using the Graphical User Interface **3-4**

Inspect Code Using the Command-Line Interface **3-7**

Code Inspection Reports **3-8**
Code Inspection Report Basics **3-8**
Interpret the Overall Inspection Result **3-10**
Analyze Code Verification Results **3-11**
Model Patterns That Might Result in Code Verification Failures **3-16**
Analyze Traceability Results **3-19**

Traceability Matrices **3-24**
Traceability Matrices Basics **3-24**
Prerequisites for Generating a Traceability Matrix **3-25**
Generate a Traceability Matrix **3-26**
Add Comments to a Traceability Matrix **3-26**
Retain Comments When Regenerating a Traceability Matrix **3-27**
Traceability Matrix Limitations **3-27**

4 DO-178C Objectives Compliance

Model-Based Design Workflow in DO-178C **4-2**

Applicable DO-178C Objectives **4-5**

1

Getting Started

- "Product Description" on page 1-2
- "Why Use This Product?" on page 1-3
- "Code Inspector Capabilities" on page 1-4
- "General Approach to Code Inspection" on page 1-10
- "Inspect Generated Code for a Sample Model" on page 1-14

Product Description

Automate source code reviews for safety standards

Simulink® Code Inspector™ automatically compares generated code with its source model to satisfy code-review objectives in DO-178 and other high-integrity standards. The code inspector systematically examines blocks, parameters, and settings in a model to determine whether they are structurally equivalent to operations, operators, and data in the generated code. Simulink Code Inspector provides detailed model-to-code and code-to-model traceability analysis. It generates structural equivalence and traceability reports that you can submit to certification authorities to satisfy DO-178 software coding verification objectives.

Support for industry standards is available through DO Qualification Kit (for DO-178).

Key Features

- Structural equivalence analysis and reports
- Bidirectional traceability analysis and reports
- Compatibility checker to restrict model, block, and coder usage to operations typically used in high-integrity applications
- Tool independence from Simulink code generators
- Tool qualification support (with DO Qualification Kit)

Why Use This Product?

Use Simulink Code Inspector tooling to:

- Prepare for code inspection during model development.
- Run inspections on code generated from models and review reported results.
- Automatically generate code verification reports to support software certification.

While developing a model intended for generating code, you can incrementally and iteratively check the model for compatibility with Code Inspector rules. This process significantly reduces the amount of time to achieve satisfactory inspection results.

For companies and organizations that must certify software under DO-178C, the Code Inspector significantly reduces the time and cost associated with verifying code against requirements. Instead of completing manual line-by-line code reviews with a project checklist, which is time intensive and error prone, you can run the Code Inspector and review a detailed inspection report.

Code Inspector Capabilities

In this section...

"About Code Inspector Capabilities" on page 1-4

"Model Interface" on page 1-5

"Block Behavior" on page 1-6

"Block Connectivity and Execution Order" on page 1-7

"Data and File Packaging" on page 1-8

"Local Variables" on page 1-9

"Limitations" on page 1-9

About Code Inspector Capabilities

Simulink Code Inspector automatically compares generated code with its source model to satisfy code-review objectives in DO-178C and other high-integrity standards. The Code Inspector systematically examines blocks, parameters, and settings in a model to determine whether they are structurally equivalent to operations, operators, and data in the generated code. The tool captures results in structural equivalence and traceability reports.

Sections in this topic provide details about what the Code Inspector examines relative to:

- Model interface
- Block behavior
- Block connectivity and execution order
- Data and file packaging
- Local variables

Each section provides a table that lists what the Code Inspector examines. For each entry the table provides:

- An identifier, which you can use, for example, to refer to an entry from a tool qualification document.
- An example of a condition that the Code Inspector can discover.
- The level of support provided — full or partial; footnotes give more detail for checks providing partial support.

For detailed descriptions of Code Inspector constraints and corresponding model compatibility checks, see “About Model Configuration Constraints”, “About Block Constraints”, and “Simulink Code Inspector Checks”.

Note Before you use Simulink Code Inspector, compare the Code Inspector capabilities with your code review checklist. If you find code review checks for which corresponding Code Inspector capabilities exist, you must separately verify those checklist items.

Model Interface

ID	Check Whether...	Example of Detectable Condition	Level of Support
MDLINTFUNCGEN	Model interface functions were generated	Model step function is missing.	Full
MDLINTDATAGEN	Model interface data structures were generated	Root input data structure for a bus is missing.	Partial*
MDLINTFUNCSIG	Model interface functions have expected signatures	Model step function argument sequence differs from function prototype control specification.	Partial**
MDLINTIOGEN	Expected input and output data structures were generated	External input for initialization function was not initialized as expected.	Partial*

* Arrays and built-in types are supported. For structures, the name or tag is verified, but not the structure fields.
** The data types of interface arguments are not verified in all cases.

Block Behavior

ID	Check Whether...	Example of Detectable Condition	Level of Support
BLKCOMPS	Code generated for a block includes all components of functionality	Code for a Unit Delay block does not include code for updating its state variable.	Full
BLKCOMPSEXP	Code generated for a block includes only expected instances of component functionality	Code includes two independent addition operations that trace to the same Sum block.	Full
BLKCOMPSTRACE	With exception of system logic code, code segments trace back to block component functionality; system logic code traces back to system functionality	A segment of code exists that does not trace back to a block source.	Full
BLKCOMPSCONFIG	Code for block component functionality represents the current block configuration	A Relational Operator block is configured for an equal (==) operation, but it traces to code that applies a not equal (!=) operation.	Full
BLKCOMPSSYSFUNC	Code for block component functionality is in the corresponding system function	The output code for a Unit Delay block is in the start function of the parent system.	Full
BKLCOMPSPROPS	Property settings in the code, such as dimension, complexity, and data type, are compatible with settings for corresponding source blocks	A Gain block with an output data type of `double` traces to code that assigns the block output to variable of type `real32_T`.	Full

Block Connectivity and Execution Order

ID	Check Whether...	Example of Detectable Condition	Level of Support
BLKDATADEPEND	Data dependency between two block components is preserved in the code	A Gain block generates a multiplication operation with one operand as its parameter and another operand as a variable not written to by the source of the Gain block.	Full
BLKDATADEFUSE	Data definition and use dependencies in the code reflect dependencies in the model	A variable buffer is written to by the operation of block A. It is written to again by the operation of block B before a destination block for block A has read the first value.	Full

BLKINPUT	Sources of block input are represented in the code in the expected order	A Gain block uses input from a muxed signal for input ports 1 and 2 (in that order). The generated multiplication code for the Gain block represents the block input sources in a different order than expected. For example,
		`[y1, y2] = [k2, k1] .* [u1 u2]`
		or
		`[y1, y2] = [k1, k2] .* [u2 u1]`
		instead of
		`[y1, y2] = [k1, k2] .* [u1 u2]`
			Full

ID	Check Whether...	Example of Detectable Condition	Level of Support
BLKINDEX	Selection of data in the code uses the expected index	A Gain block is fed by a Bus Selector that selects field `f1` from bus `foobus`. The multiplication operation in the code is on `foobus`.	Full
BLKEXEORDER	Code execution order is consistent with model element execution order	Gain block A feeds a Unit Delay block B. The update code of Unit Delay block B appears before the output code of Gain block A.	Full

Data and File Packaging

ID	Check Whether...	Example of Detectable Condition	Level of Support
SIGOBJAUTO	Signal objects with storage class `auto` are represented in the code as expected	Signal `sig1` is specified with the `auto` storage class. In the code, `sig1` is represented as a global variable instead of an element of the output data structure.	Full
PARAMOBJAUTO	Parameter objects with storage class `auto` are represented in the code as expected	Parameter `K` is specified with the `auto` storage class. In the code, the literal value of the parameter is represented as a global variable instead of its literal value or an element of the parameter data structure.	Full
PARAMINLINE	Inlined parameter values are represented in the code as expected	A Gain block has its **Gain** parameter set to 3.0. The code uses the literal value 4.0 in the multiplication operation.	Full

Local Variables

ID	Check Whether...	Example of Detectable Condition	Level of Support
LCLVARUSED	Local variables are used	Local variable `tmp` is defined but not used	Full
LCLVARDEF	Local variables are defined before being used	Local variable `tmp` is used, but is not defined	Full

Limitations

The Simulink Code Inspector has the following limitations:

- If the code generated for your model contains a step function with the function argument `real-time model`, the Simulink Code Inspector will not inspect the generated code. The `real-time model` function argument is not supported for code generation.

General Approach to Code Inspection

In this section...

"Simulink® Code Inspector™ and DO-178 Workflows" on page 1-10

"End-to-End General Workflow" on page 1-11

Simulink Code Inspector and DO-178 Workflows

The overall workflow for Simulink Code Inspector and meeting DO-178 objectives encompasses the following sub-workflows:

- **Model compatibility checking:** Check a Simulink model for compatibility with Code Inspector rules.

  Model compatibility checking can significantly reduce the amount of time to achieve satisfactory code inspection results by exposing issues early in the model development process. The compatibility checks also promote model, block, and coder usage patterns that tend to align with the needs of high-integrity applications, such as maintaining a high degree of traceability. Model compatibility checking is an incremental and iterative process. For more detail on the model compatibility checking, see "Model Compatibility Checking Basics" on page 2-2.

- **Code inspection:** Run the Code Inspector to compare generated C code with its source model.

  During code inspection, Simulink Code Inspector systematically examines blocks, parameters, and settings in a model to determine whether they are structurally equivalent to operations, operators, and data in the generated code, and generates structural equivalence and traceability reports that can be used to support software certification. Code inspection is an incremental and iterative process. For more detail on code inspection, see "Code Inspection Basics" on page 3-2.

- **Software certification:** Use the code inspection reports as part of a certification package to satisfy DO-178 software coding verification objectives.

  The DO Qualification Kit product provides an artifacts explorer that allows you to manage a certification package for your DO-178C project. The kit also provides detailed information on how to apply Model-Based Design to

DO-178C. For more information, see "Model-Based Design Workflow in DO-178C" on page 4-2 and http://www.mathworks.com/products/do-178/.

- **Tool qualification:** Use the code inspection reports as part of qualifying Simulink Code Inspector for projects based on the DO-178C standard.

  MathWorks® provides a DO Qualification Kit product that supports you in qualifying Simulink Code Inspector and other MathWorks verification tools for projects based on the DO-178C standard. For more information, see "Model-Based Design Workflow in DO-178C" on page 4-2, "Applicable DO-178C Objectives" on page 4-5, and http://www.mathworks.com/products/do-178/.

End-to-End General Workflow

The end-to-end general workflow for Simulink Code Inspector is as follows:

1. Open a model. If you want to operate on a working copy of the model, save a copy of the model to a working folder, and change directory to the work folder.

2. Configure model compatibility checks.

   a. Set the model parameter `AdvancedOptControl` to the value `'-SLCI'`, if it is not already set. This setting constrains the code optimizations that Embedded Coder® uses to a subset that is compatible with code inspection. With the top model window selected, issue the following command:

   ```
   >> set_param(gcs, 'AdvancedOptControl', '-SLCI')
   ```

   b. From the top model window, select **Code > Simulink Code Inspector**. This opens the Simulink Code Inspector dialog box.

   c. Examine the dialog box parameters that apply to model compatibility checking. If you are checking a model that references other models, you can choose to check only the top model or the entire model reference hierarchy. Selecting the option **Inspect all referenced models** includes referenced models in model compatibility checking as well as code inspection.

3. Run model compatibility checks. Click **Check this model** or **Check all models**. The compatibility checker displays a progress bar.

4. Analyze the model compatibility check results.

   a. If you opted to check only the top model, results are displayed directly in the Model Advisor dialog box. You can use the dialog box to explore and rerun individual checks and save the results.

   b. If you opted to check all models, results are displayed in the command window and in an HTML summary report window. You can click links in the HTML summary report to view the detailed Model Advisor Report for each model and referenced model that was checked.

   If the checks pass, the model is ready for code inspection. If incompatibilities are reported, fix or work around the issues and recheck the model for compatibility.

5. Generate C code for the model, if it has not already been generated. You can generate code implicitly as part of code inspection (using the Simulink Code Inspector dialog box option **Generate code before code inspection**), or use Embedded Coder separately to generate the model code. If code was generated previously and placed in a configuration management system, make sure the code is available and ready for inspection.

6. Configure code inspection.

   a. Open the Simulink Code Inspector dialog box, if it is not already open.

   b. Examine and configure the dialog box parameters that apply to code inspection.

      - If you are inspecting a model that references other models, you can choose to inspect only the top model or the entire model reference hierarchy. Selecting the option **Inspect all referenced models** includes referenced models in model compatibility checking as well as code inspection.
      - If your generated code does not use the default Embedded Coder folder structure created by code generation, update the **Code placement** parameter appropriately.
      - Optionally, you can change the location to which code inspection writes the code inspection report, using the dialog box parameter **Report folder**.

7 Inspect the generated code. Click **Inspect Code** or **Generate and inspect code**. The Code Inspector displays a progress bar.

8 Analyze the code inspection results.

   a If you opted to inspect only the top model, results are displayed directly in the detailed code inspection report for the top model.

   b If you opted to inspect all models, results are displayed in an HTML summary report window. You can click links in the HTML summary report to view the detailed code inspection report for each model and referenced model that was inspected.

   If all models get the overall inspection result Passed, the code inspection is complete. If Warning or Failed status is returned for a model, fix or work around the reported conditions and reinspect the model.

9 If you are using the code inspection reports as part of a certification package to satisfy DO-178 software coding verification objectives, further steps will be determined by the larger certification process. For information on how to apply Model-Based Design to meeting DO-178B objectives, see "Model-Based Design Workflow in DO-178C" on page 4-2, "Applicable DO-178C Objectives" on page 4-5, and http://www.mathworks.com/products/do-178/..

Inspect Generated Code for a Sample Model

The following example shows how to use the Simulink Code Inspector dialog box to perform key tasks in the code verification workflow. In this example, you:

- Prepare a model hierarchy for code generation and code inspection.
- Automatically generate code for the model hierarchy.
- Verify the generated code independently of the code generation tool.
- Purposely introduce an error into the generated code and inspect for failure.

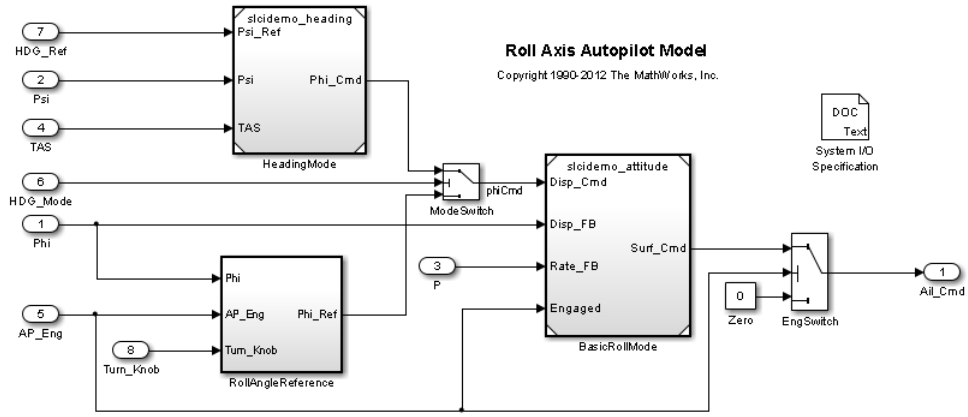
Note The example `slcidemo_intro` illustrates the same code verification workflow using MATLAB commands.

---

1. Open the example model `slcidemo_roll_orig` using the following command:

   ```
   >> slcidemo_roll_orig
   ```

   Save a copy of the model, renaming it to `slcidemo_roll`, and change the MATLAB® current working folder to the location of the saved model. The top level of the model appears as follows.



This model represents a basic roll axis autopilot with two operating modes: roll attitude hold and heading hold. The mode logic for these modes is external to this model. The model architecture represents the heading hold mode and basic roll attitude function as referenced models. The model includes:

- Virtual subsystem `RollAngleReference`, which implements the basic roll angle reference calculation. Embedded Coder code generation inlines this calculation directly into the main function for `slcidemo_roll`.
- Model block `HeadingMode`, referencing a separate model that computes the roll command to track the heading.
- Model block `BasicRollMode`, referencing a separate model that computes the roll attitude control function.

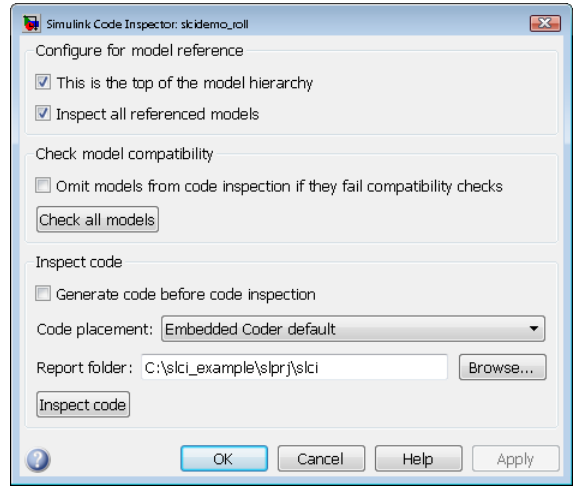
2 Prepare the model for code generation and code inspection.

Note If you try this example with a model other than `slcidemo_roll`, set the model parameter `AdvancedOptControl` to the value `'-SLCI'`. This setting constrains the code optimizations that Embedded Coder uses to a subset that is compatible with code inspection. With the top model window selected, issue the following command:

```
>> set_param(gcs, 'AdvancedOptControl', '-SLCI')
```

a From the top model window, select **Code > Simulink Code Inspector**. The Simulink Code Inspector dialog box opens.

b Configure model compatibility checks. For this example, select **Inspect all referenced models** and click **Apply**. This setting includes referenced models in model compatibility checking as well as code inspection. The dialog box should appear as follows:



c Run the model compatibility checks by clicking **Check all models**. The compatibility checker displays a progress bar.

Results appear in the command window and in an HTML summary report window.

- The MATLAB Command Window displays results similar to the following:

```
Running Model Advisor... ... ...

Systems passed: 3 of 3

Systems with warnings: 0 of 3

Systems failed: 0 of 3
Summary Report
>>
```

- The HTML summary report window displays results similar to the following.

Model Advisor Command-Line Summary

Simulink version: 7.9 **Current run:** 23-Jan-2012 16:37:41

Configuration file: Not Applicable **Number of systems:** 3

Run Summary

Systems passed	3 of 3
Systems with warnings	0 of 3
Systems failed	0 of 3

Systems Run

System	Passed	Failed	Warnings	Not Run	Model Advisor Report
slcidemo_attitude	31	0	0	0	../Report.html
slcidemo_heading	31	0	0	0	../Report.html
slcidemo_roll	31	0	0	0	../Report.html

Valid Check IDs

mathworks.slci.CodeGenerationSettings

mathworks.slci.DataImportSettings

mathworks.slci.DiagnosticsSettings

Note This HTML report also is linked from the command window results. It is saved as file `summaryReport.html` in subfolder `/slprj/modeladvisor` under the current working folder.

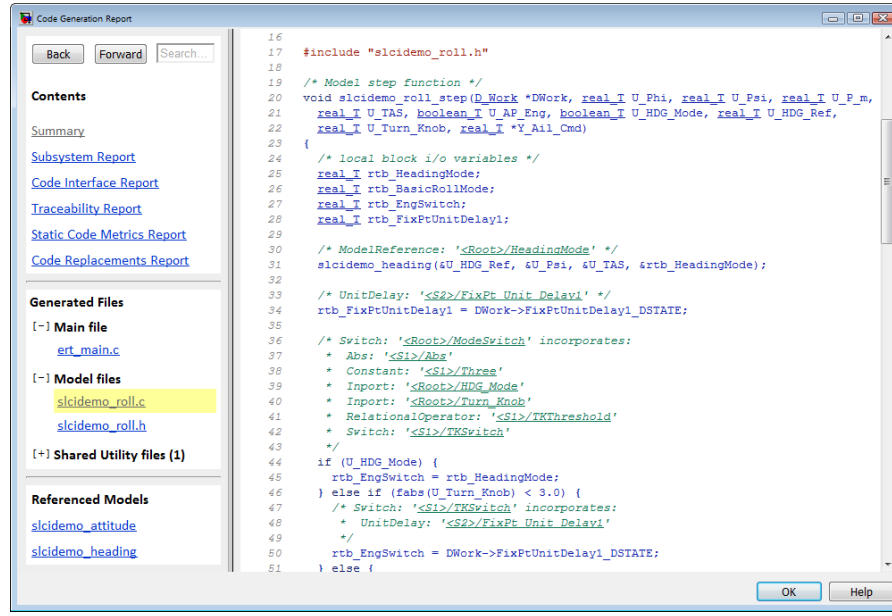
3 Generate code for the model. You can generate code implicitly as part of code inspection (using the Simulink Code Inspector dialog box option

Generate code before code inspection), or perform code generation and code inspection as separate steps. This example separates the code generation step from the code inspection step.

a In the top model window, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box. In the **Code Generation > Report** pane, select the option **Open report automatically**. (If you try this example with a model other than `slcidemo_roll`, also select **Code-to-model**, **Model-to-code**, and the four **Traceability Report Contents** options in the **Report** pane.) Click **Apply** and save the model changes.

b Go to the **Code Generation** main pane and click **Generate code**. (If the **Generate code** button does not appear for your model, select the **Generate code only** option to enable the button.) Progress is displayed in the MATLAB Command Window.

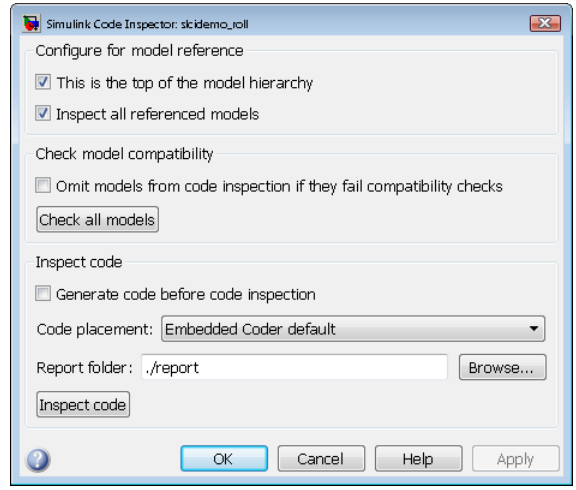
c Embedded Coder code generation displays results in an HTML report window.



4 Inspect the generated code.

a Open the Simulink Code Inspector dialog box if it is not already open, and examine the code inspection parameter settings. The **Code placement** parameter is set to `Embedded Coder default`, which configures code inspection to use the default Embedded Coder folder structure created by code generation.

b Optionally, you can change the location to which code inspection writes the code inspection report, using the dialog box parameter **Report folder**. For example, enter the path string `./report` and click **Apply**.



c To inspect the generated code, click **Inspect Code**. The Code Inspector displays a progress bar.

d The Code Inspector displays a summary in an HTML report window.

Simulink Code Inspector Summary Report for slcidemo_roll

Model	Code inspection status	Report
slcidemo_roll	Passed	slcidemo_roll_report.html
slcidemo_attitude	Passed	slcidemo_attitude_report.html
slcidemo_heading	Passed	slcidemo_heading_report.html

The summary report links to detailed code inspection reports for the top model and each referenced model. For example, here is the topmost portion of the code inspection report for the top model, `slcidemo_roll`.

Simulink Code Inspector Report for slcidemo_roll.slx

Inspected Model File :	S:\matlabwork\example\slcidemo_roll.slx
Model Version :	1.81
Simulink Version :	8.0 (R2012b)
Model Checksum :	1691736902 410456173 2825258068 1900093013
Model Last Modified On :	12-Apr-2012 08:38:30
Inspected Code Files :	S:\matlabwork\example\slcidemo_roll_ert_rtw/slcidemo_roll.c
Code Inspection Run On :	12-Apr-2012 08:39:49

Overall Inspection Result : Passed

Code Verification Results : Verified

Function Interface Verification Results : Verified

Function	Status	Details
slcidemo_roll_initialize	Verified	-
slcidemo_roll_step	Verified	-

Model To Code Verification Results : Verified

Status	Details	
Verified	Model objects with status Verified :	42
	Model objects with status Partially processed :	0
	Model objects with status Unable to process :	0
	Model objects with status Failed to verify :	0

Code To Model Verification Results : Verified

Function	Status	Details	
slcidemo_roll_initialize	Verified	Lines of code with status Verified :	2
		Lines of code with status Partially processed :	0
		Lines of code with status Unable to process :	0
		Lines of code with status Failed to verify :	0

The summary report and the detailed code inspection reports are saved as HTML files in the **Report folder** location that you specified.

5 Insert an error into the generated code and inspect for failure.

To show a failed result, this example inserts an intentional error in the generated code. The Logical Operator block inside the `RollAngleReference` subsystem is changed in the generated code from an OR operation (`||`) to an AND operation (`&&`), using the example utility function `slcidemo_modifycode`.

a To highlight the block for which corresponding code is modified, issue the following command:

```
>> hilite_system('slcidemo_roll/RollAngleReference/Or');
```

b To modify the OR to an AND, issue the following commands:

```
>> cfile = fullfile('.','slcidemo_roll_ert_rtw','slcidemo_roll.c');
>> slcidemo_modifycode(cfile,'<S1>/Or','||','&&')
```

The `slcidemo_modifycode` utility function displays the following output:

```
Modified line 51 of file .\slcidemo_roll_ert_rtw\slcidemo_roll.c.
Before:     if ((U_Phi >= 6.0) || (U_Phi <= -6.0)) {
After :     if ((U_Phi >= 6.0) && (U_Phi <= -6.0)) {
```

c To reinspect the generated code, open the Simulink Code Inspector dialog box if it is not already open, and click **Inspect code**.

d View the code inspection reports.

The summary report displays a failure for the top model.

Simulink Code Inspector Summary Report for slcidemo_roll

Model	Code inspection status	Report
slcidemo_roll	Failed	slcidemo_roll_report.html
slcidemo_attitude	Passed	slcidemo_attitude_report.html
slcidemo_heading	Passed	slcidemo_heading_report.html

The code inspection report for the top model contains several indications of a failed comparison between the Logical Operator block and the corresponding code. The top of the report shows the following.

Simulink Code Inspector Report for slcidemo_roll.slx

Inspected Model File :	S:\matlabwork\example\slcidemo_roll.slx
Model Version :	1.80
Simulink Version :	8.0 (R2012b)
Model Checksum :	1691736902 410456173 2825258068 1900093013
Model Last Modified On :	12-Apr-2012 09:30:52
Inspected Code Files :	S:\matlabwork\example\slcidemo_roll_ert_rtw/slcidemo_roll.c
Code Inspection Run On :	12-Apr-2012 09:43:29

Overall Inspection Result : Failed

Code Verification Results : Failed to verify

Function Interface Verification Results : Verified

Function	Status	Details
slcidemo_roll_initialize	Verified	-
slcidemo_roll_step	Verified	-

Model To Code Verification Results : Failed to verify

Status	Details	
Failed to verify	Model objects with status Verified :	40
	Model objects with status Partially processed :	0
	Model objects with status Unable to process :	0
	Model objects with status Failed to verify :	2

Code To Model Verification Results : Failed to verify

Function	Status	Details	
slcidemo_roll_initialize	Verified	Lines of code with status Verified :	2
		Lines of code with status Partially processed :	0
		Lines of code with status Unable to process :	0

Further down in the report, under **Code Verification Details > Model-to Code Verification**, the mismatch between block and code is flagged.

<model>/RollAngleReference/MinusSix	Verified	-
<model>/RollAngleReference/NotEngaged	Verified	-
<model>/RollAngleReference/Or	Failed to verify	Failed to verify (Unable to match) for model step <model>/RollAngleReference/LatchPhi/FixPt Unit Delay1 (*DWork.FixPtUnitDelay1_DSTATE)
<model>/RollAngleReference/RefSwitch	Failed to verify	Failed to verify (Unable to match) for model step <model>/RollAngleReference/LatchPhi/FixPt Unit Delay1 (*DWork.FixPtUnitDelay1_DSTATE)
<model>/RollAngleReference/RefThreshold1	Verified	-
<model>/RollAngleReference/RefThreshold2	Verified	-

Additionally, under **Traceability Details > Model-to Code Traceability**, the mismatch between block and code is flagged.

<model>/RollAngleReference/MinusSix	slcidemo_roll.c:92	-
<model>/RollAngleReference/NotEngaged	slcidemo_roll.c:83 , 103	-
<model>/RollAngleReference/Or	-	Failed to trace (Verification status : Failed to verify)
<model>/RollAngleReference/RefSwitch	slcidemo_roll.c:92 , 94-95 , 99	Failed to trace (Verification status : Failed to verify)
<model>/RollAngleReference/RefThreshold1	slcidemo_roll.c:92	-
<model>/RollAngleReference/RefThreshold2	slcidemo_roll.c:92	-

6 Optionally, try modifying the model or other aspects of the generated code to see how code inspection results are affected.

2

Model Compatibility Checking

- "Model Compatibility Checking Basics" on page 2-2
- "Check Model Compatibility Using the Graphical User Interface" on page 2-5
- "Check Model Compatibility Using the Command-Line Interface" on page 2-9
- "Fix or Work Around Incompatibilities" on page 2-10

Model Compatibility Checking Basics

When developing a model from which you intend to generate code that will be verified using Simulink Code Inspector, you can incrementally and iteratively check the model for compatibility with Code Inspector rules. Model compatibility checking can significantly reduce the amount of time to achieve satisfactory code inspection results by exposing issues early in the model development process. The compatibility checks also promote model, block, and coder usage patterns that tend to align with the needs of high-integrity applications, such as maintaining a high degree of traceability.

During a model compatibility check, the software checks for model and block configuration settings that help produce an in-memory representation of the model that is compatible with Code Inspector rules. You can set model and block configuration parameters many different ways and produce a compatible in-memory representation. Compatibility checks scan for a subset of those ways. Although a model can fail a compatibility check and still pass inspection, passing compatibility checks increases the likelihood of satisfactory code inspection.

The compatibility checks look for conditions that violate Code Inspector constraints on model configuration parameters, other modelwide attributes, and block usage. Items affected by Code Inspector constraints include:

- Model parameters for
  - Solver use
  - Data import/export
  - Optimization
  - Diagnostics
  - Hardware implementation
  - Model referencing
  - Code generation
- Modelwide attributes
  - Unconnected objects
  - Function specifications

  - Model arguments
  - Unsupported blocks
  - Tunable workspace variables
  - Sample times
  - Global data stores
  - Fixed-point instrumentation
  - Root outport usage
  - Bus usage
- Block usage
  - Data types and ports
  - Block parameters

For detailed description of Code Inspector constraints and the corresponding model compatibility checks, see “About Model Configuration Constraints”, “About Block Constraints”, and “Simulink Code Inspector Checks”.

To initiate compatibility checking for your model, you can do any of the following:

- From the model window, select **Code > Simulink Code Inspector**, and use the Simulink Code Inspector dialog box to control model compatibility checking. For more information, see “Check Model Compatibility Using the Graphical User Interface” on page 2-5.
- Use the slci.Configuration interface to programmatically control model compatibility checking. For more information, see “Check Model Compatibility Using the Command-Line Interface” on page 2-9.
- Use the slciadvisor interface to open an SLCI Advisor session (equivalent to Model Advisor preloaded with Simulink Code Inspector checks) for the open model. This function provides direct access to SLCI model compatibility checking that can streamline iterative checking of a model.

Model compatibility checking generates a detailed HTML report for each model checked. If you checked models in a model reference hierarchy, the software reports summary status at the MATLAB command line and displays

a summary HTML report. You can click links in the HTML summary report to view the detailed Model Advisor Report for each model and referenced model that was checked. If you checked only one model, the detailed model results are displayed directly in a Model Advisor dialog box.

In the detailed results, the result of each check is explained, and if you need to fix your model, recommended actions are provided. The available model compatibility checks are listed in report order and described in the "Simulink Code Inspector Checks" reference.

Check Model Compatibility Using the Graphical User Interface

1. Open a model that you want to check for compatibility with Simulink Code Inspector. To use an example model, you can do the following:

   a. Open the example model `slcidemo_roll_orig` using the following command:

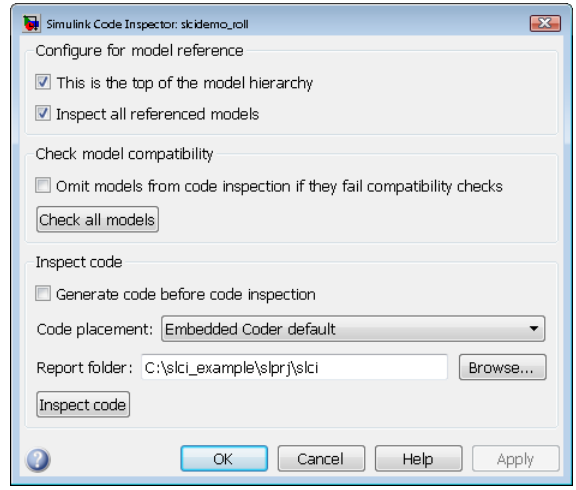
   ```
   >> slcidemo_roll_orig
   ```

   b. Save a copy of the model to a work folder, renaming it to `slcidemo_roll`. Change directory to the work folder.

2. Open the Simulink Code Inspector dialog box and configure model compatibility checks.

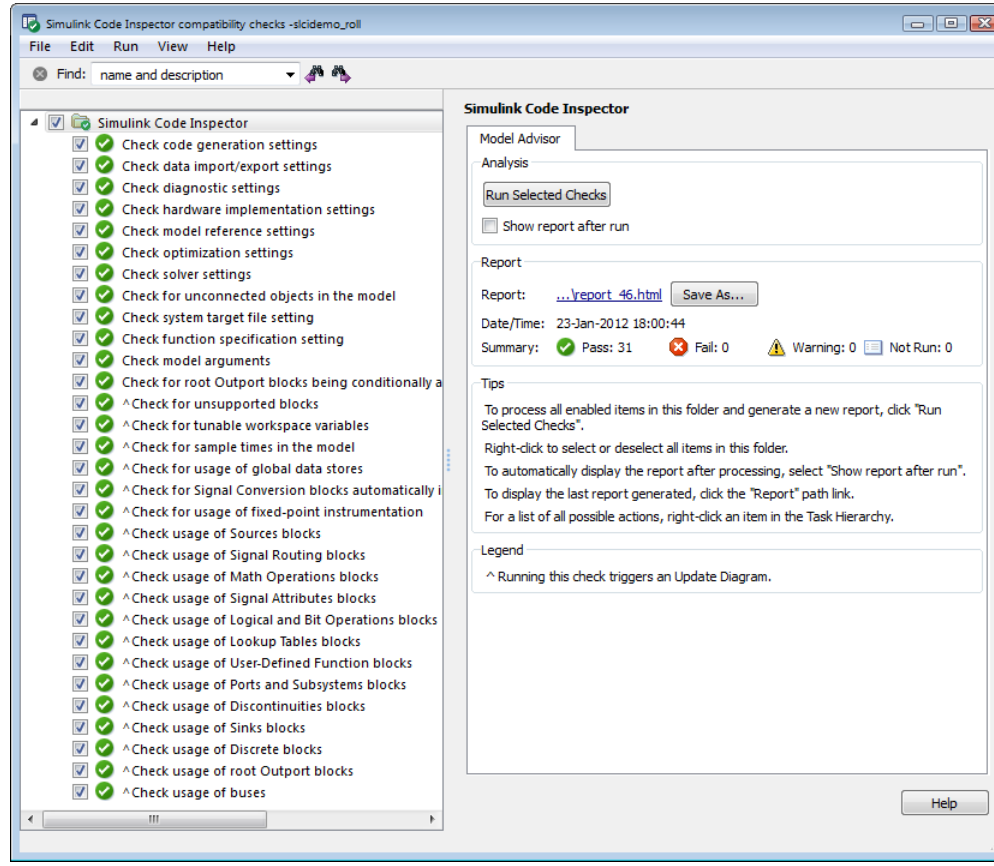
   a. From the top model window, select **Code > Simulink Code Inspector**.

   b. Examine the parameters that apply to model compatibility checking. If you are checking a model that references other models, consider whether to select the option **Inspect all referenced models**. This option includes referenced models in model compatibility checking as well as code inspection. If you select this option, the button **Check this model** changes to **Check all models**.



3 To run model compatibility checks, click **Check this model** or **Check all models**. The compatibility checker displays a progress bar.

4 If you opted to check only the top model, results are displayed directly in the Model Advisor dialog box. You can use the dialog box to explore and rerun individual checks and save the results.



If you opted to check all models, results are displayed in the command window and in an HTML summary report window.

- The MATLAB Command Window displays results similar to the following:

```
Running Model Advisor... ... ...

Systems passed: 3 of 3

Systems with warnings: 0 of 3

Systems failed: 0 of 3
Summary Report
>>
```

- The HTML summary report window displays results similar to the following:

Model Advisor Command-Line Summary

Simulink version: 7.9 **Current run:** 23-Jan-2012 16:37:41

Configuration file: Not Applicable **Number of systems:** 3

Run Summary

Systems passed	3 of 3
Systems with warnings	0 of 3
Systems failed	0 of 3

Systems Run

System	Passed	Failed	Warnings	Not Run	Model Advisor Report
slcidemo_attitude	31	0	0	0	../Report.html
slcidemo_heading	31	0	0	0	../Report.html
slcidemo_roll	31	0	0	0	../Report.html

Valid Check IDs

mathworks.slci.CodeGenerationSettings

mathworks.slci.DataImportSettings

mathworks.slci.DiagnosticsSettings

Note This HTML report also is linked from the command window results. It is saved as file `summaryReport.html` in subfolder `/slprj/modeladvisor` under the current working folder.

To view the detailed Model Advisor Report for a model listed in the HTML summary report, go to the **Systems Run** table, and click the corresponding link in the **Model Advisor Report** column.

5 If the checks pass, the model is ready for inspection. If incompatibilities are reported, fix the issues and recheck the model for compatibility.

Check Model Compatibility Using the Command-Line Interface

To programmatically control model compatibility checking, use the `slci.Configuration` interface.

In the MATLAB Command Window or within a script, you issue a call to `slci.Configuration.checkCompatibility`, specifying the handle to a Simulink Code Inspector configuration object for the model, previously returned by *`cfgObj`* `= slci.Configuration(`*`modelName`*`);`. The `checkCompatibility` function returns objects containing results information.

The following example shows how to programmatically run the compatibility checker and report results.

```
fprintf('\nInvoking compatibility checker ...\n');

config = slci.Configuration('slcidemo_roll');
result = config.checkCompatibility('DisplayResults','None');

for i = 1:length(result)
    fprintf('\nModel ''%s'' passed %d checks with %d issues.',...
        result{i}.system,...
        result{i}.numPass, result{i}.numWarn + result{i}.numFail)
end
```

If the checks pass, the model is ready for inspection. If incompatibilities are reported, fix the issues and recheck the model for compatibility.

For an example of using the command-line interface to control the complete code inspection workflow, see the example `slcidemo_intro`.

Fix or Work Around Incompatibilities

In this section...

"Fix or Work Around Unsupported Blocks" on page 2-10

"Fix or Work Around Global Data Store Usage" on page 2-10

Fix or Work Around Unsupported Blocks

If the compatibility checker identifies one or more unsupported blocks in your model, possible actions include:

- Replace an unsupported block with a supported block. Supported blocks are listed in "Supported Blocks — By Category", and also can be viewed in the `slcilib` block library.
- Replace an unsupported block with an equivalent combination of supported blocks.
- Replace an unsupported block with an S-Function block created using the Legacy Code Tool.
- If one or more unsupported blocks cannot be removed, use referenced models to isolate the unsupported block(s), and/or use a partial verification work flow that omits the unsupported block(s).

Fix or Work Around Global Data Store Usage

If the compatibility checker identifies one or more externally defined signal objects that are being referenced as global data stores by Data Store Read or Write blocks in the model, possible actions include:

- If possible, avoid use of externally defined signal objects that are referenced as global data stores by Data Store Read or Data Store Write blocks. This usage causes Simulink software to create hidden Data Store memory blocks at root level, which is incompatible with code inspection.
- Move the affected Data Store Read or Data Store Write blocks into Model blocks.

3

Code Inspection

- "Code Inspection Basics" on page 3-2
- "Inspect Code Using the Graphical User Interface" on page 3-4
- "Inspect Code Using the Command-Line Interface" on page 3-7
- "Code Inspection Reports" on page 3-8
- "Traceability Matrices" on page 3-24

Code Inspection Basics

Code inspection automatically compares generated code with its source model to satisfy code-review objectives in DO-178C and other high-integrity standards. The code inspection process builds an in-memory representation of the model that is independent of the code generation process. The Code Inspector systematically examines blocks, parameters, and settings in a model to determine whether they are structurally equivalent to operations, operators, and data in the generated code, and generates reports that can be used to support software certification.

The aspects of a Simulink model that are analyzed by code inspection include the following:

- Model interface
- Block behavior
- Block connectivity and execution order
- Data and file packaging
- Local variables

For more information on what the Code Inspector examines, see “Code Inspector Capabilities” on page 1-4.

When developing a model from which you intend to generate code that will be verified using Simulink Code Inspector, you can incrementally and iteratively check the model for compatibility with Code Inspector rules. Model compatibility checking can significantly reduce the amount of time to achieve satisfactory code inspection results by exposing issues early in the model development process. Before inspecting the code for a model, you should check that the model passes Simulink Code Inspector compatibility checks. For more information, see “Model Compatibility”.

You can generate the model code to be inspected as part of code inspection, or perform code generation and code inspection as separate steps.

To initiate code inspection for a model that has passed Simulink Code Inspector compatibility checks, you can do either of the following:

- From the model window, select **Code > Simulink Code Inspector**, and use the Simulink Code Inspector dialog box to control code inspection. For more information, see "Inspect Code Using the Graphical User Interface" on page 3-4.
- Use the slci.Configuration interface to programmatically control code inspection. For more information, see "Inspect Code Using the Command-Line Interface" on page 3-7.

Code inspection generates a detailed HTML report for each model inspected. If you inspected all models in a model reference hierarchy, the software displays a summary HTML report. You can click links in the HTML summary report to view the detailed code inspection report for each model and referenced model that was inspected. If you inspected only one model, the model results are displayed directly in a detailed code inspection report.

The detailed report for a model documents code verification and traceability results. The code inspection report contains the following major sections:

- Code Verification — Summary and detailed reports on verification of structural equivalence between model and code elements.
- Traceability — Summary and detailed reports on model-to-code and code-to-model traceability

For companies and organizations that must certify software under DO-178C, the Code Inspector significantly reduces the time and cost associated with verifying code against requirements. Instead of completing manual line-by-line code reviews with a project checklist, which is time intensive and error prone, you can run the Code Inspector and review a detailed inspection report. For more information about the code inspection report, see "Code Inspection Reports" on page 3-8.

Inspect Code Using the Graphical User Interface

1. Open a model for which you want to generate and inspect code using Simulink Code Inspector. To use an example model, you can do the following:

   a. Open the example model `slcidemo_roll_orig` using the following command:

   ```
   >> slcidemo_roll_orig
   ```

   b. Save a copy of the model to a work folder, renaming it to `slcidemo_roll`. Change directory to the work folder.

2. If the model has not previously passed model compatibility checking, follow the procedure in "Check Model Compatibility Using the Graphical User Interface" on page 2-5. When the model passes the Simulink Code Inspector compatibility checks, return to this procedure.

3. Generate code for the model. You can generate code implicitly as part of code inspection (using the Simulink Code Inspector dialog box option **Generate code before code inspection**), or perform code generation and code inspection as separate steps. This example separates the code generation step from the code inspection step.

   a. In the top model window, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box. If you want to generate an HTML code generation report for later reference (recommended), go to the **Code Generation > Report** pane, and select the option **Open report automatically**. (If you try this example with a model other than `slcidemo_roll`, it is recommended to select all options in the **Report** pane.) Click **OK** and save the model changes.

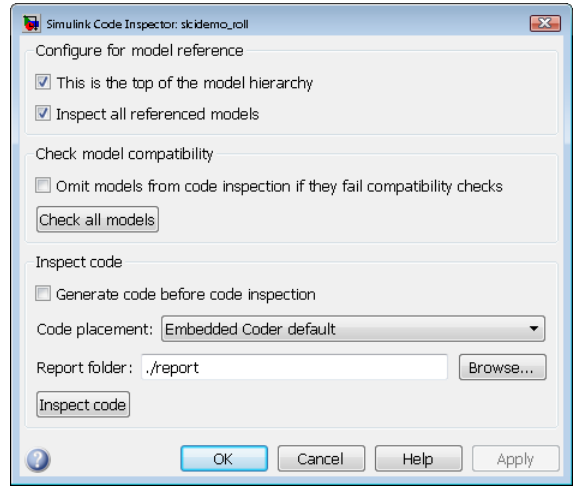
   b. Go to the **Code Generation** main pane and click **Generate code**. (If the **Generate code** button does not appear for your model, select the **Generate code only** option to enable the button.) Progress is displayed in the MATLAB Command Window.

   c. Embedded Coder code generation displays results in an HTML report window.

4. Inspect the generated code.

a Open the Simulink Code Inspector dialog box if it is not already open, and examine the code inspection parameter settings. The **Code placement** parameter is set to `Embedded Coder default`, which configures code inspection to use the default Embedded Coder folder structure created by code generation.

b Optionally, you can change the location to which code inspection writes the code inspection report, using the dialog box parameter **Report folder**. For example, enter the path string `./report` and click **Apply**.



c To inspect the generated code, click **Inspect Code**. The Code Inspector displays a progress bar.

d The Code Inspector displays a summary in an HTML report window:

Simulink Code Inspector Summary Report for slcidemo_roll

Model	Code inspection status	Report
slcidemo_roll	Passed	slcidemo_roll_report.html
slcidemo_attitude	Passed	slcidemo_attitude_report.html
slcidemo_heading	Passed	slcidemo_heading_report.html

The summary report links to detailed code inspection reports for the top model and each referenced model. For example, here is the topmost portion of the code inspection report for the top model, `slcidemo_roll`:

Simulink Code Inspector Report for slcidemo_roll.slx

Inspected Model File :	S:\matlabwork\example\slcidemo_roll.slx
Model Version :	1.81
Simulink Version :	8.0 (R2012b)
Model Checksum :	1691736902 410456173 2825258068 1900093013
Model Last Modified On :	12-Apr-2012 08:38:30
Inspected Code Files :	S:\matlabwork\example\slcidemo_roll_ert_rtw/slcidemo_roll.c
Code Inspection Run On :	12-Apr-2012 08:39:49

Overall Inspection Result : Passed

Code Verification Results : Verified

Function Interface Verification Results : Verified

Function	Status	Details
slcidemo_roll_initialize	Verified	-
slcidemo_roll_step	Verified	-

Model To Code Verification Results : Verified

Status	Details	
Verified	Model objects with status Verified :	42
	Model objects with status Partially processed :	0
	Model objects with status Unable to process :	0
	Model objects with status Failed to verify :	0

Code To Model Verification Results : Verified

Function	Status	Details	
slcidemo_roll_initialize	Verified	Lines of code with status Verified :	2
		Lines of code with status Partially processed :	0
		Lines of code with status Unable to process :	0
		Lines of code with status Failed to verify :	0

The summary report and the detailed code inspection reports are saved as HTML files in the **Report folder** location you specified.

Inspect Code Using the Command-Line Interface

To programmatically control code inspection, use the `slci.Configuration` interface.

In the MATLAB Command Window or within a script, you issue a call to `slci.Configuration.inspect`, specifying the handle to a Simulink Code Inspector configuration object for the model, previously returned by *`cfgObj`* = `slci.Configuration(`*`modelName`*`);`. The `inspect` function returns objects containing results information.

The following example shows how to programmatically run the Code Inspector and report results. The model is assumed to have previously passed Simulink Code Inspector compatibility checks (see `slci.Configuration.checkCompatibility`).

```
config = slci.Configuration('slcidemo_roll');
config.setTopModel(true);
config.setReportFolder(fullfile('.','report'));
result = config.inspect('DisplayResults','None');
fprintf('Model %s status: %s\n',result.ModelName, result.Status);
```

The inspection report is placed at the location specified in the call to `slci.Configuration.SetReportFolder`, which is the `report` subfolder of the current working folder. To display the generated report, issue the following command:

```
web(fullfile('.', 'report','slcidemo_roll_report.html'));
```

For an example of using the command-line interface to control the complete code inspection workflow, see the example `slcidemo_intro`.

Code Inspection Reports

In this section...

"Code Inspection Report Basics" on page 3-8

"Interpret the Overall Inspection Result" on page 3-10

"Analyze Code Verification Results" on page 3-11

"Model Patterns That Might Result in Code Verification Failures" on page 3-16

"Analyze Traceability Results" on page 3-19

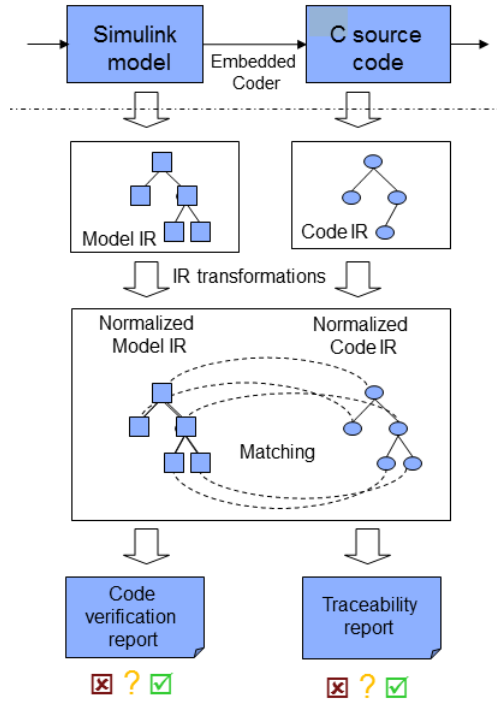
Code Inspection Report Basics

Code inspection generates an HTML code inspection report for a Simulink model and its generated code. The report provides detailed analysis of structural equivalence and bidirectional traceability between the model and the code generated from the model.

Code inspection automatically compares generated code with its source model to satisfy code-review objectives in DO-178C and other high-integrity standards. The code inspection process builds an in-memory representation of the model that is independent of the code generation process. The Code Inspector systematically examines blocks, parameters, and settings in a model to determine whether they are structurally equivalent to operations, operators, and data in the generated code, and generates reports that can be used to support software certification.

Simulink Code Inspector code inspection carries out a *translation validation*. Inputs to the Code Inspector are a Simulink model and the C source code generated by the Embedded Coder code generator for the model. The Code Inspector processes these two inputs into internal representations (IRs), called *model IR* and *code IR*. These IRs are transformed into normalized representations to facilitate further analysis. In this process, the model IR represents the expected pattern, and the code IR constitutes the actual pattern to be verified. To verify the generated code, the Code Inspector attempts to match the normalized model IR with the normalized code IR.

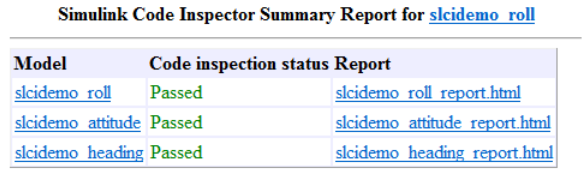
Note Simulink Code Inspector code inspection has been implemented independently of Embedded Coder code generation. In particular, the IRs used by Code Inspector are different from the IRs used by the code generator.



Use of normalization techniques allows the Code Inspector to inspect code generated by an highly-optimizing code generator. The results of this matching process are reported to the user by means of a code verification report and a traceability report. When code inspection completes, the code verification report documents the translation validation process, and the traceability report provides a bidirectional mapping between model elements and their counterparts in the generated code.

The code inspection results are presented in a hierarchical manner. A summary report lists the top model and, if the code inspection encompassed

referenced models, each model in the model reference hierarchy. For each model, the summary report provides aggregated status information and a link to a detailed code inspection report for the model.



Simulink Code Inspector Summary Report for slcidemo_roll

Model	Code inspection status	Report
slcidemo_roll	Passed	slcidemo_roll_report.html
slcidemo_attitude	Passed	slcidemo_attitude_report.html
slcidemo_heading	Passed	slcidemo_heading_report.html

The detailed code inspection reports provide the following information for each model:

- Overall Inspection Result — **Passed**, **Warning**, or **Failed** — based on aggregated status of the code verification and traceability results
- Code Verification Results — Summary and detailed reports on verification of structural equivalence between model and code elements
- Traceability Results — Summary and detailed reports on bidirectional model-to-code mapping

Interpret the Overall Inspection Result

The detailed code inspection report for a model or submodel provides a header section that includes the **Overall Inspection Result** value for the model — **Passed**, **Warning**, or **Failed** — based on aggregated status of the code verification and traceability results.

Simulink Code Inspector Report for slcidemo_roll.slx

Inspected Model File : S:\matlabwork\example\slcidemo_roll.slx
Model Version : 1.82
Simulink Version : 8.0 (R2012b)
Model Checksum : 114891003 1569487027 1850181305 2434736257
Model Last Modified On : 13-Apr-2012 11:40:14
Inspected Code Files : S:\matlabwork\example\slcidemo_roll_ert_rtw/slcidemo_roll.c
Code Inspection Run On : 13-Apr-2012 11:40:47

Overall Inspection Result : Passed

The **Overall Inspection Result** value is aggregated from the following values:

- **Code Verification Results** — The overall code verification result aggregated from the code verification report subsections. Possible values are **Verified**, **Partially verified**, or **Failed to verify**. For more information about how the **Code Verification Results** value is aggregated, see "Analyze Code Verification Results" on page 3-11.
- **Traceability Results** — The overall traceability result aggregated from the traceability report subsections. Possible values are **Traced**, **Partially traced**, or **Failed to trace**. For more information about how the **Traceability Results** value is aggregated, see "Analyze Traceability Results" on page 3-19.

The following table shows how code verification and traceability results are aggregated into the **Overall Inspection Result**.

	Verified	**Partially verified**	**Failed to verify**
Traced	Passed	Warning	Failed
Partially traced	Warning	Warning	Failed
Failed to trace	Failed	Failed	Failed

Analyze Code Verification Results

The detailed code inspection report for a model or submodel provides sections named **Code Verification Results** and **Code Verification Details**, which provide summary and detailed reports on verification of structural equivalence between the model and code generated from the model. The code verification report provides information about:

- Verification of the interfaces of generated code functions
- Verification of structural equivalence between model and code
- Temporary variable usage

The following sample report excerpt shows summary code verification results for generated code that is structurally equivalent to its corresponding model. Model elements that are outside of the supported language subset and

corresponding code fragments are indicated as “Unable to process” in the code verification report.

Code Verification Results : Verified

Function Interface Verification Results : Verified

Function	Status	Details
slcidemo_roll_initialize	Verified	-
slcidemo_roll_step	Verified	-

Model To Code Verification Results : Verified

Status	Details	
Verified	Model objects with status Verified :	38
	Model objects with status Partially processed :	0
	Model objects with status Unable to process :	0
	Model objects with status Failed to verify :	0

Code To Model Verification Results : Verified

Function	Status	Details	
slcidemo_roll_initialize	Verified	Lines of code with status Verified :	2
		Lines of code with status Partially processed :	0
		Lines of code with status Unable to process :	0
		Lines of code with status Failed to verify :	0
slcidemo_roll_step	Verified	Lines of code with status Verified :	22
		Lines of code with status Partially processed :	0
		Lines of code with status Unable to process :	0
		Lines of code with status Failed to verify :	0

Temporary Variable Usage Results : Verified

Function	Status	Details
slcidemo_roll_initialize	Verified	Function does not have any temporary variable declarations
slcidemo_roll_step	Verified	Temporary variables with status Failed to verify : 0 Temporary variables with status Verified : 4

The **Code Verification Results** section displays:

- The overall code verification result value aggregated from the code verification report subsections. Possible values for **Code Verification Results** are **Verified**, **Partially verified**, or **Failed to verify**.
- Subsection-level verification results:
  - **Function Interface Verification Results**
  - **Model To Code Verification Results**
  - **Code To Model Verification Results**
  - **Temporary Variable Usage Results**

  The subsection-level verification result values are aggregated from the verification status of every *object* (function interface, model element, code line, or temporary variable) in the subsection. Each subsection returns the value **Verified**, **Partially verified**, or **Failed to verify**.
- Object level verification results. Each function interface, model element, code line, or temporary variable within a subsection returns a verification status value:
  - For model elements or code lines: **Verified**, **Partially processed**, **Unable to process**, or **Failed to verify**.
  - For function interfaces or temporary variables: **Verified** or **Failed to verify**.

When the object-level verification status is aggregated to produce the subsection-level status value, the most severe object status becomes the subsection status.

Most Severe Object-Level Status	Resulting Subsection-Level Status
Failed to verify	**Failed to verify**
Partially processed or **Unable to process**	**Partially verified**
Verified	**Verified**

Correspondingly, when the section-level verification status is aggregated to produce the **Code Verification Results** value, the most severe subsection status becomes the overall code verification status.

Most Severe Subsection-Level Status	Resulting Overall Code Verification Status
Failed to verify	Failed to verify
Partially verified	Partially verified
Verified	Verified

The following sample report excerpt illustrates how code verification status is aggregated when one or more objects fails to verify.

Overall Inspection Result : Failed

Code Verification Results : Failed to verify

Function Interface Verification Results : Verified

Function	Status	Details
slcidemo_roll_initialize	Verified	-
slcidemo_roll_step	Verified	-

Model To Code Verification Results : Failed to verify

Status	Details	
Failed to verify	Model objects with status Verified :	36
	Model objects with status Partially processed :	0
	Model objects with status Unable to process :	0
	Model objects with status Failed to verify :	2

Code To Model Verification Results : Failed to verify

Function	Status	Details	
slcidemo_roll_initialize	Verified	Lines of code with status Verified :	2
		Lines of code with status Partially processed :	0
		Lines of code with status Unable to process :	0
		Lines of code with status Failed to verify :	0
slcidemo_roll_step	Failed to verify	Lines of code with status Verified :	20
		Lines of code with status Partially processed :	0
		Lines of code with status Unable to process :	0
		Lines of code with status Failed to verify :	2

Temporary Variable Usage Results : Verified

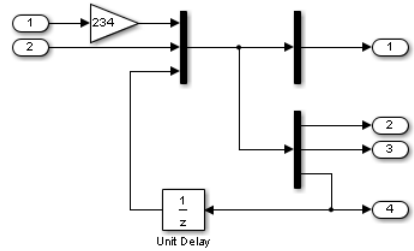
Function	Status	Details
slcidemo_roll_initialize	Verified	Function does not have any temporary variable declarations
slcidemo_roll_step	Verified	Temporary variables with status Failed to verify : 0 Temporary variables with status Verified : 4

Model Patterns That Might Result in Code Verification Failures

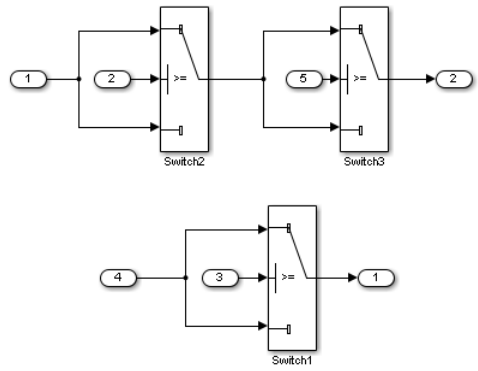
Redundant Modeling Patterns

Models with redundant modeling patterns fail code verification. When generating the code, Simulink Coder™ eliminates the redundant functionality of the block. If this elimination results in a structural change of the generated code, Simulink Code Inspector reports it as **Failed to verify**. For example:

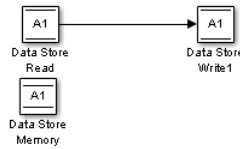
- Delay blocks with output as the source of the input. The update of the `Unit Delay` block is eliminated in the generated code.



- Switch blocks with inputs from the same local signals. The `switch 3` block is eliminated in the generated code. Switch blocks `switch 1` and `switch 2` are not eliminated because they connect to root inports.



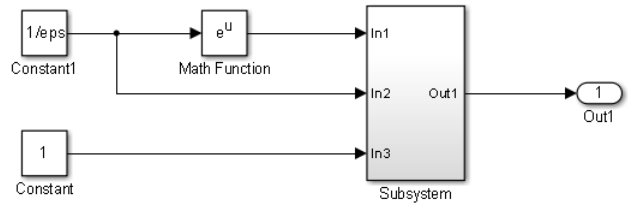
- Data store read block connected to data store write block. Both the `Data Store Read` and `Data Store Write1` blocks are eliminated in the generated code.



Blocks with Constant Non-Finite Outputs

If your model contains blocks with constant non-finite outputs, the model can fail code verification. The result is **Failed to verify**.

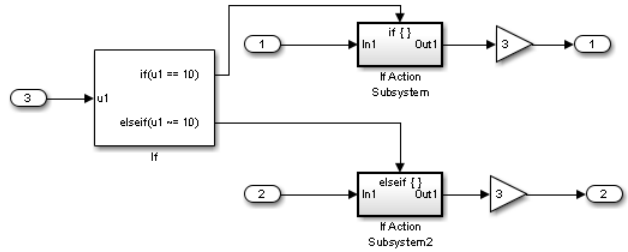
In this example, the `Math Function` block is fixed at infinity. Although the Simulink Coder generates code for the model, code verification fails.



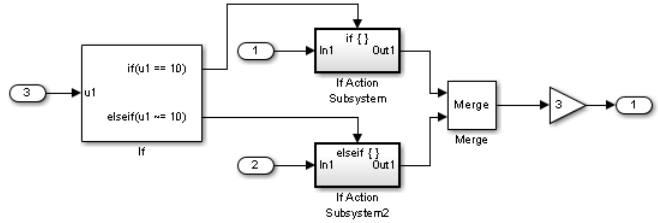
Action Subsystems with Output Not Connected to Merge Blocks

If your model contains action subsystems and the outputs are not connected to a single merge block, the model can fail code verification. The result is **Failed to verify**.

In this example, there are two action subsystems, each with output connected to a gain block. Code verification fails.



To pass code verification, consider connecting the output of all action subsystems to a single merge block, as shown below. Simulink Code Inspector can then verify the model.



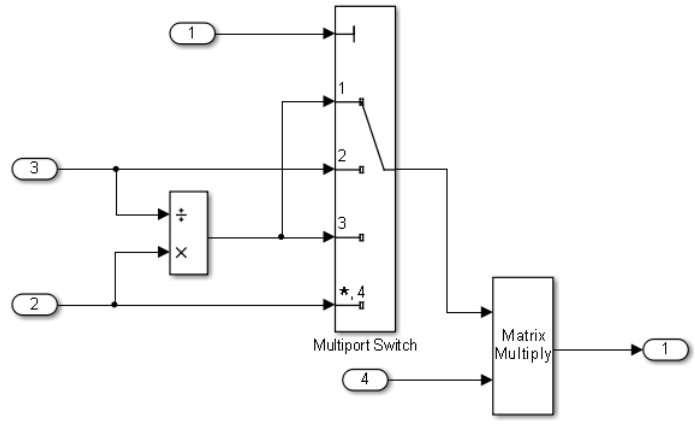
Multiport Switch Blocks with Input From Same Local Signal

Your model can fail code verification if it contains a multiport switch block with both:

- More than one input from the same local signal.
- Output to a local signal.

The generated code might have switch case statements with fall through case statements. The result is **Failed to verify**.

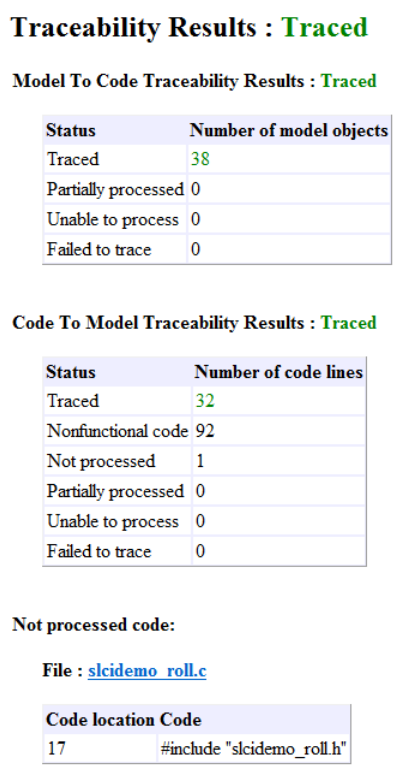
In this example, `Mulitport Switch` block input ports 1 and 3 are connected to the same local signal. Code verification fails.



Analyze Traceability Results

The detailed code inspection report for a model or submodel provides sections named **Traceability Results** and **Traceability Details**, which provide summary and detailed reports on bidirectional model-to-code mapping. The traceability report documents the code lines that implement a particular model element and the model elements that contributed to the generation of a line of code.

The following sample report excerpt shows summary traceability results for generated code that is structurally equivalent to its corresponding model.



Traceability Results : Traced

Model To Code Traceability Results : Traced

Status	Number of model objects
Traced	38
Partially processed	0
Unable to process	0
Failed to trace	0

Code To Model Traceability Results : Traced

Status	Number of code lines
Traced	32
Nonfunctional code	92
Not processed	1
Partially processed	0
Unable to process	0
Failed to trace	0

Not processed code:

File : slcidemo_roll.c

Code location	Code
17	#include "slcidemo_roll.h"

The **Traceability Results** section displays:

- The overall traceability result value aggregated from the traceability report subsections. Possible values for **Traceability Results** are **Traced**, **Partially traced**, or **Failed to trace**.
- Subsection-level traceability results:
  - **Model To Code Traceability Results**
  - **Code To Model Traceability Results**

  The subsection-level traceability result values are aggregated from the traceability status of every *object* (model element or code line) in the subsection. Each subsection returns the value **Traced**, **Partially traced**, or **Failed to trace**.

- Object level traceability results. Each model element or code line within a subsection returns a status value:
  - For model elements: **Traced**, **Partially processed**, **Unable to process**, or **Failed to trace**.
  - For code lines: **Traced**, **Partially processed**, **Unable to process**, **Failed to trace**, **Nonfunctional**, or **Not processed**.
- Not processed code — a list of C code lines that were not processed for code-to-model traceability.

Note

- **Nonfunctional** code includes empty lines, code lines that contain only comments, and opening and closing brackets ({ and }).
- **Not processed** code is code that is located outside the scope of verified functions. For example, an `#include` statement that is located outside of the scope of a model step or initialize function is not processed for code-to-model traceability.
- **Unable to process** code is code that is either of the following:
  - Does not match with any model objects.
  - One of these tokens: `';'`, `')'`, `'('`, `'['`, `']'`, or a type identifier (for example, real_T).

When the object-level traceability status is aggregated to produce the subsection-level status value, the most severe object status becomes the subsection status.

Most Severe Object-Level Status	Resulting Subsection-Level Status
Failed to trace	**Failed to trace**
Partially processed or **Unable to process**	**Partially traced**
Traced, **Nonfunctional**, or **Not processed**	**Traced**

Correspondingly, when the section-level traceability status is aggregated to produce the **Traceability Results** value, the most severe subsection status becomes the overall traceability status.

Most Severe Subsection-Level Status	Resulting Overall Traceability Status
Failed to trace	**Failed to trace**
Partially traced	**Partially traced**
Traced	**Traced**

The following sample report excerpt illustrates how traceability status is aggregated when one or more objects fails to trace.

Traceability Results : Failed to trace

Model To Code Traceability Results : Failed to trace

Status	Number of model objects
Traced	36
Partially processed	0
Unable to process	0
Failed to trace	2

Code To Model Traceability Results : Failed to trace

Status	Number of code lines
Traced	30
Nonfunctional code	92
Not processed	1
Partially processed	0
Unable to process	0
Failed to trace	2

Not processed code:

File : slcidemo_roll.c

Code location	Code
17	#include "slcidemo_roll.h"

Traceability Matrices

In this section...

"Traceability Matrices Basics" on page 3-24

"Prerequisites for Generating a Traceability Matrix" on page 3-25

"Generate a Traceability Matrix" on page 3-26

"Add Comments to a Traceability Matrix" on page 3-26

"Retain Comments When Regenerating a Traceability Matrix" on page 3-27

"Traceability Matrix Limitations" on page 3-27

Traceability Matrices Basics

When you use Model-Based Design and production code generation to develop application software components, you can generate a *traceability matrix*. The traceability matrix provides traceability among model objects, generated code, and model requirements. You can add comments to the generated traceability matrix. If you change the model and regenerate the traceability matrix, the software retains your comments.

For a given model, the generated traceability matrix can provide information about:

- Model objects that are traceable between the model and generated code, such as Simulink blocks, Stateflow® objects, and MATLAB functions.
- Model objects that are untraceable between the model and generated code, such as eliminated and virtual blocks.
- Requirements documents that you link to model objects using the Simulink Verification and Validation™ Requirements Management Interface (RMI).

Generate the traceability matrix using the `slci.ExportTraceReport` function from the MATLAB Command Window. The function creates an XLS file that contains the following worksheets:

- **Model Information** — Summary of the model configuration and checksum. The summary includes the model name, version, author,

creation date, last saved by, last updated date, checksum, and the selection of **Traceability Report Contents** parameters.

- **Code Interface** — Information about the generated code interface, such as function prototype and timing information for the model initialize and step functions.
- **Code Files** — File folders and names of the generated code files.
- **Report** — Traceability information for each model object, including model, generated code, and requirements. Each row in the worksheet pertains to a single occurrence of a model object. The information for a model object is in more than one row if the object:
  - Appears more than once in the generated code.
  - Links to more than one requirement.

Prerequisites for Generating a Traceability Matrix

Before generating a traceability matrix for model objects, generated code, and model requirements, perform the following steps:

1. Optionally, attach requirements documents. For more information, see "Workflows for Creating Links Using Selection-Based Linking" in the Simulink Verification and Validation documentation.
2. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select:
   a. **"Create code generation report"**
   b. At least one of the following **Traceability Report Contents** parameters:
      - **"Eliminated / virtual blocks"**
      - **"Traceable Simulink blocks"**
      - **"Traceable Stateflow objects"**
      - **"Traceable MATLAB functions"**
3. Generate and inspect code for the model.

Generate a Traceability Matrix

To generate a traceability matrix:

1. Open the model if it is not already open.
2. Verify that you have completed the "Prerequisites for Generating a Traceability Matrix" on page 3-25. This includes selecting report options and generating and inspecting model code.
3. In the MATLAB Command Window, enter the following command, where *`model_name`* is the name of the model and *`file_name`* is the name of the XLS file to be created:

   ```
   slci.ExportTraceReport('model_name','file_name')
   ```

   For example:

   ```
   slci.ExportTraceReport('slcidemo_roll','slcidemo_roll_tracereport')
   ```

   The software generates the traceability matrix.
4. Open the traceability matrix file, review the traceability matrix, and add comments in new columns. For more information, see "Add Comments to a Traceability Matrix" on page 3-26.

Add Comments to a Traceability Matrix

You can add comments to the traceability matrix that you generated using the `slci.ExportTraceReport` function.

To add comments to the traceability matrix, you must:

- Create new columns for your comments.
- Use unique column headings. Columns that you add must have headings.
- Add at least one entry to the column, other than the column heading.
- Retain the following columns:
  - Model Object Name
  - Model Object Path

- Model Object Subsystem
- Code File Location
- Code File Name
- Code Function
- Code Line Number
- Model Object Unique ID
- Model Object Optimized
- Code Comment Checksum

Note Comments must resolve to a text string. For example, a link to an image resolves to a text string, but a copy of the image does not.

Retain Comments When Regenerating a Traceability Matrix

To regenerate a traceability matrix and retain your comments:

1. Navigate to the working folder of the model.
2. Optionally, regenerate and reinspect code for your model. Regenerating and reinspecting code before generating the traceability matrix ensures that you have the latest model-to-code traceability information.
3. In the MATLAB Command Window, enter the following command. *`file_name`* is the name of the existing traceability matrix that you are regenerating. If the existing traceability matrix is in a different folder, include the full path to that folder in *`path`*.

   ```
   slci.ExportTraceReport('model_name', 'file_name', 'path')
   ```

   The traceability matrix regenerates.

Traceability Matrix Limitations

The traceability matrix generation capability has the following limitations:

- Does not support generating a traceability matrix for referenced models. When you generate a traceability matrix for a model that contains referenced models, the traceability matrix contains information about the Model block only. The traceability matrix does not contain information about the contents of the referenced model. If your model contains referenced models, generate a traceability matrix for the top-level model and each referenced model separately.
- Works with the Microsoft® Windows® platform only.
- In most cases, identifies comments that you add to the traceability matrix, but when comments cannot be identified, the traceability matrix includes the text:

  Row is not unique: *`comment`*

- Does not support information stored in external `.req` files. For example, when you generate a traceability matrix for a model with externally stored requirements information, the traceability matrix does not include the requirements information.

4

DO-178C Objectives Compliance

- "Model-Based Design Workflow in DO-178C" on page 4-2
- "Applicable DO-178C Objectives" on page 4-5

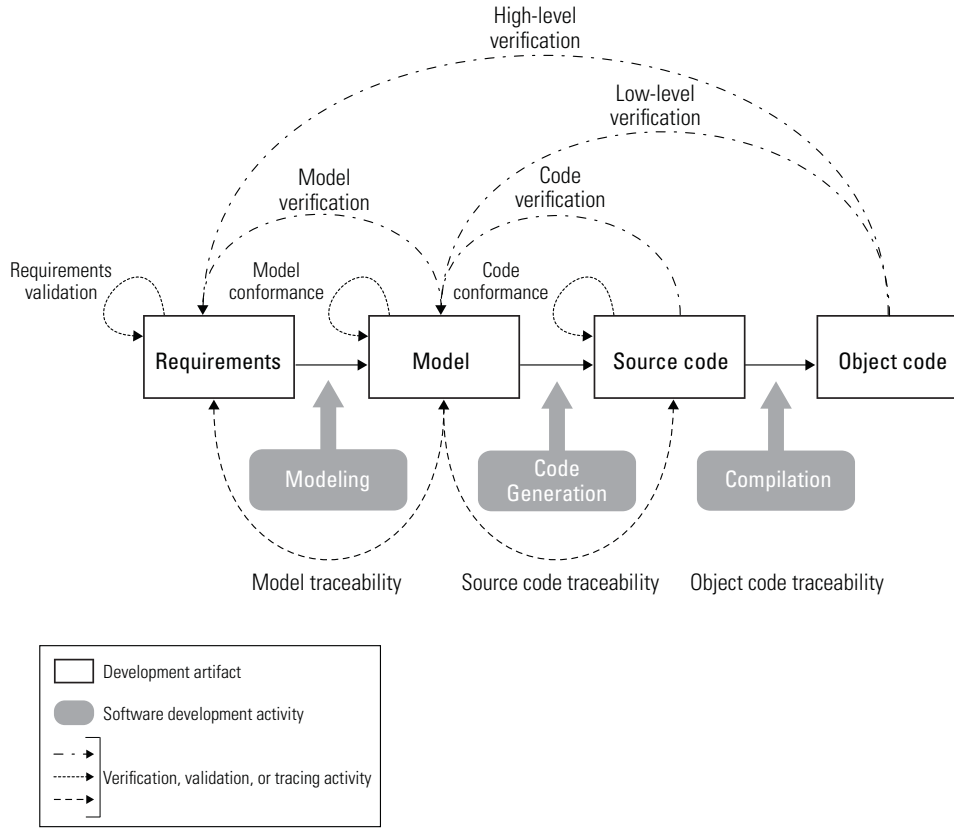
Model-Based Design Workflow in DO-178C

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, is such a standard.

MathWorks provides a DO Qualification Kit product that supports you in qualifying MathWorks verification tools for projects based on the DO-178C standard. The kit also provides detailed information on how to apply Model-Based Design to DO-178C. For more information, see http://www.mathworks.com/products/do-178/.

The DO-178C software life cycle consists of objectives that must be met for each of the life cycle stages. In Annex A of the DO-178C standard, these objectives are summarized in tables. The DO Qualification Kit document *Model-Based Design Workflow for DO-178C* summarizes those tables and provides recommendations on meeting the objectives using a Model-Based Design process.

The following diagram shows a Model-Based Design workflow that addresses the development and verification activities in a software life cycle, as described by the DO-178C standard.



The following table summarizes how Simulink Code Inspector and other MathWorks products and capabilities can be used in each step of the workflow.

Workflow Activity	Available Products and Capabilities for Model-Based Design
Requirements validation	Manual review
Modeling	Simulink, Stateflow
Model traceability	Simulink Verification and Validation — Requirements Management Interface (RMI), Simulink Report Generator™ — System Design Description report*
Model conformance	Simulink Verification and Validation — DO-178C/DO-331 checks*

Workflow Activity	Available Products and Capabilities for Model-Based Design
Model verification	SystemTest™ — Limit Check element*, Simulink Design Verifier™ — Property Proving (optional), Simulink Design Verifier — Design Error Detection (optional), Simulink Verification and Validation — Model Coverage*, Simulink Report Generator — System Design Description report*
Code generation	Embedded Coder
Source code traceability	Simulink Code Inspector — Traceability Report*
Code conformance	Polyspace® Products for C/C++ — MISRA AC AGC checks*
Code verification	Simulink Code Inspector — Code Verification Report*
Compilation	Third-party IDE or compiler
Low-level verification	SystemTest — Limit Check element*, Simulink Design Verifier — Test Generation, Embedded Coder — PIL test, Embedded Coder — Code coverage tool link (requires third-party code coverage tool), Polyspace Products for C/C++*
High-level verification	SystemTest — Limit Check element*, Embedded Coder — PIL test, Embedded Coder — Code coverage tool link (requires third-party code coverage tool), Polyspace Products for C/C++*
Object code traceability (Level A only)	Embedded Coder — Code generation report, Third-party IDE or compiler — Object code listing
*The DO Qualification Kit product may be used to support DO-178C tool qualification.	

Applicable DO-178C Objectives

The following table summarizes anticipated certification credits for Simulink Code Inspector, when used with other code verification products.

Annex A Table	Objectives	DO-178C Reference	Software Levels	Anticipated Certification Credit
A-5	(1) Source code complies with low-level requirements	Section 6.3.4.a	A, B, C	Full — Simulink Code Inspector
A-5	(2) Source code complies with software architecture	Section 6.3.4.b	A, B, C	Full — Simulink Code Inspector
A-5	(3) Source code is verifiable	Section 6.3.4.c	A, B	Full — Simulink Code Inspector
A-5	(4) Source code conforms to standards	Section 6.3.4.d	A, B, C	Full — Polyspace MISRA-AC ACG rules checker
A-5	(5) Source code is traceable to low-level requirements	Section 6.3.4.e	A, B, C	Full — Simulink Code Inspector
A-5	(6) Source code is accurate and consistent	Section 6.3.4.f	A, B, C	Partial — Simulink Code Inspector can detect uninitialized or unused variables or constants in the generated C code. Polyspace can detect overflows and data corruption in the source code. Other issues, such as stack usage, resource contention, worst case execution time, and exception handling must be assessed by other means.