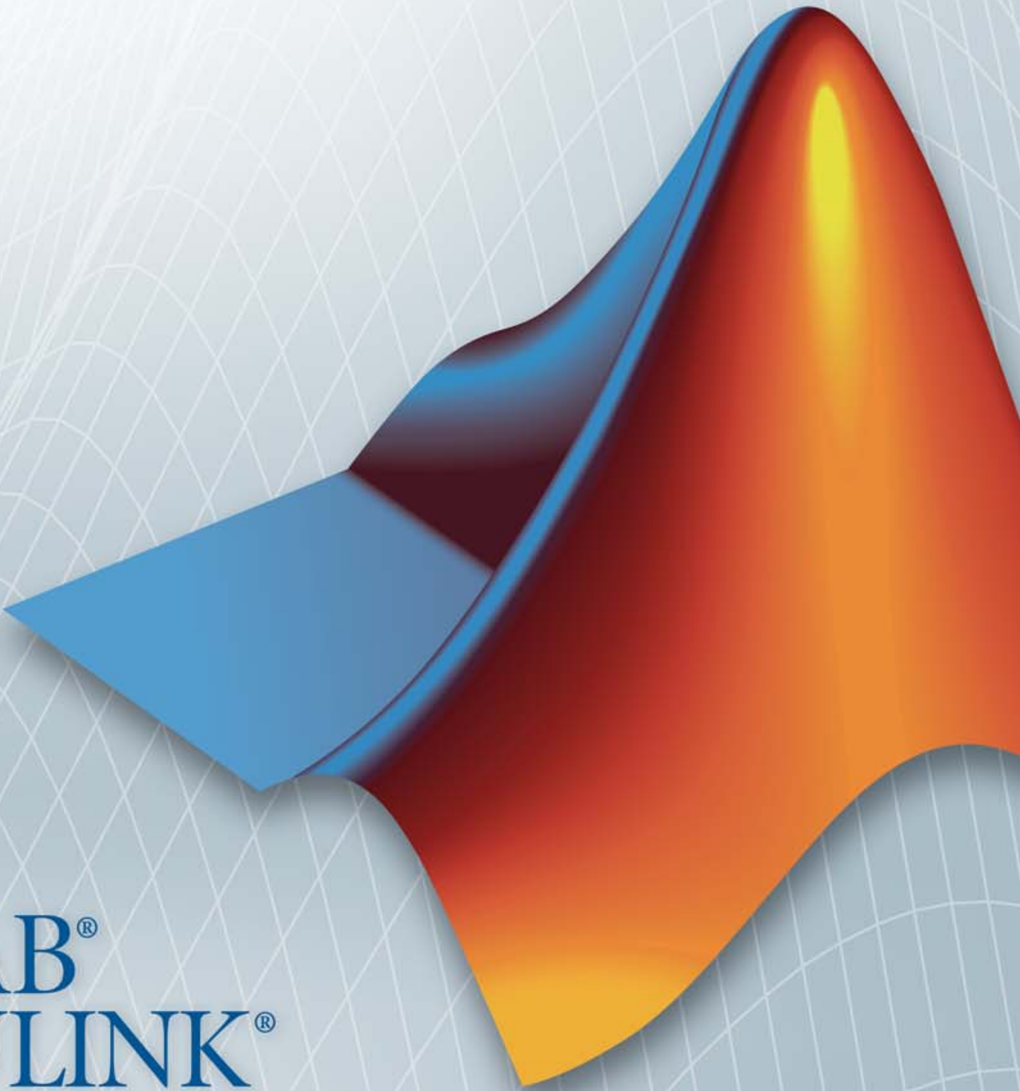
Simulink® Code Inspector™
User's Guide

R2011b



MATLAB®
&SIMULINK®

MathWorks®

Simulink® Code Inspector™ User's Guide

Revision History

September 2011	Online only	New for Version 1.0 (Release 2011b)

Contents

Getting Started

Why Use This Product?

Simulink® Code Inspector™ automatically compares generated code with its source model to satisfy code-review objectives in DO-178B 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.

Key features of Simulink Code Inspector include:

- 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

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 successful inspection results.

For company and organization Designated Engineering Representatives (DERs) and Federal Aviation Administration (FAA) Auditors who must certify software under DO178-B, 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

Simulink Code Inspector automatically compares generated code with its source model to satisfy code-review objectives in DO-178B 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 too 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 “Model Configuration Constraints Reference”, “Block Constraints Reference”, 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 no 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`.	Partial***

*** The data types of root output blocks are not verified in all cases.

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 port 1 and 2 (in this order). The multiplication code for the Gain block uses the variable representing the input port to calculate the output value for the first element of its output buffer.	Full
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	All 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

Typical Workflows

In this section...

Workflow Summary

The workflow that you use depends on your role.

If Your Role Is...	Consider...
Engineer responsible for developing a model that requires code inspection	Model development workflows
Code reviewer	Inspection workflow

Model Development Workflows

- "New Model Development Workflow" on page 1-9
- "Model Enhancement and Maintenance Workflow" on page 1-10

New Model Development Workflow

If you are developing a new application model, iterate through the following steps, incrementally combining application model components that are compatible with the Code Inspector until your application model is complete.

1. Create a model. For details on how to check compatibility while developing models, see Chapter 2, "Model Compatibility Checking".
2. Check whether the model is compatible with Code Inspector rules. If the model is compatible, go to step 4.
3. Fix reported conditions that you did not expect and return to step 2.
4. Consider combining the model with one or more other compatible models.

For details on how to check compatibility while developing models, see Chapter 2, "Model Compatibility Checking".

Model Enhancement and Maintenance Workflow

If you are enhancing or maintaining an application model, for each model that you modify, iterate through the following steps. For a referenced model, iterate through the steps for each model in the relevant model hierarchy, starting from the bottom of the hierarchy.

1. Check whether the model is compatible with Code Inspector rules. If the model is compatible, you are done.

2. Fix reported conditions that you did not expect and return to step 1.

For details on how to check compatibility while developing models, see Chapter 2, "Model Compatibility Checking".

Inspection Workflow

1. Enable the Code Inspector by setting the model configuration parameter `AdvancedOptControl` to `-SLCI`..

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

2. Check whether you must save the model. If you make changes and do not save the model before initiating an inspection, the inspection does not start. You can use the model parameter, `Dirty`, to determine whether you must save a model. For example:

   ```
   if strcmp(get_param(gcs, 'Dirty'), 'on')
       save_system;
   end
   ```

3. Check the model for compatibility with code inspection rules. See Chapter 2, "Model Compatibility Checking".

4. Run the Code Inspector.

If You Want To...	Use the...
Review the verification reports displayed at the end of inspection for a model	Simulink Code Inspector dialog box
Check the overall inspection status (VERIFIED or FAILED_TO_VERIFY)	Command line interface
Run multiple inspections in batch mode	Command line interface

5. Review the results. If modifications are required, return to model development and follow the workflow described in "Model Enhancement and Maintenance Workflow" on page 1-10. If the code inspection passes or if conditions reported are intentional and expected, continue to step 6.

6. Depending on your role, archive the report, pass the report on for further review, include the report in a certification package, and so on.

For details on how to run a code inspection, see Chapter 3, "Code Inspection".

Quick-Start Example

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 demo `slcidemo_intro` illustrates the same code verification workflow using MATLAB commands.

---

1. Set up the model in a work folder and Open the demo model `slcidemo_roll_orig` using the following command:

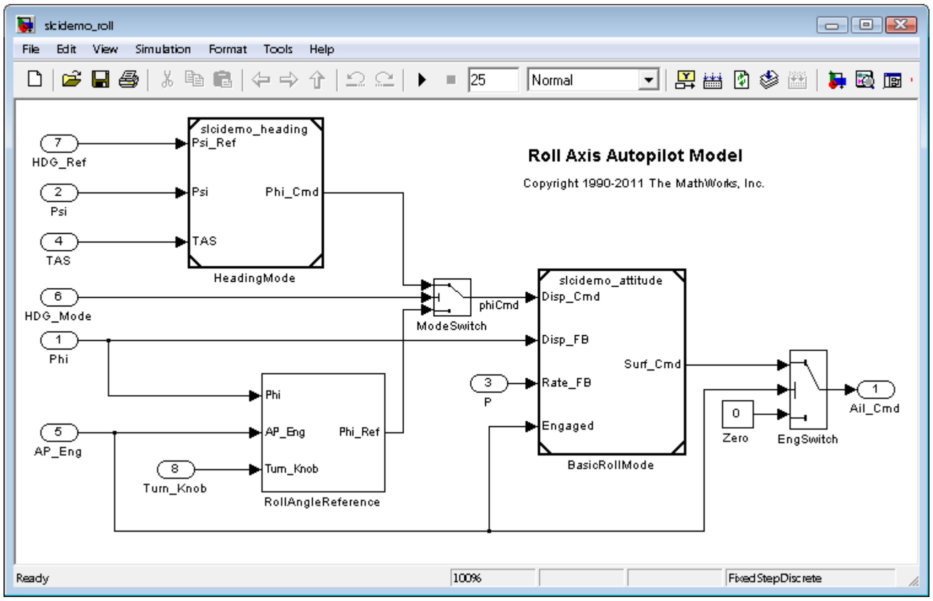
   ```
   >> slcidemo_roll_orig
   ```

   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. From the top model window, issue the following command:

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

   ---

2. Save a copy of the model to a work folder, renaming it to `slcidemo_roll`. Change the folder to the work folder. The top level of the model appears as follows.



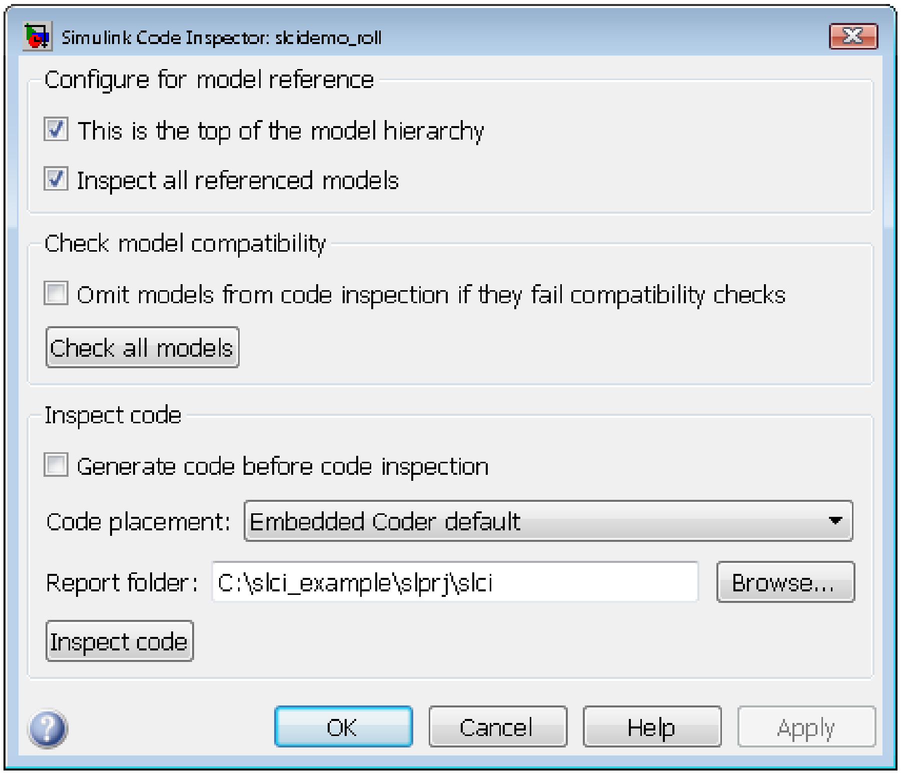
This model represents a basic roll axis autopilot with two operating modes: roll attitude hold and heading hold. The mode logic for these modes is external to this model. The model architecture represents the heading hold mode and basic roll attitude function as referenced models. The model includes:

- 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 that you want.
- Model block `BasicRollMode`, referencing a separate model that computes the roll attitude control function.

3 Prepare the model for code generation and code inspection.

a From the top model window, select **Tools > 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 appeara as follows:



Simulink Code Inspector: slcidemo_roll

Configure for model reference

[✓] This is the top of the model hierarchy

[✓] Inspect all referenced models

Check model compatibility

[ ] Omit models from code inspection if they fail compatibility checks

Inspect code

[ ] Generate code before code inspection

Code placement: Embedded Coder default

Report folder: C:\slci_example\slprj\slci

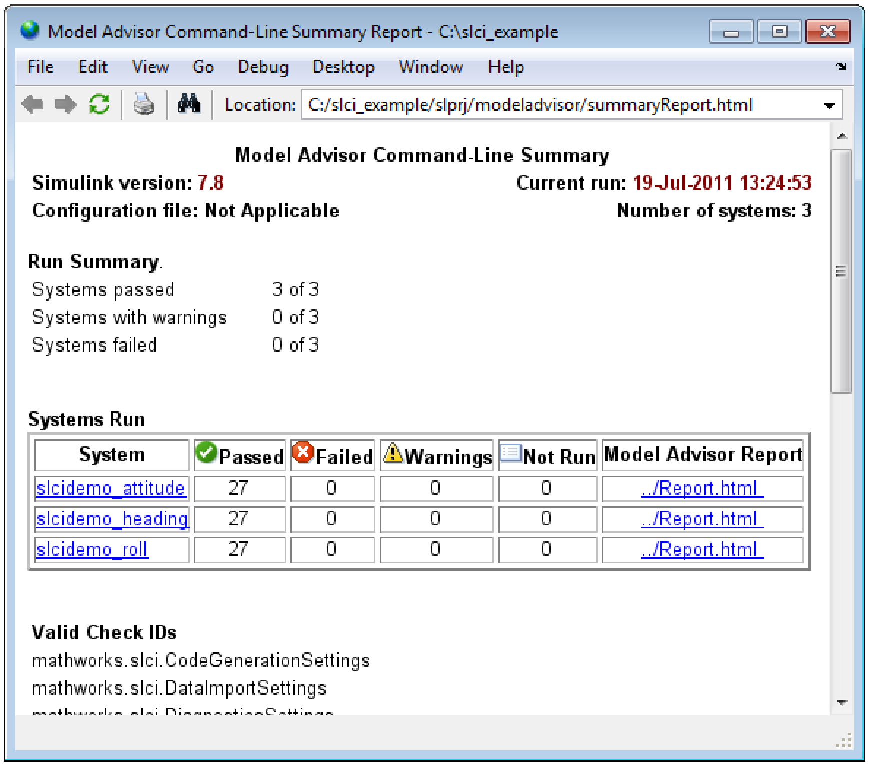
c Run the model compatibility checks by clicking **Check all models**. The compatibility checker displays a progress bar.

Results appear in the command window and in an HTML 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 report window displays results similar to the following.



Model Advisor Command-Line Summary Report - C:\slci_example

Location: C:/slci_example/slprj/modeladvisor/summaryReport.html

Model Advisor Command-Line Summary

Simulink version: 7.8 **Current run:** 19-Jul-2011 13:24:53

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	27	0	0	0	../Report.html
slcidemo_heading	27	0	0	0	../Report.html
slcidemo_roll	27	0	0	0	../Report.html

Valid Check IDs

mathworks.slci.CodeGenerationSettings
mathworks.slci.DataImportSettings

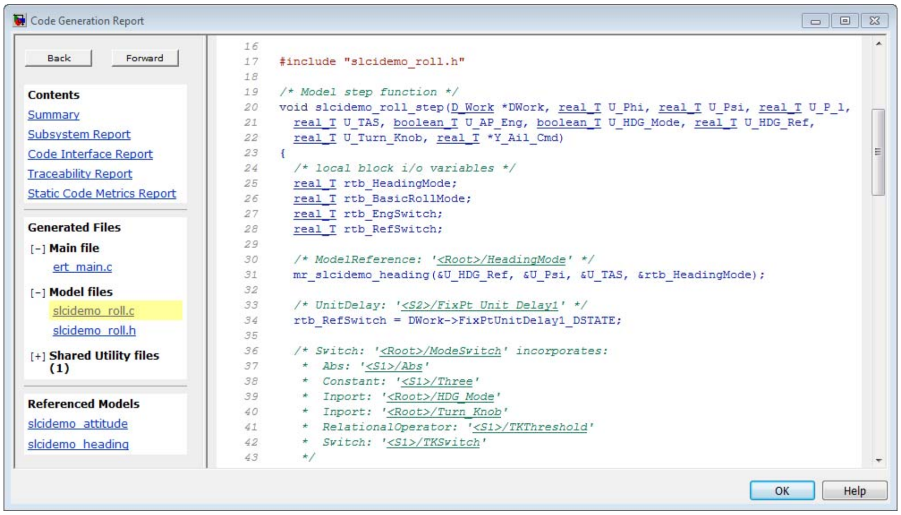
Note This HTML report is linked from the command window results. It is saved as file `summaryReport.html` in the current working folder.

4 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 > Configuration Parameters** to open the Configuration Parameters dialog box. In the **Code Generation > Report** pane, select the option **Launch report automatically**. (If you try this example with a model other than `slcidemo_roll`, 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.

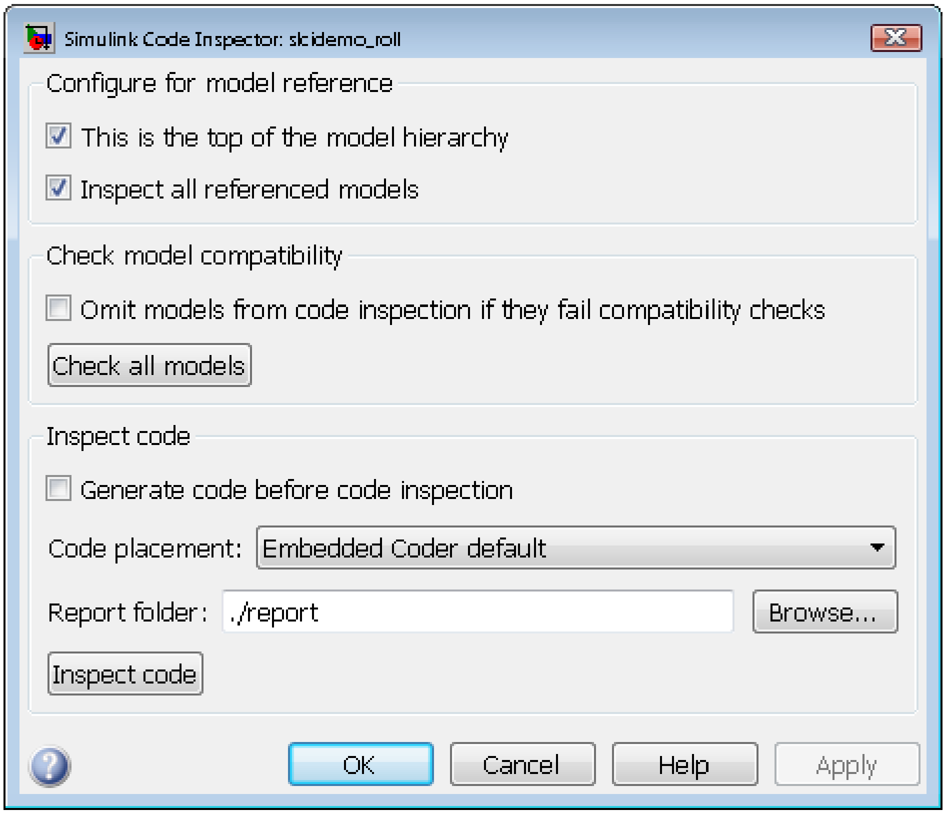


```c

#include "slcidemo_roll.h"

/* Model step function */
void slcidemo_roll_step(D_Work *DWork, real_T U_Phi, real_T U_Psi, real_T U_P_1,
  real_T U_TAS, boolean_T U_AP_Eng, boolean_T U_HDG_Mode, real_T U_HDG_Ref,
  real_T U_Turn_Knob, real_T *Y_Ail_Cmd)
{
  /* local block i/o variables */
  real_T rtb_HeadingMode;
  real_T rtb_BasicRollMode;
  real_T rtb_EngSwitch;
  real_T rtb_RefSwitch;

  /* ModelReference: '<Root>/HeadingMode' */
  mr_slcidemo_heading(&U_HDG_Ref, &U_Psi, &U_TAS, &rtb_HeadingMode);

  /* UnitDelay: '<S2>/FixPt Unit Delay1' */
  rtb_RefSwitch = DWork->FixPtUnitDelay1_DSTATE;

  /* Switch: '<Root>/ModeSwitch' incorporates:
   *  Abs: '<S1>/Abs'
   *  Constant: '<S1>/Three'
   *  Inport: '<Root>/HDG_Mode'
   *  Inport: '<Root>/Turn_Knob'
   *  RelationalOperator: '<S1>/TKThreshold'
   *  Switch: '<S1>/TKSwitch'
   */
```

5 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**.



Simulink Code Inspector: slcidemo_roll

Configure for model reference

- [✓] This is the top of the model hierarchy
- [✓] Inspect all referenced models

Check model compatibility

- [ ] Omit models from code inspection if they fail compatibility checks

Inspect code

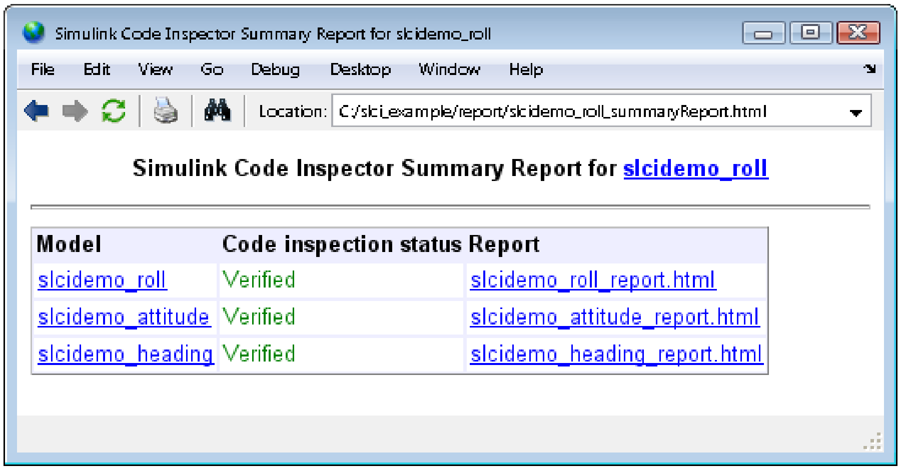
- [ ] Generate code before code inspection

Code placement: Embedded Coder default

Report folder: ./report

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

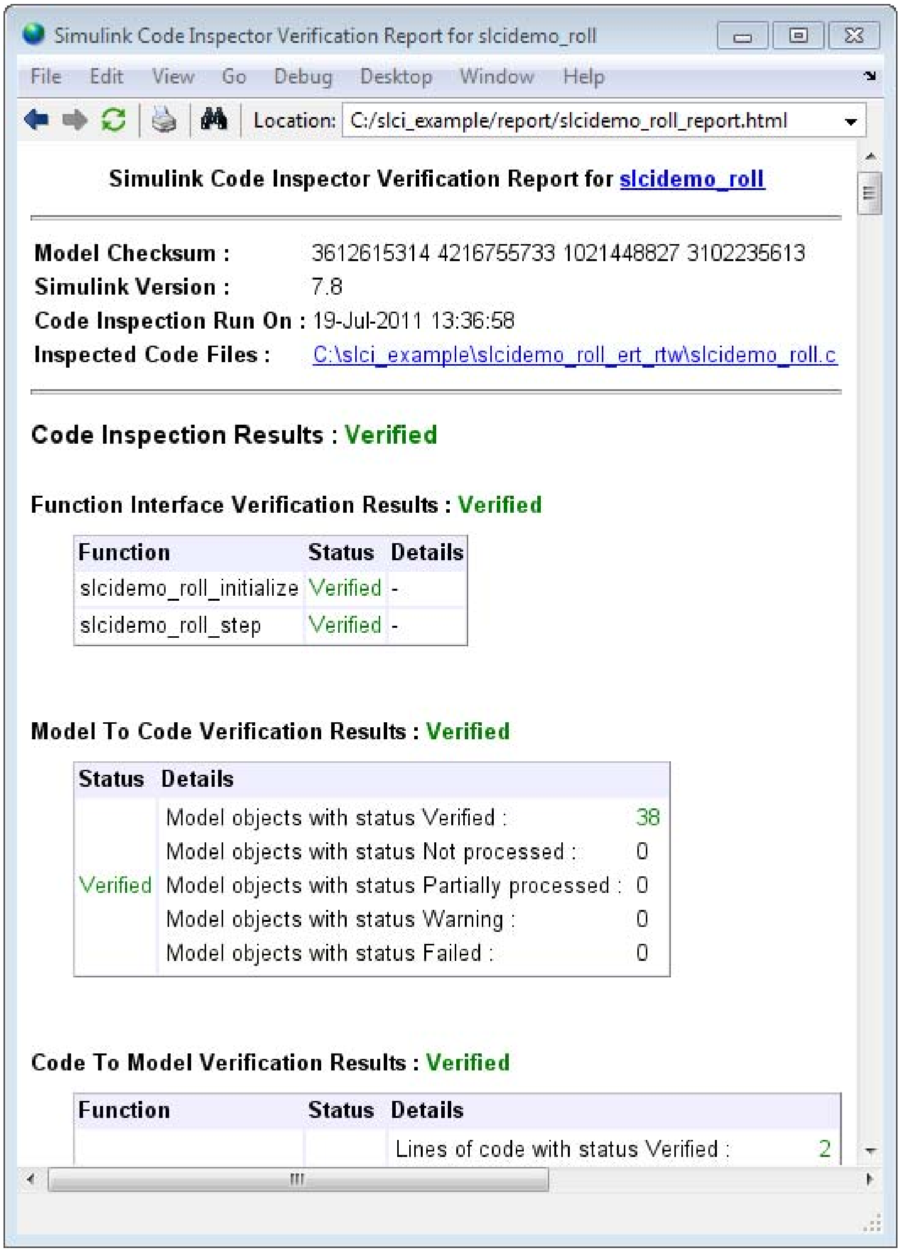
d The Code Inspector displays a summary in an HTML report window.


Simulink Code Inspector Summary Report for slcidemo_roll
File Edit View Go Debug Desktop Window Help
Location: C:/slci_example/report/slcidemo_roll_summaryReport.html

Simulink Code Inspector Summary Report for slcidemo_roll

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

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


Simulink Code Inspector Verification Report for slcidemo_roll
File Edit View Go Debug Desktop Window Help
Location: C:/slci_example/report/slcidemo_roll_report.html

Simulink Code Inspector Verification Report for slcidemo_roll

Model Checksum :	3612615314 4216755733 1021448827 3102235613
Simulink Version :	7.8
Code Inspection Run On :	19-Jul-2011 13:36:58
Inspected Code Files :	C:\slci_example\slcidemo_roll_ert_rtw\slcidemo_roll.c

Code Inspection 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 Not processed :	0
	Model objects with status Partially processed :	0
	Model objects with status Warning :	0
	Model objects with status Failed :	0

Code To Model Verification Results : Verified

Function	Status	Details	
		Lines of code with status Verified :	2

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

6 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 demo 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','||','&&')
```

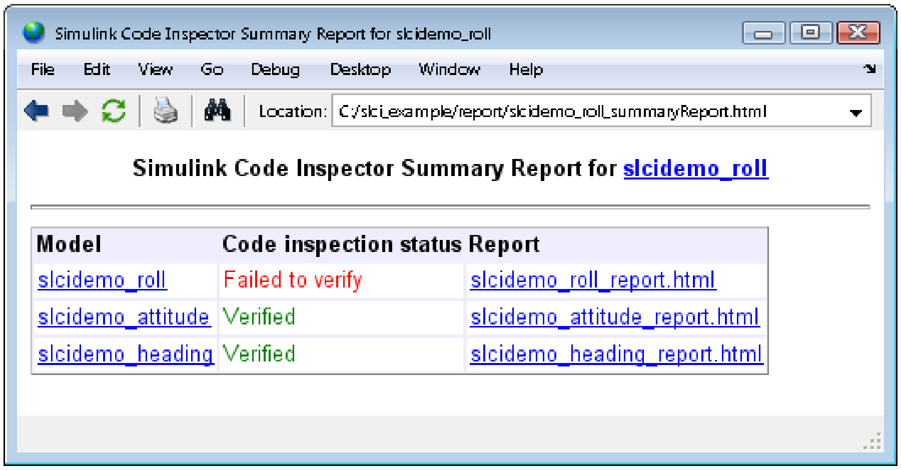
The `slcidemo_modifycode` utility function displays the following output:

```
Modified line 93 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 inspection reports.

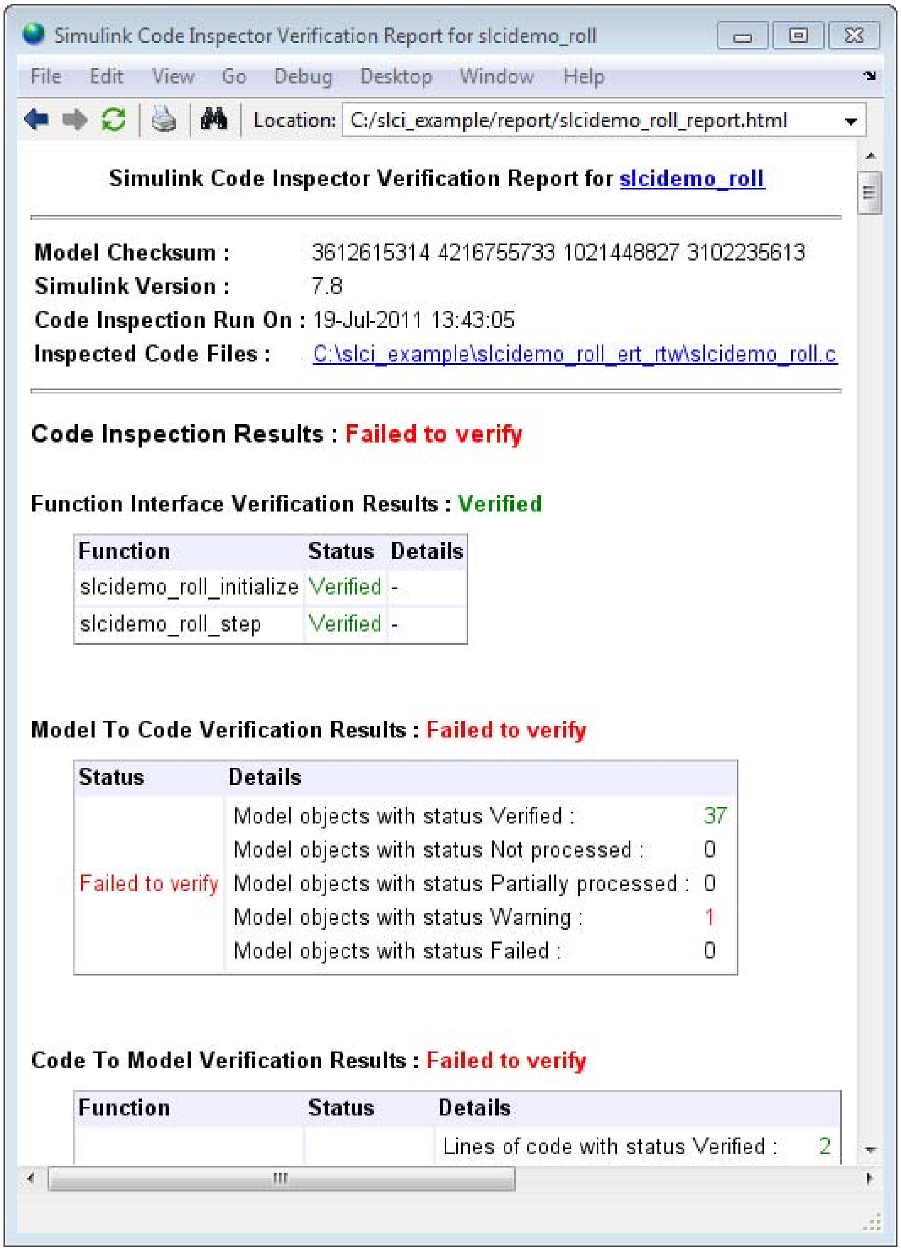
The summary report displays a failure for the top model.


Simulink Code Inspector Summary Report for slcidemo_roll
File Edit View Go Debug Desktop Window Help
Location: C:/slci_example/report/slcidemo_roll_summaryReport.html

Simulink Code Inspector Summary Report for slcidemo_roll

Model	Code inspection status	Report
slcidemo_roll	Failed to verify	slcidemo_roll_report.html
slcidemo_attitude	Verified	slcidemo_attitude_report.html
slcidemo_heading	Verified	slcidemo_heading_report.html

The verification 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 Verification Report for slcidemo_roll
File Edit View Go Debug Desktop Window Help
Location: C:/slci_example/report/slcidemo_roll_report.html

Simulink Code Inspector Verification Report for slcidemo_roll

Model Checksum :	3612615314 4216755733 1021448827 3102235613
Simulink Version :	7.8
Code Inspection Run On :	19-Jul-2011 13:43:05
Inspected Code Files :	C:\slci_example\slcidemo_roll_ert_rtw\slcidemo_roll.c

Code Inspection Results : Failed to verify

Function Interface Verification Results : Verified

Function	Status	Details
slcidemo_roll_initialize	Verified	-
slcidemo_roll_step	Verified	-

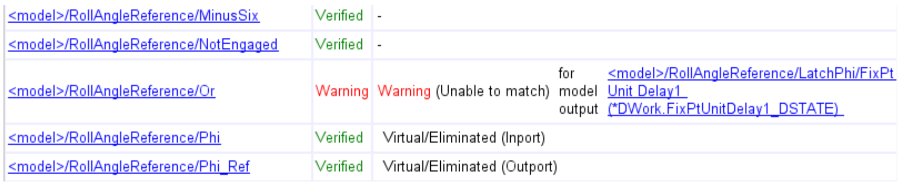
Model To Code Verification Results : Failed to verify

Status	Details	
Failed to verify	Model objects with status Verified :	37
	Model objects with status Not processed :	0
	Model objects with status Partially processed :	0
	Model objects with status Warning :	1
	Model objects with status Failed :	0

Code To Model Verification Results : Failed to verify

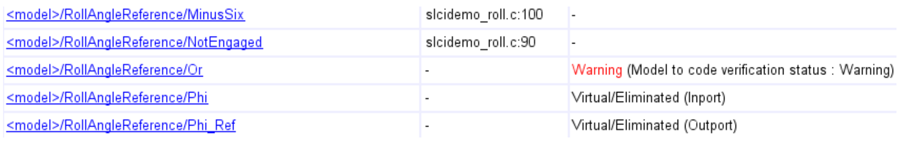
Function	Status	Details	
		Lines of code with status Verified :	2

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



<model>/RollAngleReference/MinusSix	Verified	-		
<model>/RollAngleReference/NotEngaged	Verified	-		
<model>/RollAngleReference/Or	Warning	Warning (Unable to match)	for model output	<model>/RollAngleReference/LatchPhi/FixPt Unit Delay1 (*DWork.FixPtUnitDelay1_DSTATE)
<model>/RollAngleReference/Phi	Verified	Virtual/Eliminated (Inport)		
<model>/RollAngleReference/Phi_Ref	Verified	Virtual/Eliminated (Outport)		

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



<model>/RollAngleReference/MinusSix	slcidemo_roll.c:100	-
<model>/RollAngleReference/NotEngaged	slcidemo_roll.c:90	-
<model>/RollAngleReference/Or	-	Warning (Model to code verification status : Warning)
<model>/RollAngleReference/Phi	-	Virtual/Eliminated (Inport)
<model>/RollAngleReference/Phi_Ref	-	Virtual/Eliminated (Outport)

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

Model Compatibility Checking

About Model Compatibility Checking

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 needed to achieve successful 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, 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, compatibility checks increase inspection success.

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
  - 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 “Model Configuration Constraints Reference”, “Block Constraints Reference”, and “Simulink Code Inspector Checks”.

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

- From the model window, select **Tools > Simulink Code Inspector**, and use the Simulink Code Inspector dialog box to control model compatibility checking. For more information, see “Checking Model Compatibility from the Model Window” on page 2-5.
- Use the slci.Configuration interface to programmatically control model compatibility checking. For more information, see “Checking Model Compatibility Using the Command-Line Interface” on page 2-9.

Alternatively, you can initiate model compatibility checking by opening the Model Advisor dialog box and selecting and running the Simulink Code Inspector checks.

Model compatibility checking generates a detailed HTML report for each model checked. If you checked all models in a model reference hierarchy, the software reports summary status at the MATLAB command line and displays a summary HTML report. The summary results list the number of checks that passed, failed, displayed a warning, or did not run, and provides links to the detailed HTML report for each model. 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 correction is needed, recommended actions are provided. The available model compatibility checks are listed in report order and described in the "Simulink Code Inspector Checks" reference.

Checking Model Compatibility from the Model Window

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

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

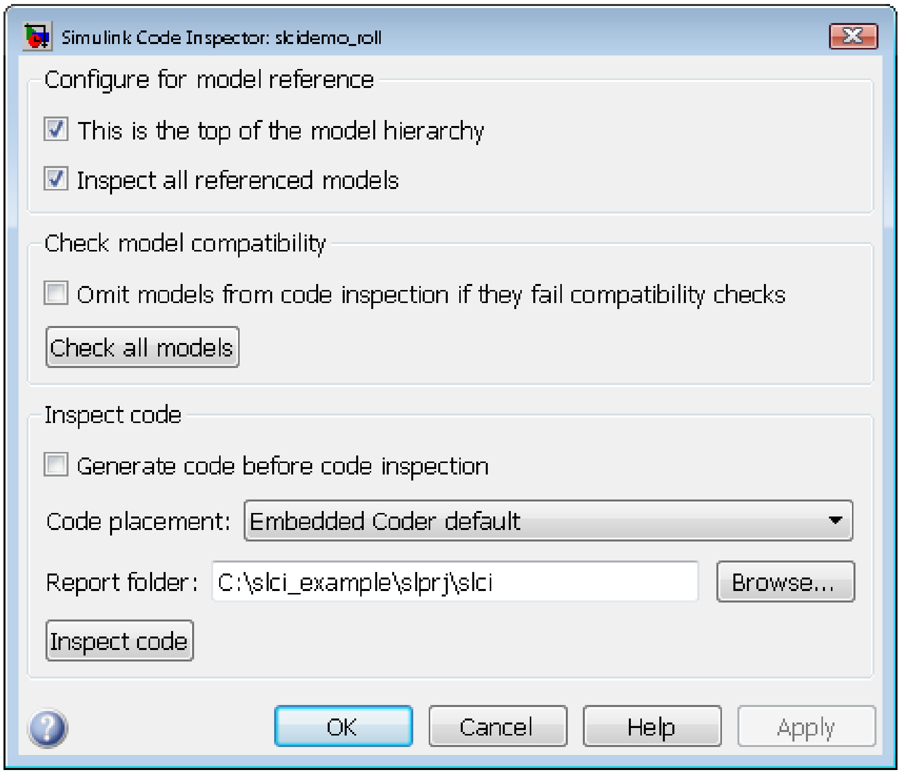
      ```
      >> slcidemo_roll_orig
      ```

   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 **Tools > 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**.


Simulink Code Inspector: slcidemo_roll
Configure for model reference
Check model compatibility
Check all models
Inspect code
Inspect code
Browse...
OK
Cancel
Help
Apply

[✓] This is the top of the model hierarchy

[✓] Inspect all referenced models

[ ] Omit models from code inspection if they fail compatibility checks

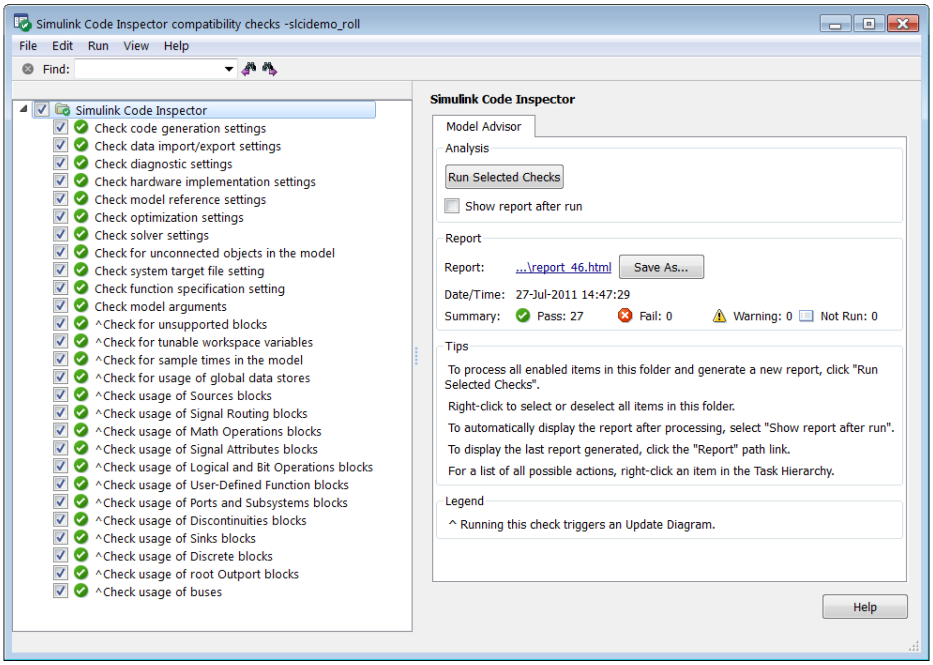
[ ] Generate code before code inspection

Code placement: Embedded Coder default

Report folder: C:\slci_example\slprj\slci

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.



Simulink Code Inspector
- ✓ Check code generation settings
- ✓ Check data import/export settings
- ✓ Check diagnostic settings
- ✓ Check hardware implementation settings
- ✓ Check model reference settings
- ✓ Check optimization settings
- ✓ Check solver settings
- ✓ Check for unconnected objects in the model
- ✓ Check system target file setting
- ✓ Check function specification setting
- ✓ Check model arguments
- ✓ ^Check for unsupported blocks
- ✓ ^Check for tunable workspace variables
- ✓ ^Check for sample times in the model
- ✓ ^Check for usage of global data stores
- ✓ ^Check usage of Sources blocks
- ✓ ^Check usage of Signal Routing blocks
- ✓ ^Check usage of Math Operations blocks
- ✓ ^Check usage of Signal Attributes blocks
- ✓ ^Check usage of Logical and Bit Operations blocks
- ✓ ^Check usage of User-Defined Function blocks
- ✓ ^Check usage of Ports and Subsystems blocks
- ✓ ^Check usage of Discontinuities blocks
- ✓ ^Check usage of Sinks blocks
- ✓ ^Check usage of Discrete blocks
- ✓ ^Check usage of root Outport blocks
- ✓ ^Check usage of buses

Simulink Code Inspector

Model Advisor

Analysis

☐ Show report after run

Report

Report: ...\report_46.html

Date/Time: 27-Jul-2011 14:47:29

Summary: ✓ Pass: 27 ✗ Fail: 0 ⚠ Warning: 0 ☐ Not Run: 0

Tips

To process all enabled items in this folder and generate a new report, click "Run Selected Checks".

Right-click to select or deselect all items in this folder.

To automatically display the report after processing, select "Show report after run".

To display the last report generated, click the "Report" path link.

For a list of all possible actions, right-click an item in the Task Hierarchy.

Legend

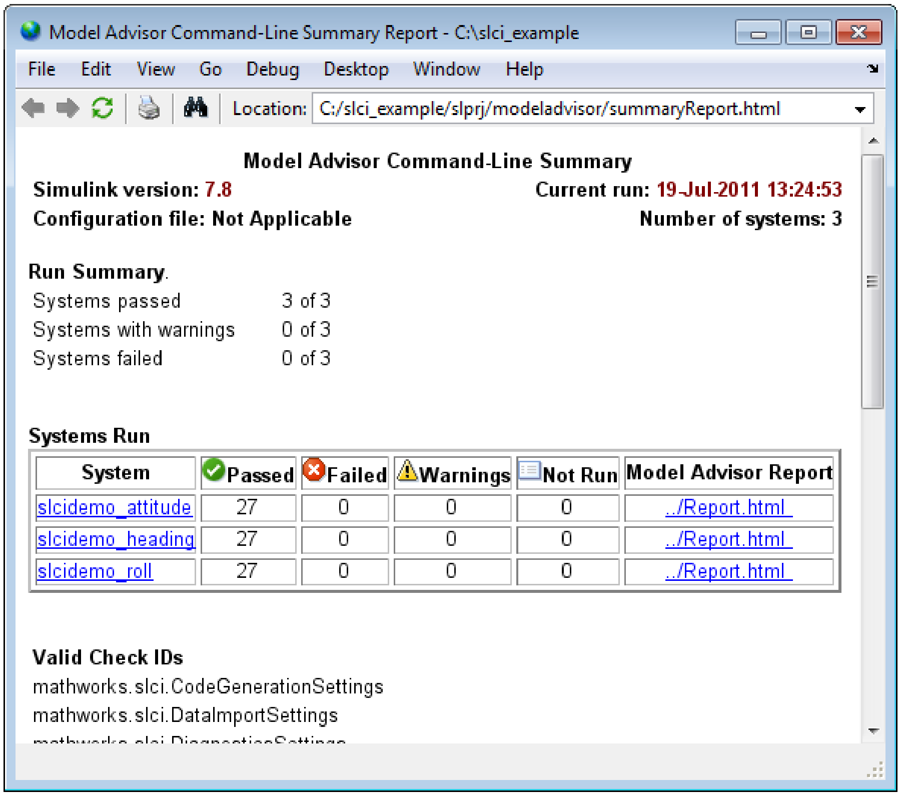
^ Running this check triggers an Update Diagram.

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 Report - C:\slci_example
File Edit View Go Debug Desktop Window Help
Location: C:/slci_example/slprj/modeladvisor/summaryReport.html

Model Advisor Command-Line Summary

Simulink version: 7.8 Current run: 19-Jul-2011 13:24:53

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	27	0	0	0	../Report.html
slcidemo_heading	27	0	0	0	../Report.html
slcidemo_roll	27	0	0	0	../Report.html

Valid Check IDs

mathworks.slci.CodeGenerationSettings

mathworks.slci.DataImportSettings

Note This HTML summary report is linked from the command window results, and is saved as file `summaryReport.html` in 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 all checks pass, the model is ready for inspection. If incompatibilities are reported, correct the issues and recheck the model for compatibility.

Checking Model Compatibility Using the Command-Line Interface

To programmatically control model compatibility checking, use the `slci.Configuration` interface. For a complete list of applicable `slci.Configuration` methods, see the "Model Compatibility Checking" category in the function reference documentation.

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 all checks pass, the model is ready for inspection. If incompatibilities are reported, correct 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 demo `slcidemo_intro`.

Correcting or Working 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).

Correcting or Working 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.

Code Inspection

About Code Inspection

Code inspection automatically compares generated code with its source model to satisfy code-review objectives in DO-178B 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 needed to achieve successful 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 Chapter 2, "Model Compatibility Checking".

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 **Tools > Simulink Code Inspector**, and use the Simulink Code Inspector dialog box to control code inspection. For more information, see "Inspecting Code from the Model Window" on page 3-4.
- Use the slci.Configuration interface to programmatically control code inspection. For more information, see "Inspecting Code Using the Command-Line Interface" on page 3-9.

Code inspection generates an HTML code verification report, which documents code inspection results with the following major sections:

- Code Inspection — Summary and detailed reports on structural equivalence between model and code elements. Categories include:
  - Function Interface Verification
  - Model To Code Verification
  - Code To Model Verification
  - Temporary Variable Usage
- Traceability — Summary and detailed reports on
  - Model To Code Traceability
  - Code To Model Traceability

For company and organization code reviewers who must certify software under DO178-B, 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 are time intensive and error prone, you can run the Code Inspector and review a detailed inspection report.

Inspecting Code from the Model Window

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

   a Open the demo 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 "Checking Model Compatibility from the Model Window" on page 2-5. When the model passes all 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 > 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 **Launch 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.

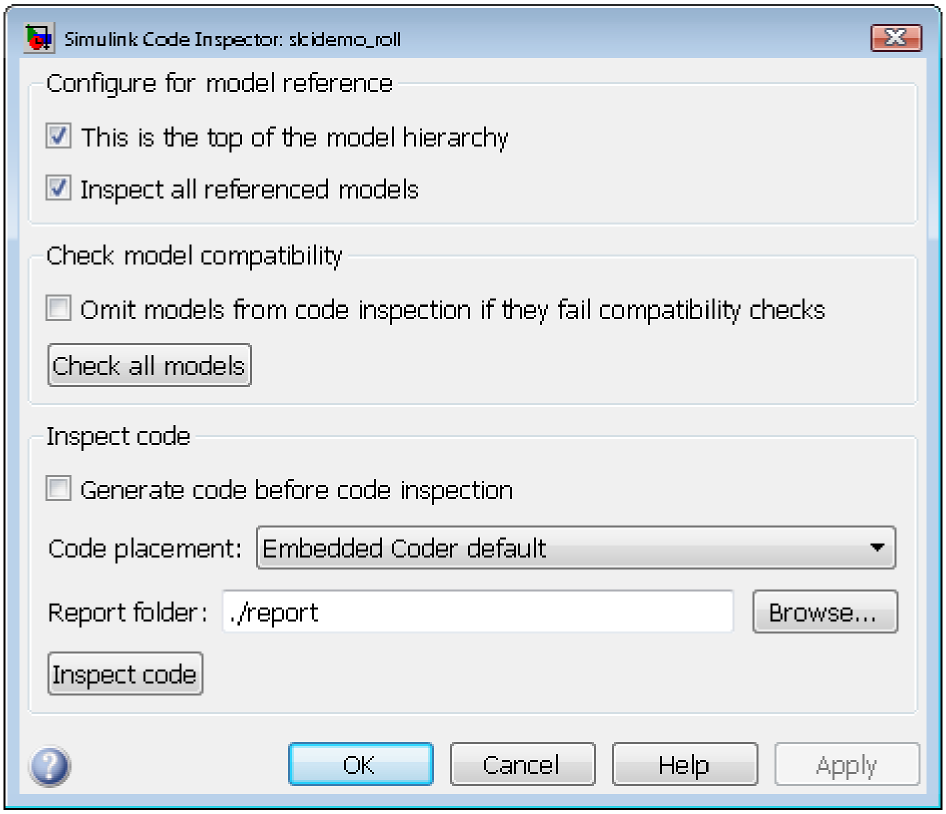
   b Go to the **Code Generation** main pane and click **Generate code**. (If the **Generate code** button does not appear for your model, select the **Generate code only** option to enable the button.) Progress is displayed in the MATLAB Command Window.

   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**.



Simulink Code Inspector: slcidemo_roll

Configure for model reference

- [✓] This is the top of the model hierarchy
- [✓] Inspect all referenced models

Check model compatibility

- [ ] Omit models from code inspection if they fail compatibility checks

Inspect code

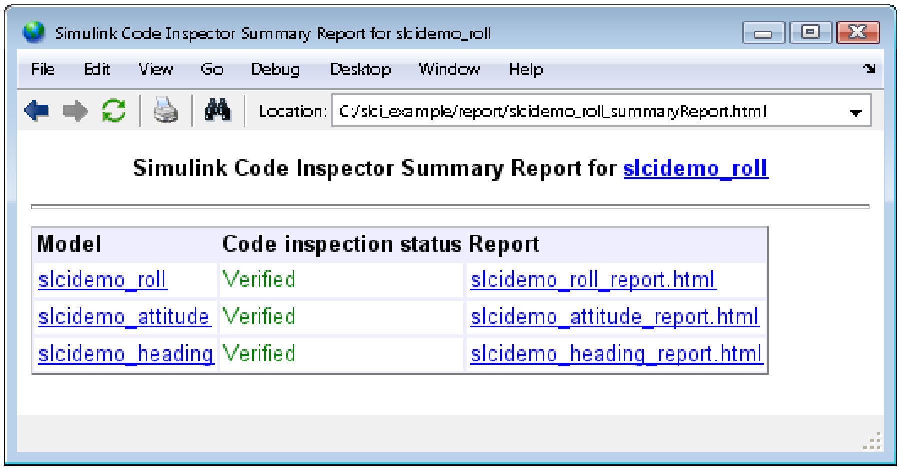
- [ ] Generate code before code inspection

Code placement: Embedded Coder default

Report folder: ./report

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

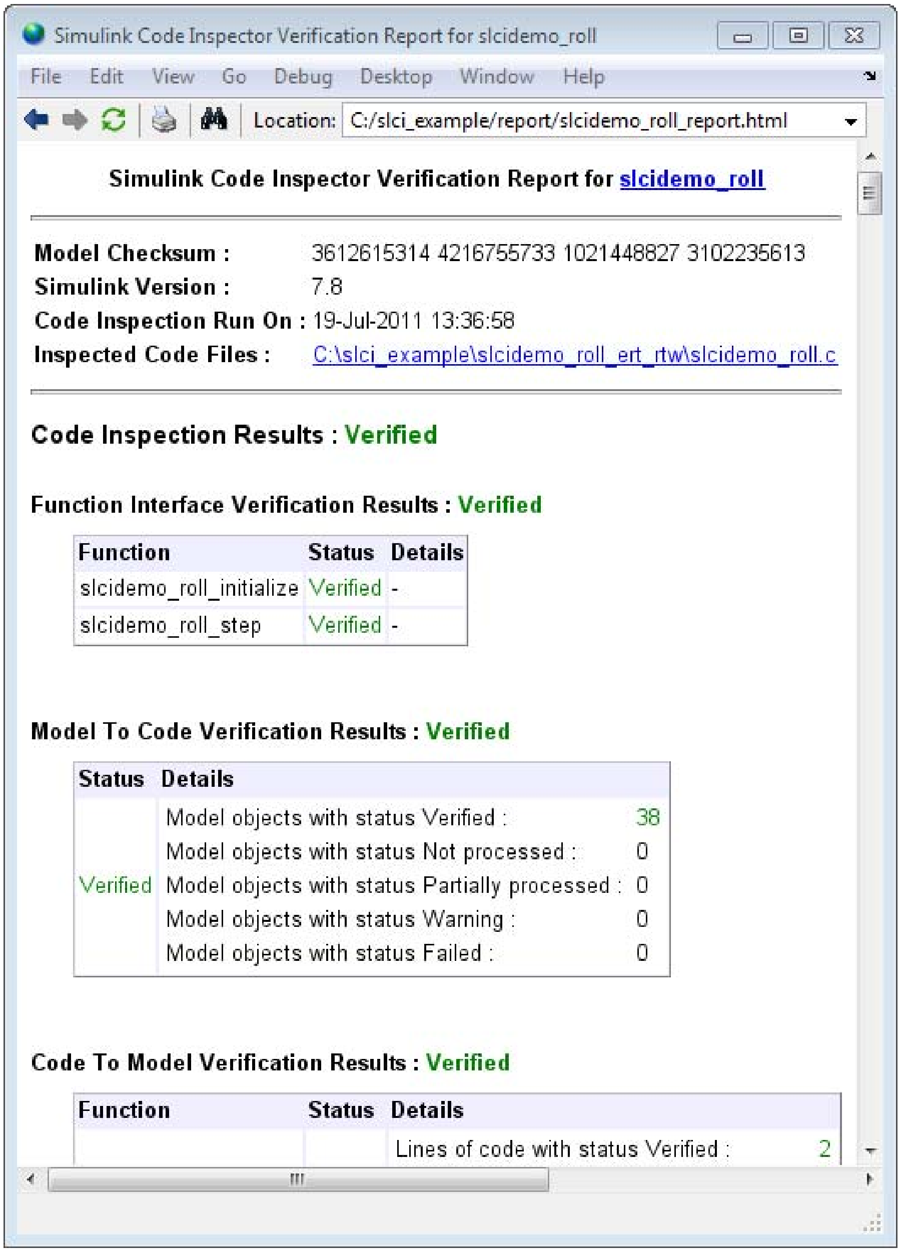
d The Code Inspector displays a summary in an HTML report window:


Simulink Code Inspector Summary Report for slcidemo_roll
File Edit View Go Debug Desktop Window Help
Location: C:/slci_example/report/slcidemo_roll_summaryReport.html

Simulink Code Inspector Summary Report for slcidemo_roll

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

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


Simulink Code Inspector Verification Report for slcidemo_roll
File Edit View Go Debug Desktop Window Help
Location: C:/slci_example/report/slcidemo_roll_report.html

Simulink Code Inspector Verification Report for slcidemo_roll

Model Checksum :	3612615314 4216755733 1021448827 3102235613
Simulink Version :	7.8
Code Inspection Run On :	19-Jul-2011 13:36:58
Inspected Code Files :	C:\slci_example\slcidemo_roll_ert_rtw\slcidemo_roll.c

Code Inspection 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 Not processed :	0
	Model objects with status Partially processed :	0
	Model objects with status Warning :	0
	Model objects with status Failed :	0

Code To Model Verification Results : Verified

Function	Status	Details	
		Lines of code with status Verified :	2

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

Inspecting Code Using the Command-Line Interface

To programmatically control code inspection, use the `slci.Configuration` interface. For a complete list of applicable `slci.Configuration` methods, see the "Code Inspection" category in the function reference documentation.

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 demo `slcidemo_intro`.

DO-178B Objectives Compliance

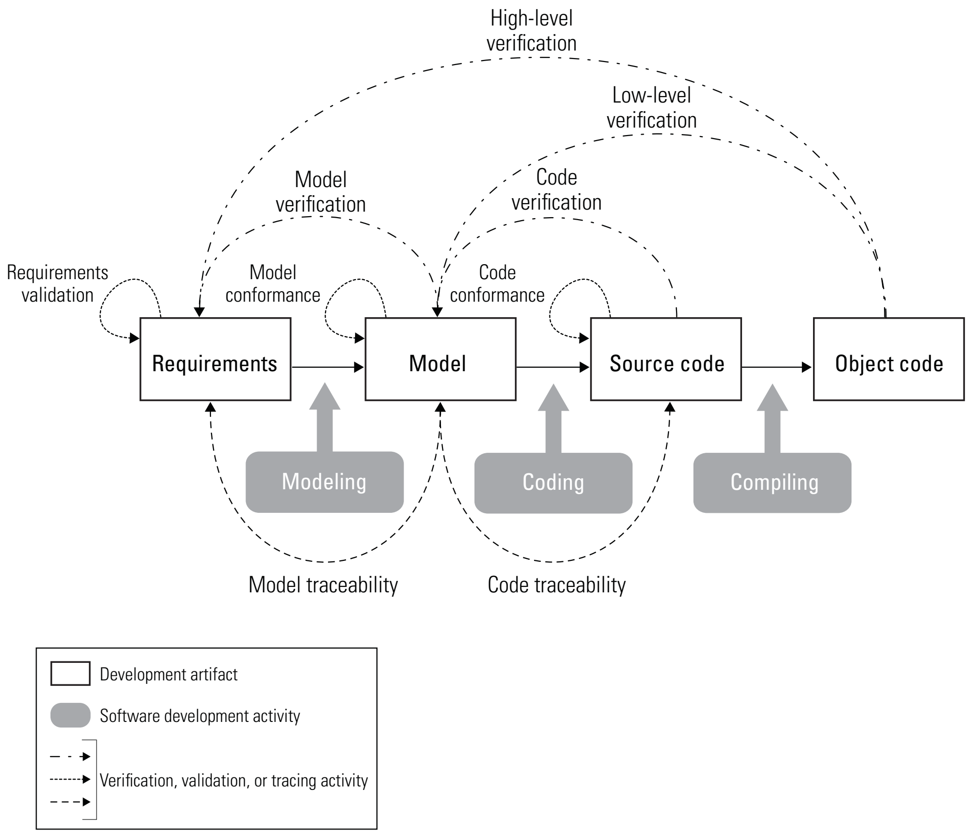
Model-Based Design Workflow in DO-178B

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards. DO-178B, 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-178B standard. The kit also provides detailed information on how to apply Model-Based Design to DO-178B. For more information, see http://www.mathworks.com/products/do-178/.

The DO-178B software life cycle consists of objectives that must be met for each of the life cycle stages. In Appendix A of the DO-178B standard, these objectives are summarized in tables. The DO Qualification Kit document *Model-Based Design Workflow for DO-178B* 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-178B 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 Step	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*

Workflow Step	Available Products and Capabilities for Model-Based Design
Model conformance	Simulink — Model Advisor checks, Simulink® Coder™ — Model Advisor checks, Simulink Verification and Validation — Model Advisor checks, Simulink Verification and Validation — DO-178B checks*, Simulink Report Generator — System Design Description report*
Model verification	SystemTest™ — Limit Check element*, Simulink® Design Verifier™ — Property Proving, Simulink Verification and Validation — Model Coverage*, Simulink Report Generator — System Design Description report*
Coding	Embedded Coder
Code traceability	Simulink Code Inspector
Code conformance	Polyspace® Products for C/C++*
Code verification	Simulink Code Inspector
Compiling	Embedded Coder — IDE Link
Low-level verification	SystemTest — Limit Check element*, Simulink Design Verifier — Test Generation, Embedded Coder — IDE Link, Polyspace Products for C/C++*
High-level verification	SystemTest — Limit Check element*, Embedded Coder — IDE Link, Polyspace Products for C/C++*
*The DO Qualification Kit product may be used to support DO-178B tool qualification.	

Compatible DO-178B Objectives

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

Annex A Table	Objectives	DO-178B Reference	Software Levels	Anticipated Certification Credit
A-5	(1) Source code complies with low-level requirements	Section 6.3.4a	A, B, C	Full — Simulink Code Inspector
A-5	(2) Source code complies with software architecture	Section 6.3.4b	A, B, C	Full — Simulink Code Inspector
A-5	(3) Source code is verifiable	Section 6.3.4c	A, B	Full — Simulink Code Inspector, Polyspace dead code analysis
A-5	(4) Source code conforms to standards	Section 6.3.4d	A, B, C	Full — Polyspace MISRA-AC ACG rules checker
A-5	(5) Source code is traceable to low-level requirements	Section 6.3.4e	A, B, C	Full — Simulink Code Inspector
A-5	(6) Source code is accurate and consistent	Section 6.3.4f	A, B, C	Full for source code based criteria — Simulink Code Inspector, Polyspace verifier