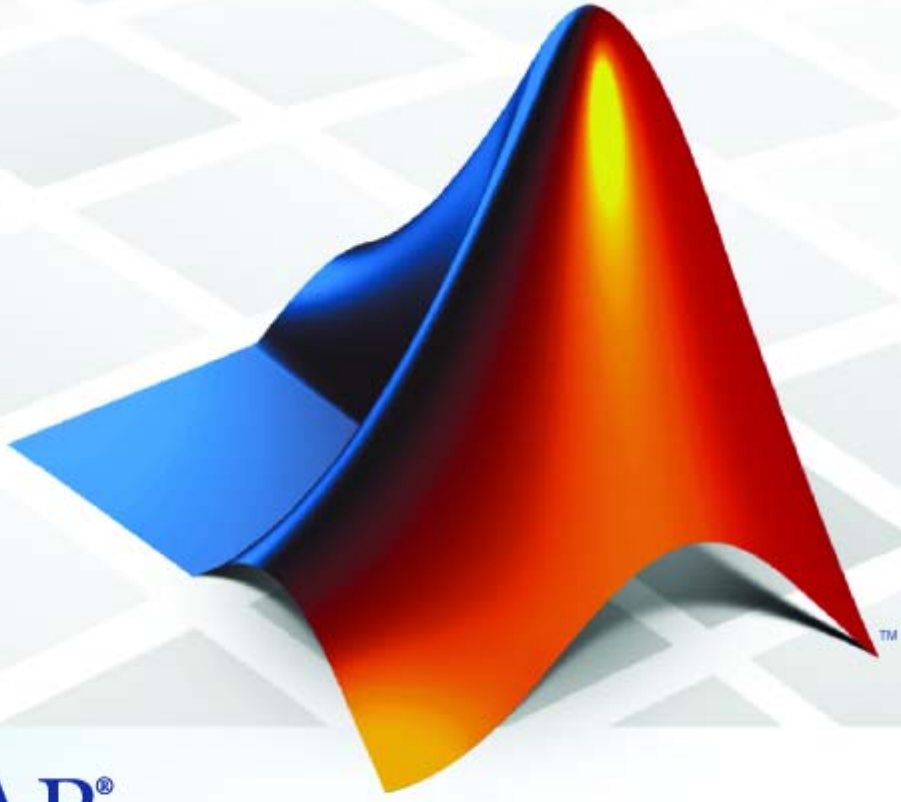
Simulink® Verification and Validation™ 2
User's Guide



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The 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® Verification and Validation™ User's Guide

Revision History

June 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Online only	Revised for Version 1.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.0.2 (Release 14SP2)
April 2005	Second printing	Revised for Version 1.1 (Web release)
September 2005	Online only	Revised for Version 1.1.1 (Release 14SP3)
March 2006	Online only	Revised for Version 1.1.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.0 (Release 2006b)
March 2007	Online only	Revised for Version 2.1 (Release 2007a)
September 2007	Online only	Revised for Version 2.2 (Release 2007b)
March 2008	Online only	Revised for Version 2.3 (Release 2008a)
October 2008	Online only	Revised for Version 2.4 (Release 2008b)
March 2009	Online only	Revised for Version 2.5 (Release 2009a)

Contents

1 Getting Started

2 Managing Model Requirements

3 Managing Model Requirements with DOORS Software

4 Managing Model Verification Blocks

5 Using Model Coverage

6 Customizing the Model Advisor

7 Function Reference

8 Class Reference

9 Alphabetical List

10 Block Reference

11 Model Advisor Checks

Examples

A

Index

1

Getting Started

The Simulink® Verification and Validation™ software uses component tools that contribute to the work of certifying the correct design, implementation, and testing of Simulink® models. Use the following topics to become familiar with the Simulink Verification and Validation software.

Product Overview

The Simulink Verification and Validation software is a Simulink product that helps you do the following:

- Establish requirements for a Simulink model by linking them with model elements that satisfy them
- Verify proper function of the model by monitoring model signals during extensive testing
- Validate the model, making sure that all possible model decisions are taken through testing.
- Customize the Model Advisor to analyze a model for settings that result in inaccuracies or inefficiencies.

In short, the elements of the Simulink Verification and Validation software give you confidence in the behavior of your Simulink models.

System Requirements

In this section...

Operating System Requirements

The Simulink Verification and Validation software works with the following operating systems:

- Microsoft® Windows® XP and Windows Vista™
- UNIX® systems where the MATLAB® software supports the Java™ programming language (for HTML-based requirements documents only)

Product Requirements

The Simulink Verification and Validation software requires the following MathWorks™ products:

- MATLAB
- Simulink

If you want to use the Requirements Management Interface with Stateflow® charts, the Simulink Verification and Validation software requires the following MathWorks product:

- Stateflow

The Requirements Management Interface in the Simulink Verification and Validation software allows you to associate requirements with Simulink models and Stateflow charts. The software supports the following applications for documenting requirements:

- Microsoft Word 2000 or later
- Microsoft® Excel® 98 or later
- Telelogic® DOORS® 6.0 or later

Organization of This User's Guide

The component tools of the Simulink Verification and Validation software are organized on the basis of work flow that you follow in certifying the correct and complete behavior of your models. This work flow is described in the following steps:

1. Establish performance requirements for the model and link them with model elements using the Requirements Management Interface, which is described in the following chapters:
   - Chapter 2, "Managing Model Requirements" — Instructions for using the standard version of the Requirements Management Interface. Use this to associate Simulink models, Stateflow charts, and MATLAB M-files with requirements in HTML, Microsoft Word, and Microsoft Excel documents.
   - Chapter 3, "Managing Model Requirements with DOORS Software" — Instructions for using the DOORS® software with the Requirements Management Interface. Use this if you want to associate Simulink models, Stateflow charts, and MATLAB M-files with requirements in the DOORS software.
2. Verify proper performance of the model by monitoring model signals during extensive testing with model verification blocks using the Verification Manager, which is described in the following chapter:
   - Chapter 4, "Managing Model Verification Blocks" — Shows you how to use verification blocks individually in Simulink models and how to manage them as a group for testing.
3. Validate the model by making sure that all possible model decisions are taken through testing, by using the Model Coverage tool, which is described in the following chapter:
   - Chapter 5, "Using Model Coverage" — Shows you how to generate and interpret model coverage reports and displays for validating model decisions.
4. Customize the Model Advisor to analyze your model for conditions and configuration settings that result in inaccurate or inefficient simulation or code generation. You can write custom checks, tasks, and callback functions, as described in the following chapter:

- Chapter 6, "Customizing the Model Advisor" — Shows you how to define custom checks and tasks, write callback functions, and register customizations for the Model Advisor.

The last portion of the User's Guide is comprised of function and block reference chapters:

- Chapter 7, "Function Reference" — Provides a categorical list of functions used in executing and managing model coverage tests and reports from the MATLAB prompt. Automate your model coverage tests with scripts of MATLAB commands calling these functions.
- Chapter 9, "Alphabetical List" — Provides an alphabetical reference of functions used in executing and managing model coverage tests and reports from the MATLAB prompt.
- Chapter 10, "Block Reference" — Provides reference information for the Simulink Verification and Validation library, which currently contains only one block, System Requirements. This block lets you list all the requirements for a model or subsystem on its Simulink diagram.

2

Managing Model Requirements

The Requirements Management Interface (RMI) associates requirements documents with objects in Simulink models. To learn how to use the RMI, see the following sections:

What Is the Requirements Management Interface?

The Requirements Management Interface (RMI) allows you to associate requirements with Simulink models and Stateflow charts. A requirement has the following attributes:

- A requirement description of up to 255 characters
- The path name of a requirements document, such as a Microsoft Word file. (The RMI supports several built-in document formats and also allows you to register your own custom types of requirements documents.)
- A link to a location inside the requirements document

Use the RMI to:

- Associate requirements with:
  - Simulink models
  - Simulink subsystems and blocks
  - Stateflow charts, states, transitions, boxes, and functions
- Navigate from a Simulink block or Stateflow object in a diagram or in the Model Explorer to a requirement
- Navigate from an embedded link in a requirements document to the corresponding Simulink or Stateflow object (when you create two-way links using the selection-based linking mechanism)
- View objects in Simulink or Stateflow diagrams that have requirements associated with them

Configuring the Requirements Management Interface

Before you start using the RMI, in the MATLAB Command Window type:

```
rmi setup
```

This command runs a setup script that installs Microsoft® ActiveX® controls for establishing two-way selection-based links. If you have installed Telelogic DOORS software on the machine, this command also invokes the corresponding setup script. For more information, see “Configuring the Requirements Management Interface for DOORS Software” on page 3-3.

Adding and Viewing Requirement Links

In this section...

Object and Document Types

You can add requirements to the following types of objects:

- Simulink model
- Simulink block
- Stateflow chart, state, transition, box, or function

Note You can add requirements to top-level Model and Subsystem blocks but not to their contents. For example, if you copy a subsystem with multiple blocks from a library, you can add requirements to the Subsystem block in your model, but not to its component blocks.

The RMI supports the following built-in types of requirements documents:

- Text
- HTML
- PDF
- Microsoft Word 2007 and earlier
- Microsoft Excel 2007 and earlier

You can also link to an item in the Telelogic DOORS software or register your own custom type of requirements documents to link to. For more information, see “Linking Objects to DOORS Requirements” on page 3-9 and “Linking to Custom Types of Requirements Documents” on page 2-30.

Location Types

Depending on the requirements document type, you can link to specific locations within a document.

Requirements Document Type	Location Options
Text	• `Search text` — Type a string in the **Location** text field. The RMI searches for the first occurrence of the text string within the document. This search is not case sensitive. • `Line number` — Type a line number in the **Location** text field. The RMI makes a link to the specified line.
HTML	You can link only to a named anchor. For example, if you define the anchor `<A NAME=valve_timing> ...contents... </A>` in your HTML requirements document, you can enter `valve_timing` in the **Location** text field. Or, you can click the **Document Index** tab to select `valve_timing` from an automatically generated list of anchors in the document.

Requirements Document Type	Location Options
Microsoft Word	• `Search text` — Type a string in the **Location** text field. The RMI searches for the first occurrence of the text string within the document. This search is not case sensitive. • `Named item` — Link to a bookmark within the document. The RMI automatically generates a document index based on its headings and bookmarks, or you can type the name in the **Location** text field. • `Page/item number` — Type a page number in the **Location** text field. The RMI creates a link to the top of the page.
Microsoft Excel	• `Search text` — Type a string in the **Location** text field. The RMI searches for the first occurrence of the text string within the document. This search is not case sensitive. • `Named item` — Link to a named item within the document (defined in the Excel® software using **Insert > Name**). Type the name in the **Location** text field. • `Sheet range` — Type a cell number or a range of cells (such as `C5:D7`) in the **Location** text field. The RMI creates a link to the specified cell or cells.

Requirements Document Type	Location Options
PDF	• `Named item` — Link to a bookmark within the document. The RMI automatically generates a document index based on the document's headings and bookmarks. Alternatively, you can enter the bookmark name in the **Location** field. • `Page/item number` — Type a page number in the **Location** text field. The RMI creates a link to the top of the page.
Web Browser URL	You can link to a URL location only. Type the URL location string in the **Document** text field. When you click the link, the document opens in a Web browser.

Adding Requirement Links to an Object

You use the Requirements dialog box to associate a requirements document with a requirements object. You can link a location in a Microsoft Word or HTML document to a block in a Simulink model or Stateflow object in a Stateflow chart.

In this procedure, you add three requirement links to a Simulink block in the demo model, `sf_car`. Later topics explain how to modify both the links and the documents they point to.

1. Create and save a Microsoft Word 2007 document, `requirements.docx`, with the following content. Style the header lines ("Primary Requirements", "Second Requirements", "Tertiary Requirements") as `Heading 1` in Microsoft Word.

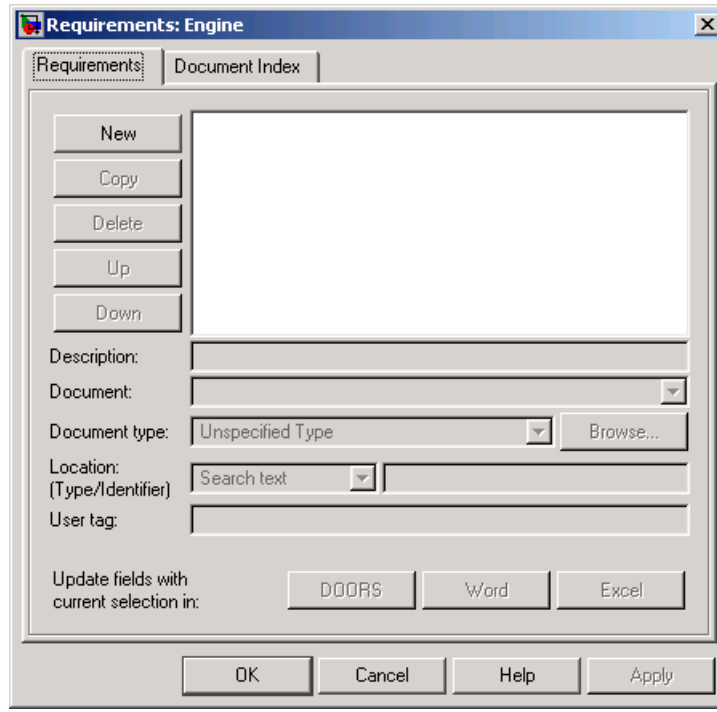

requirements.docx - Mi
Home
Insert
Page Layout
References
Mailings
Review
Paste
Clipboard
Times New Roman
11
Font
Paragraph
Primary Requirements
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Secondary Requirements
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Tertiary Requirements
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.
Requirement text. Requirement text. Requirement text.

Note The requirements document used in this exercise is in Microsoft Word 2007 format. If that software is not available, you can use earlier versions of the Microsoft Word software to run these steps.

2 Type `sf_car` at the MATLAB prompt to open the demo model, `sf_car`.

3 Right-click the Engine block and select **Requirements > Edit/Add Links**.

The Requirements dialog box for the Engine block opens.



4 In the Requirements: Engine dialog box, click **New** to add a new requirement.

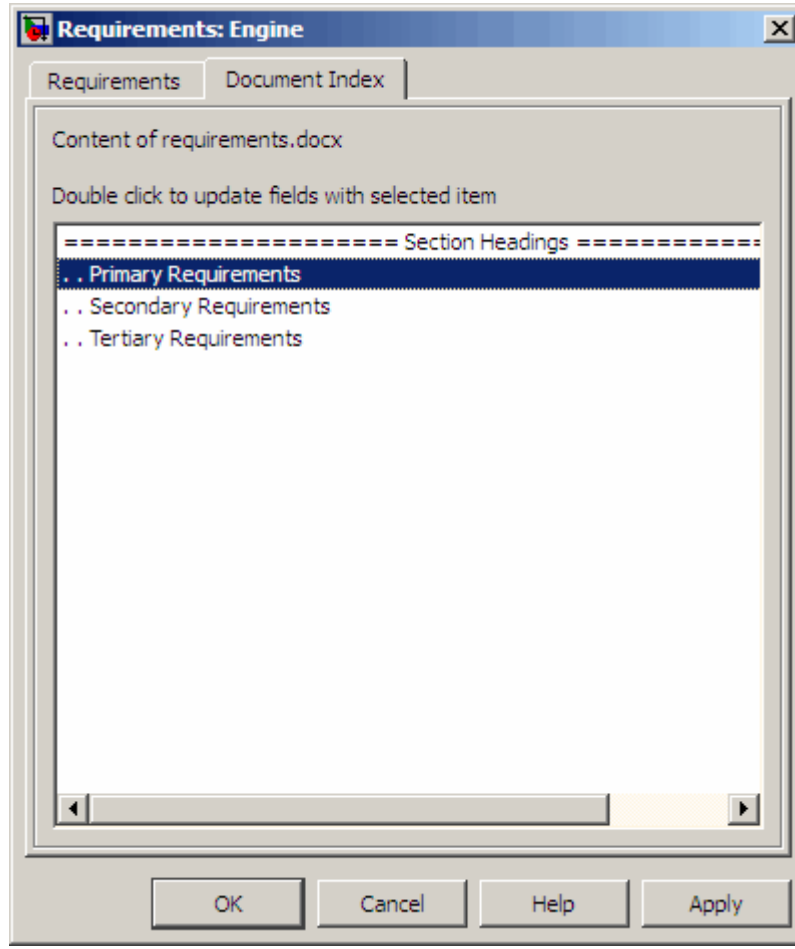
5 In the **Description** field, enter `Requirement 1`.

6. Click **Browse** next to the **Document type** field, browse to the requirements document, `requirements.docx`, and select **Open**.

   The **Document type** field is now `Microsoft Word`. If you specify the document type in the **Document type** field before browsing for the requirements document, only the files of the specified type appear.

7. To define a particular location in the document, click the **Document Index** tab to create an index of the requirements document.

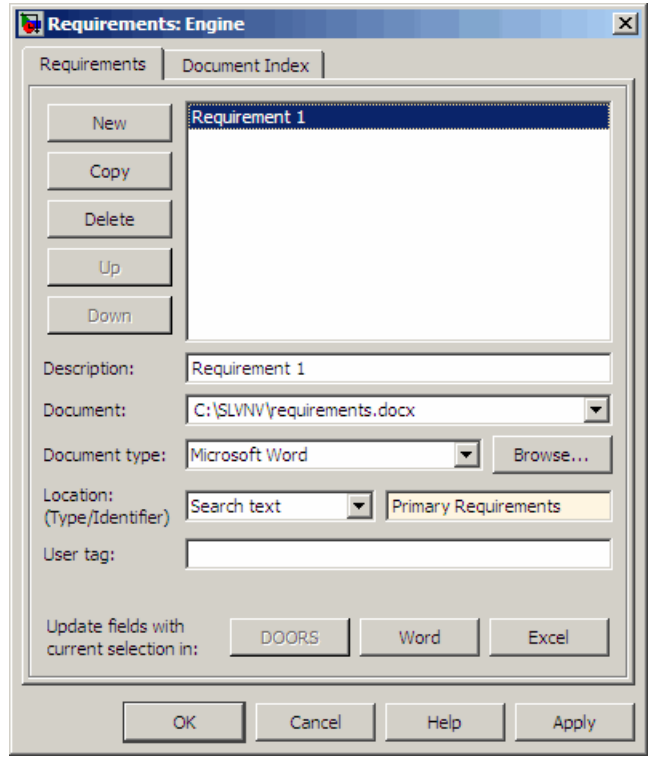
8. Select `Primary Requirements` from the automatically generated list of headings and bookmarks in the document.



9 Click **Apply** to create the link from that requirements document text to the Engine block and redisplay the **Requirements** tab.

10 If your document does not contain headings or bookmarks, click the **Requirements** tab.

Select `Search text` from the **Location** drop-down list and enter `Primary Requirements` in the text field. The search text feature is not case sensitive.



11 To provide additional details about the current requirement, enter text in the **User tag** field. This is optional.

12 Click **Apply**.

13 Select `Requirement 1` and click **Copy** to create a copy of Requirement 1, which appears in the **Description** text box, ready to be altered as a new requirement.

14 In the **Description** text box, remove

```
Copy of Requirement 1
```

and enter instead

```
Requirement 2
```

15 Link Requirement 2 to the text “Secondary Requirements” in `requirements.docx`.

16 Click **Apply** to apply the requirement links you have added. Click **OK** to close the Requirements dialog box.

Note When you add a requirement link to a Model or Subsystem block, the software does not add the requirement to children of the block.

---

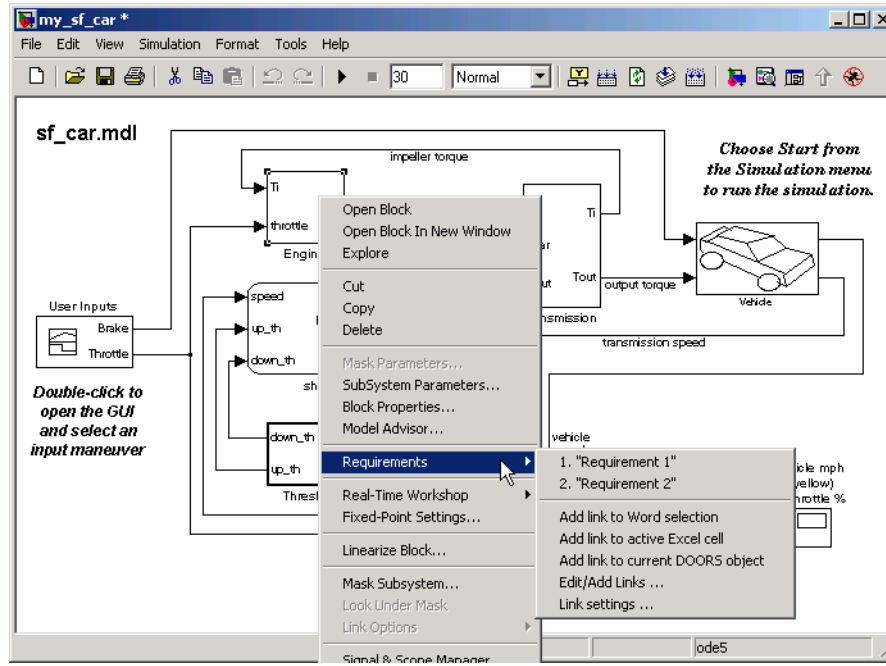
17 Save the model as `my_sf_car.mdl`.

Viewing Requirements Documents

You can access a requirements document through its associated model element. In this example, you access the requirements document using one of the Engine block requirements links:

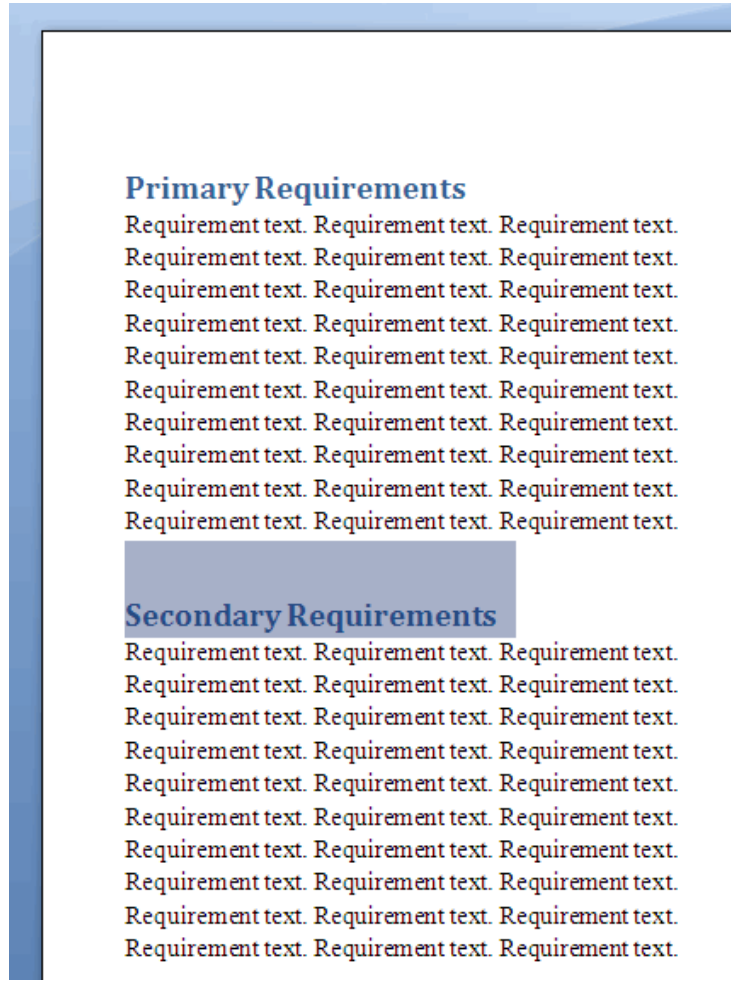
1 In the `my_sf_car` model, right-click the Engine block and select **Requirements** from the context menu.

The requirements you previously added appear as submenu selections.



2 Select **"Requirement 2"** from the submenu.

`requirements.docx` opens in the Microsoft Word editor, with the first occurrence of "Secondary Requirements" highlighted.



If you do not specify any text, or the specified string does not exist in the requirements document, the requirements document opens with the cursor at the beginning of the file.

3 Keep `requirements.docx` and the `my_sf_car` model open for use in "Adding Requirement Links to Multiple Objects Simultaneously" on page 2-17.

Resolving the Document Path

When you browse and select a document to enter it in a requirements link, the software enters the location of the document with a fully specified absolute path. You can also enter a relative path for the document location. A relative path can be a partial path or no path at all. If you use a relative path, the document is not constrained to a single location in the file system. With a relative path, the RMI resolves the exact location of the requirements document in this order:

1. The software attempts to resolve the path relative to the current MATLAB directory.
2. If there is no path specification and the document is not in the current directory, the software uses the MATLAB search path to locate the file.
3. If the document is not located relative to the current directory or the MATLAB search path, the path is resolved relative to the model file directory.

The following examples illustrate the procedure for locating the specified requirements document.

Relative Path Specified Example

Current MATLAB directory	`C:\work\scratch`
Model file	`C:\work\models\controllers\pid.mdl`
Document link	`..\reqs\pid.html`
Documents searched for (in order)	`C:\work\reqs\pid.html` `C:\work\models\reqs\pid.html`

No Path Specified Example

Current MATLAB directory	`C:\work\scratch`
Model file	`C:\work\models\controllers\pid.mdl`

Requirements document	`pid.html`

Documents searched for (in order)	`C:\work\scratch\pid.html`
	`<MATLAB path dir>\pid.html`
	`C:\work\models\controllers\pid.html`

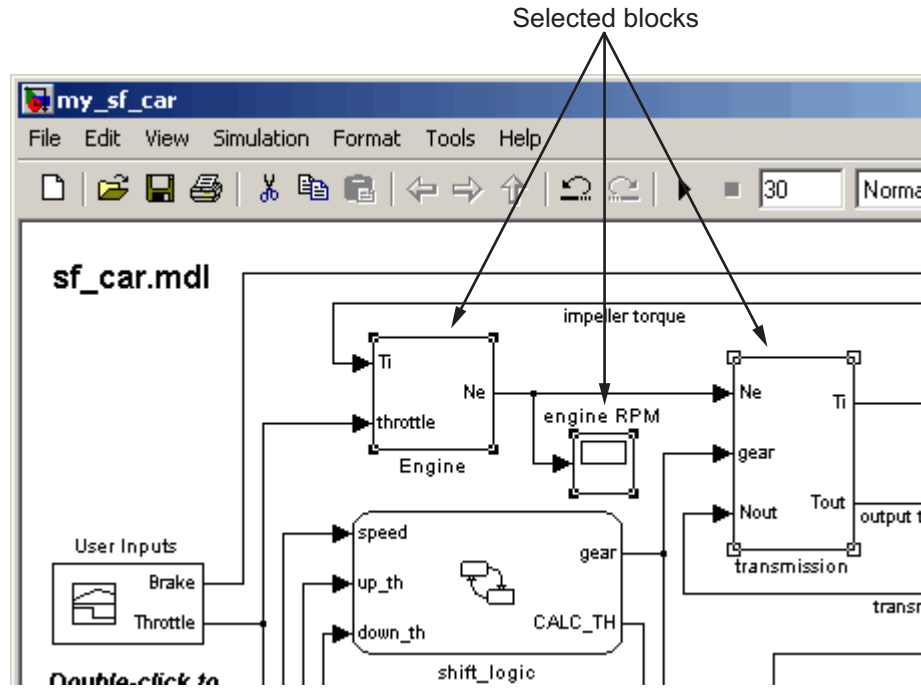
Absolute Path Specified Example

Current MATLAB directory	`C:\work\scratch`
Model file	`C:\work\models\controllers\pid.mdl`
Requirements document	`C:\work\reqs\pid.html`
Documents searched for	`C:\work\reqs\pid.html`

Adding Requirement Links to Multiple Objects Simultaneously

You can add or delete requirement links for a selection of multiple Simulink blocks or Stateflow objects as follows:

1. In the `my_sf_car` model, select the Engine, engine RPM, and transmission blocks.

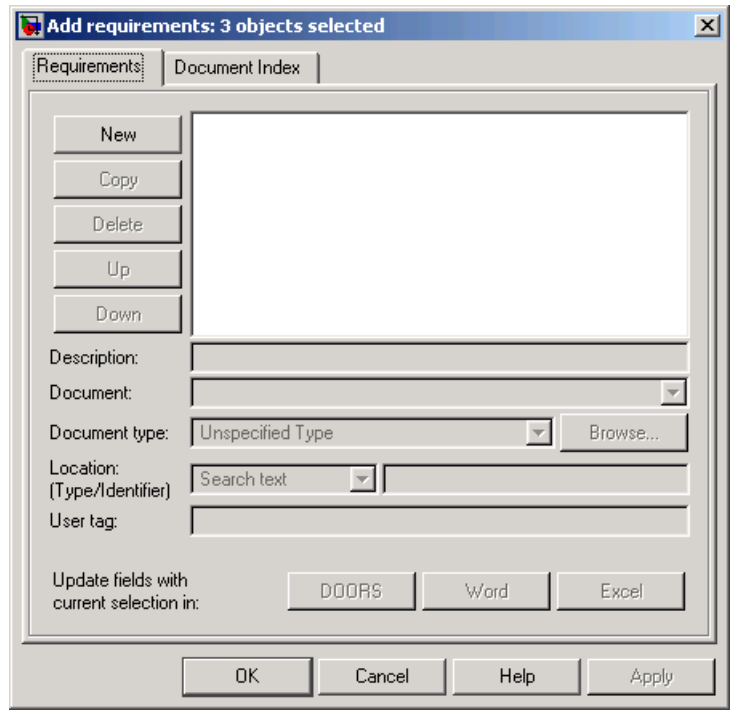


You can select multiple Simulink blocks or Stateflow objects in one of the following ways:

- Hold down the **Shift** key while clicking each block.
- Click and drag a selection rectangle around the blocks.

2 Right-click any of the selected blocks and select **Requirements > Add Links to All**.

The Add requirements dialog box opens for the three selected blocks.



3 Add a new Requirement 3 for these blocks that points to the text "Tertiary Requirements" in the file, `requirements.docx`.

   Add the requirement as you would for a single block, as described in "Adding Requirement Links to an Object" on page 2-7.

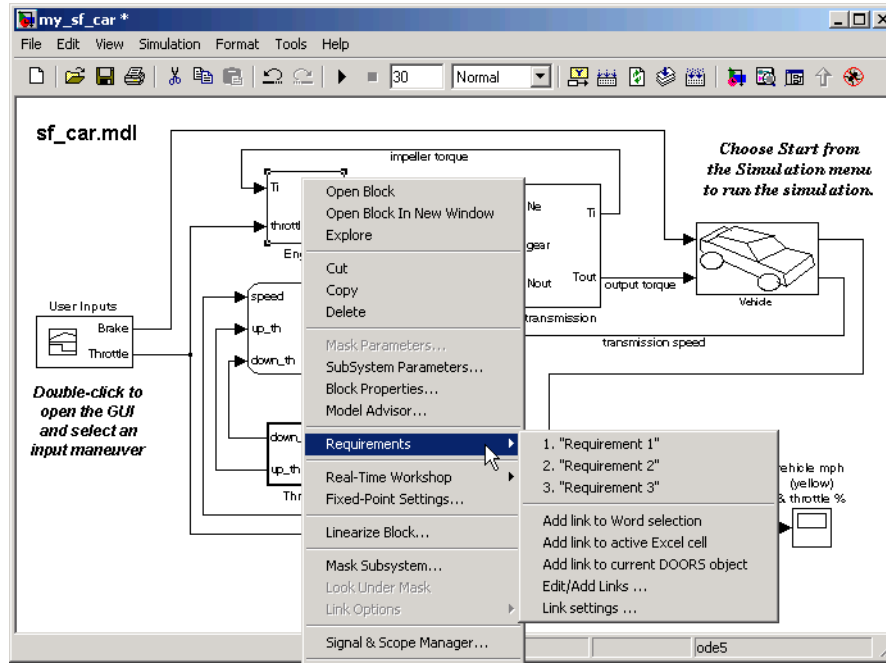
4 Click **Apply** to apply the requirement links.

5 Click **OK** to close the Requirements dialog box.

6 In the `my_sf_car` model, click outside the three objects to clear their selection.

7 Right-click the Engine block and select **Requirements**.

   The Engine block now has three requirements.



8 Right-click the engine RPM and transmission blocks to verify that they have only one requirement—Requirement 3.

9 Save the `my_sf_car` model.

Selection-Based Linking

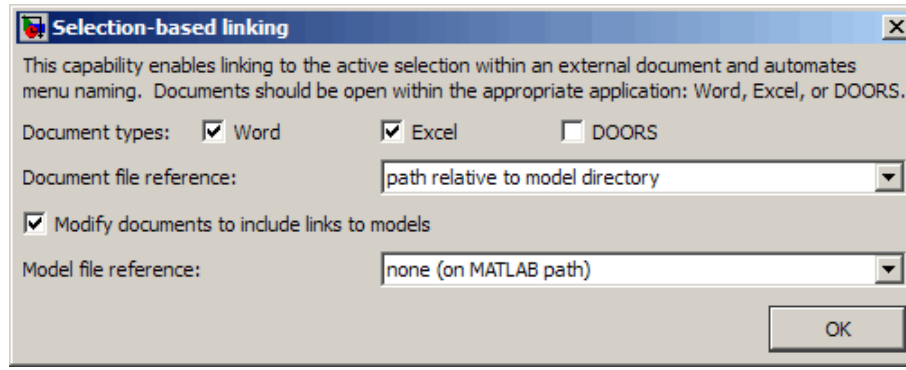
Selection-based linking is a quick way to create links between model elements and selected portions of a requirements document. The requirements document can only be either a Microsoft Word or Excel file. This method creates a two-way link by embedding a Microsoft ActiveX control into the requirements document next to the selected string or cell.

Configuring Selection-Based Linking

To configure selection-based linking:

1 Open the `my_sf_car` model.

2 In the model window, select **Tools > Requirements > Link settings**. The Selection-based linking dialog box opens.



3 Select the check box next to **Word** if it is not already selected.

4 Specify the following preferences:

- In the **Document file reference** drop-down list, specify how to store the document location. For more information, see "Resolving the Document Path" on page 2-16.
- To create two-way links, select **Modify documents to include links to models**. Two-way linking embeds an ActiveX® control in your requirements document. This control lets you navigate from the requirements document to the Simulink model and from the model to the requirements document.
- If you create two-way links using the **Model file reference** drop-down list, specify how to locate the model path from a requirements document using one of these options:
  - `none (on MATLAB path)` — Locate the model on the MATLAB path.
  - `Absolute` — Use the absolute path to locate the model.

5 Click **OK** to close the Selection-based linking dialog box.

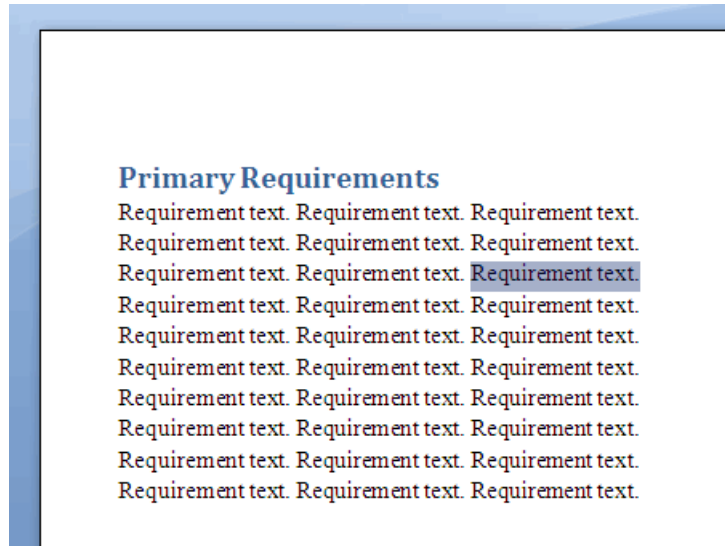
Making Selection-Based Links

To create selection-based requirement links:

1 Right-click the Engine block and select **Requirements > "Requirement 1"**.

`requirements.docx` opens in a Microsoft Word window, with "Primary Requirements" selected.

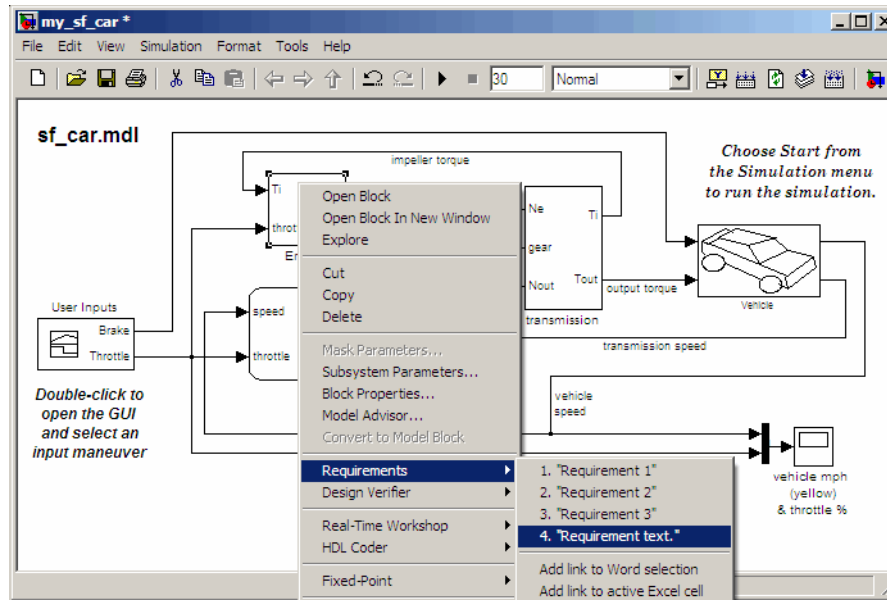
2 Select a portion of the text under "Primary Requirements", as shown.



3 In the Simulink model, right-click the Engine block and select **Requirements > Add link to Word selection**.

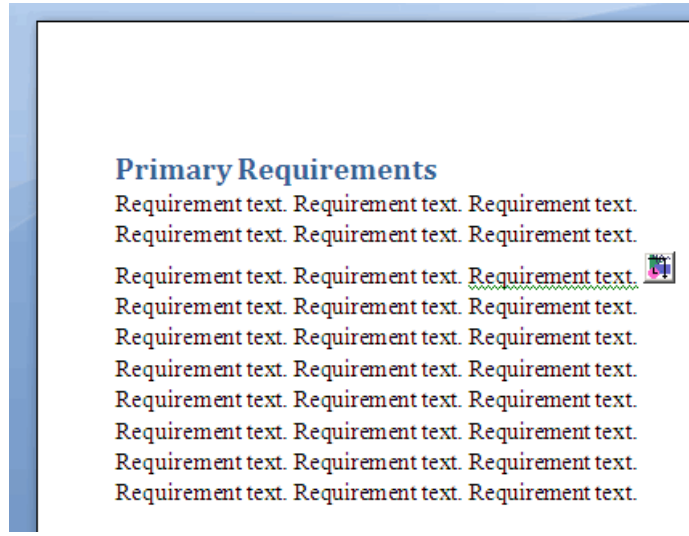
Note The Word document must be open and you must have selected some text for the **Add link to Word selection** option to work. If no Word document is open, the software asks you to open a document or cancel the operation.

The RMI creates the link. If you right-click the Engine block and select **Requirements**, the Engine block now has four requirement links.

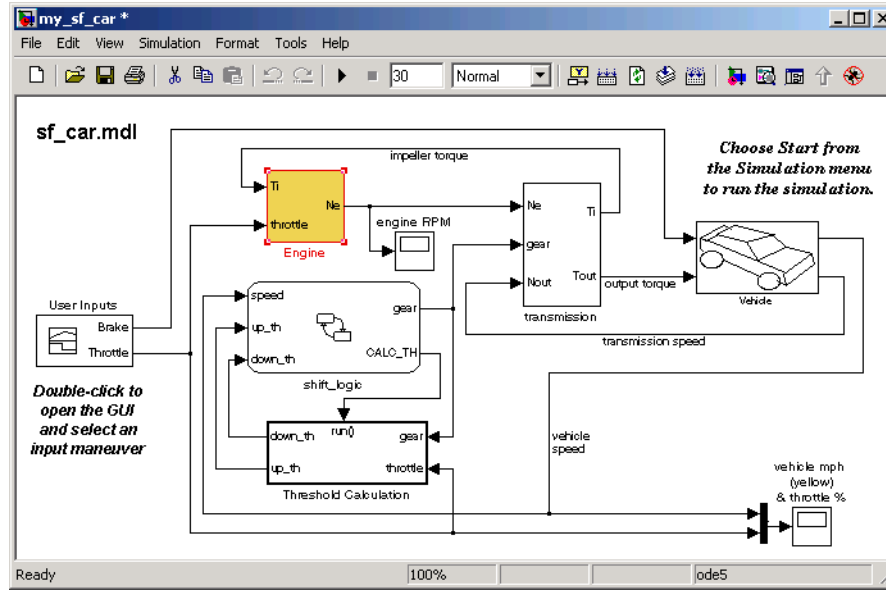


If you select **Requirements > Requirement text.**, a Microsoft Word window opens the requirements document at the appropriate text.

If you configured two-way linking, an ActiveX control  is embedded in the requirements document next to the selected string.



4 Double-click the ActiveX control to display the `my_sf_car` model, with the Engine block highlighted.



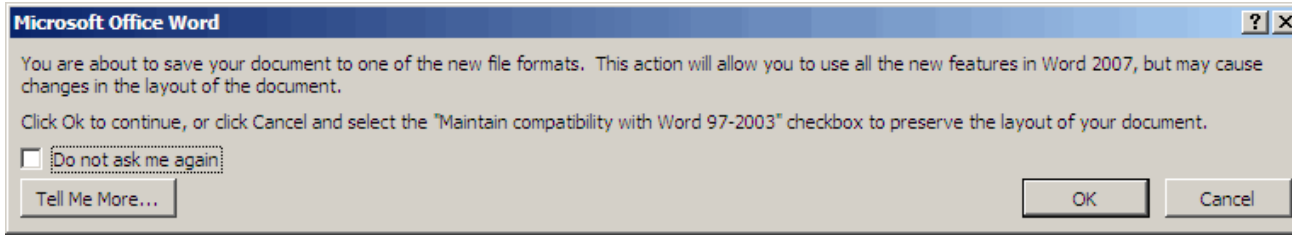
Troubleshooting ActiveX Controls in Microsoft Office 2007

Warning When Saving Requirements Documents to Microsoft Office 2007 Format. If you created a requirements document with a version of Microsoft Office prior to Office 2007, the two-way links work as designed. If you save the document in Microsoft Office 2007 format, they will continue to work. Follow these steps:

1 Click the **Microsoft Office Button**.

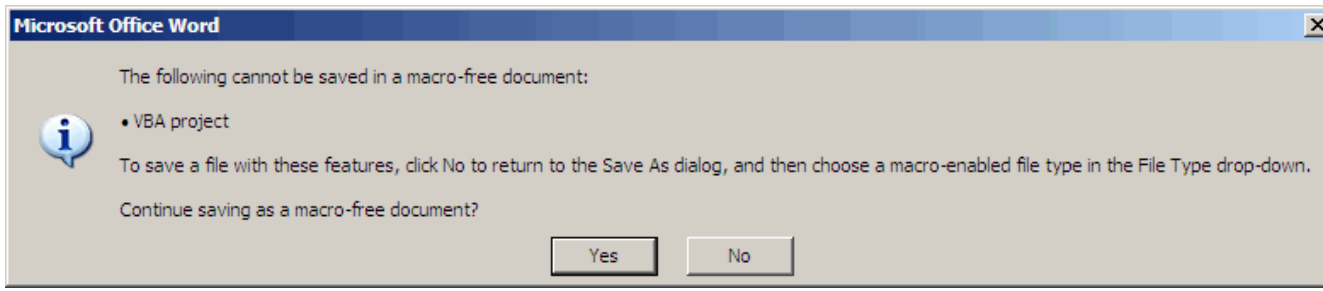
2 Select **Save As > Word Document**.

The following dialog box appears.



3 Click **OK**.

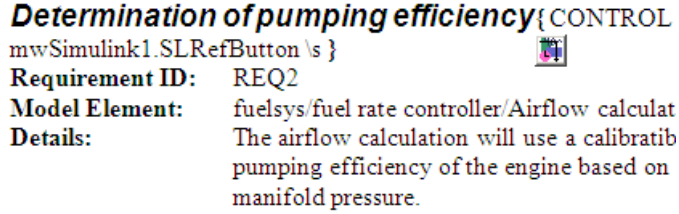
The following dialog box appears.



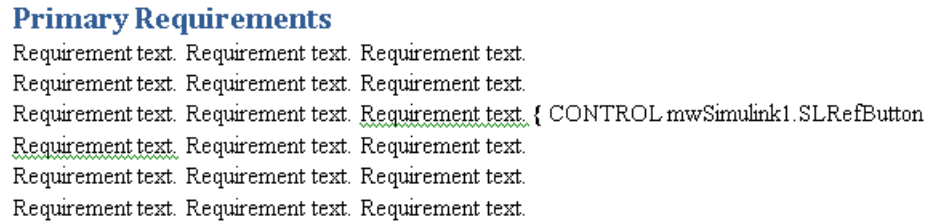
4 Click **Yes** to save the current document in Microsoft Word 2007 format, with a `.docx` extension.

Field Codes Appear in Requirements Document. If your Microsoft Word requirements document displays the field codes in addition to or instead of the ActiveX icon, as in one of the following graphics, you must change a setting in Microsoft Word.

A requirements document created in Microsoft Word 2003:



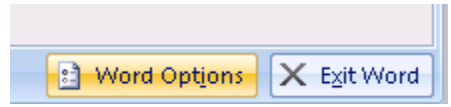
A requirements document created in Microsoft Word 2007:



To hide the field codes and display the ActiveX icon:

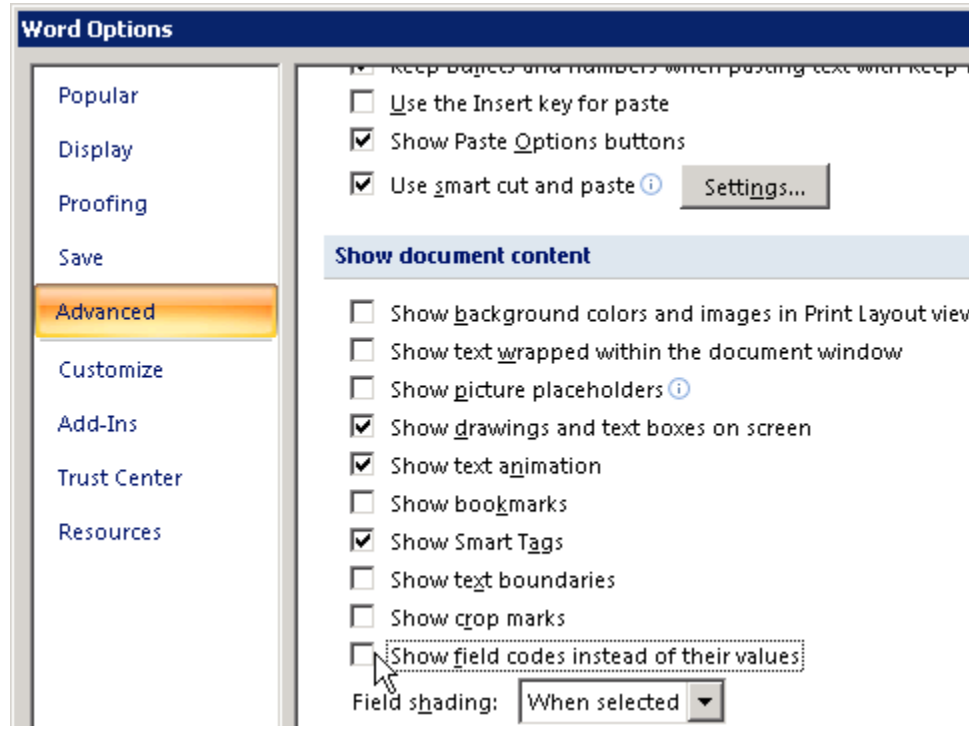
1 In the Microsoft Word window, in the upper-left corner, click the **Microsoft Office Button**.

2 In the pane that opens, at the bottom, click **Word Options**.



3 On the left-hand pane, click **Advanced**.

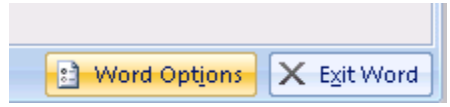
4 On the **Advanced** pane, scroll to the **Show document content** section and clear the **Show field codes instead of their values** option.

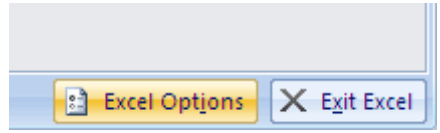


Clicking ActiveX Controls Do Not Link to Model Element. If you click an ActiveX control that links to a Simulinkor Stateflow element and the element doesn't open, you need to enable ActiveX controls:

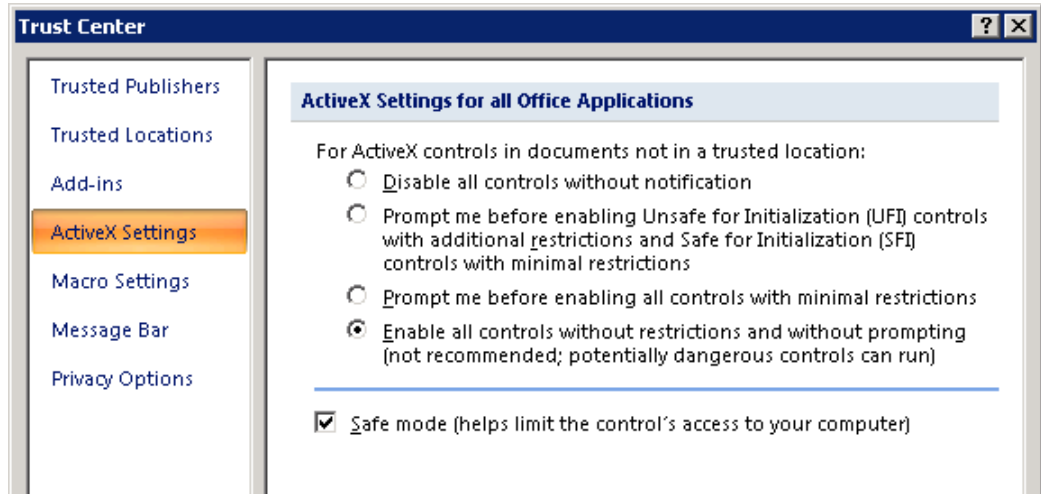
1. In the Microsoft Word window, in the upper-left corner, click the **Microsoft Office Button**.

2. In the pane that opens, at the bottom, click **Word Options** or **Excel Options**.





3. In the left-hand pane, click **Trust Center**.
4. In the **Trust Center** pane, click **Trust Center Settings**.
5. On the **Trust Center** pane on the left, select **ActiveX Settings**.



6 Select the setting that you want for ActiveX controls:

- **Prompt me for enabling all control with minimum restrictions** if you want to decide each time you click an ActiveX control if you want to enable all controls.
- **Enable all controls without restrictions and without prompting** if you want to enable all ActiveX controls.

Deleting All Requirement Links for Multiple Objects Simultaneously

To delete the existing requirements for a group of selected blocks, right-click one of the selected blocks and select **Requirements > Delete All**. This option deletes all the requirement links for all the selected blocks, whether you added them individually or as a group.

Linking to Custom Types of Requirements Documents

In this section...

Built-In Link Types

The files for built-in link types are in the private directory of the requirements management tool (*matlabroot*`\toolbox\slvnv\reqmgt\private`):

```
linktype_rmi_doors.m
linktype_rmi_excel.m
linktype_rmi_html.m
linktype_rmi_pdf.m
linktype_rmi_text.m
linktype_rmi_word.m
```

Built-in link types use the same format and naming convention as custom types. However, built-in link types use a different system for identification in the model file that supports backward and forward compatibility. Use the built-in link types as examples when developing your custom link types.

Why Create a Custom Link Type?

In addition to linking to built-in types of requirements documents, you can register custom requirements document types with the RMI. Then you can create requirement links to these types of documents.

Custom link types let you define how you open and navigate to a document, browse for a document, and view an index of its contents. When you define a

custom link type, you create MATLAB M-code functions that perform these operations. The RMI invokes the registered code:

- When navigating to a document with the new link type you created.
- When browsing for a document or displaying the index of a document within the Requirements dialog box.

Using the external interfaces supported by the MATLAB software, you can interface with external applications and run programs from the command shell. You can also use the built-in Web browser and text editor to display custom variants of HTML and text files without loading external applications.

With custom link types, you can:

- Link to requirement items in commercial requirement tracking software
- Link to in-house database systems
- Link to document types that the tool does not support

Custom Link Type Registration

You register custom link types with a unique MATLAB function name. The function must exist on the MATLAB path and must not require any input arguments. It must return a single output argument that is an instance of the requirements link type class. You can register your link type with the following MATLAB command:

```
rmi register mytargetfilename
```

mytargetfilename is the name of the MATLAB function contained in the M-file named *mytargetfilename*.m.

Once you register a link type, it appears as an entry in the **Document type** drop-down list in the Requirements dialog box. A file in your preference directory contains the list of registered link types, so you can restore it in new MATLAB sessions. You can remove a link type with the following MATLAB command:

```
rmi unregister mytargetfilename
```

When you create links using custom link types, the software saves the registration name in the model. When you attempt to navigate to a link, the RMI resolves the link type against the registered list. If the software cannot find the link type, it displays an error message.

Link Properties

Requirement links are the data structures, saved in the Simulink model, that identify a specific location within a document. You get and set the links on a block using the `rmi` command. Link information is encapsulated within a MATLAB structure array. Each element of the array is a single requirement link.

Links and link types work together to perform navigation and manage requirement interfaces. The document and ID fields of links uniquely identify the linked item in an external document. The RMI passes both of these strings to the navigation command of the associated link type when it follows a link from the model or a generated report.

Link Type Properties

Link type properties define how links are created, identified, navigated to, and stored within the requirement management tool. The following table describes each of these properties.

Property	Description
`Registration`	The name of the M-file that creates the link type. This name is stored in the Simulink model.
`Label`	A string to identify this link type. This string appears on the **Document type** drop-down list in the Requirements dialog box for a Simulink or Stateflow object.

Property	Description
`IsFile`	A Boolean property that indicates if the linked documents are files within the computer file system. If a document is a file: • The software uses the standard method for resolving the path. • When you click **Browse** in the Requirements dialog box, the file selection dialog box opens.
`Extensions`	An array of file extensions. These file extensions are used as filter options when you click **Browse** in the Requirements dialog box. The extensions are also used to infer the link type based on the document name. If more than one link type is registered for the same file extension, the link type that was registered first takes priority.
`LocDelimiters`	A string containing the list of supported navigation delimiters. The first character in the ID of a requirement specifies the type of identifier. For example, an identifier can refer to a specific page number (`#4`), a named bookmark (`@my_tag`), or some searchable text (`?search_text`). The valid location delimiters determine the possible entries on the **Location** drop-down list in the Requirements dialog box.
`NavigateFcn`	The MATLAB callback that is invoked when you click a link. The function has two input arguments: the document field and the ID field of the link: `feval(LinkType.NavigateFcn, Link.document, Link.id)`

Property	Description
`ContentsFcn`	The MATLAB callback that is invoked when you click the **Document Index** tab in the Requirements dialog box. This function has a single input argument that contains the full path of the resolved function, or the **Document** field entry if the link type is not a file. The function returns three outputs: • Labels • Depths • Locations
`BrowseFcn`	The MATLAB callback that is invoked when you click **Browse** in the Requirements dialog box. This function is unnecessary when the link type is a file. The function does not take any input arguments and returns a single output argument that identifies the selected document.

Creating a Custom Link Requirement Type

In this example, you implement a custom link type to a hypothetical document type, a text file with the extension `.abc`. Within a document, the requirement items are identified with a special text string, `Requirement::`, followed by a single space and then the requirement item inside quotation marks (").

In the following example, you create a document index containing a list of all the requirement items. When navigating from the Simulink model to the requirements document, the document opens in the MATLAB Editor at the line of the requirement that you want.

To create a custom link requirement type:

1. Write a function that implements the custom link type and save it as an M-file on the MATLAB path. In this example, the file, `rmicustabcinterface.m`, containing the function, `rmicustabcinterface`, that implements the ABC files is included in the installation. You can view it here, or by typing `edit rmicustabcinterface` at the MATLAB prompt.

```
function linkType = rmicustabcinterface
```

```
%RMICUSTABCINTERFACE - Example custom requirement link type
%
% This file implements a requirements link type that maps
% to "ABC" files.
% You can use this link type to map a line or item within an
% ABC file to a Simulink or Stateflow object.
%
% You must register a custom requirement link type before
% using it. Once registered, the link type will be reloaded in
% subsequent sessions until you unregister it. The following
% commands perform registration and registration removal.
%
% Register command:   >> rmi register rmicustabcinterface
% Unregister command: >> rmi unregister rmicustabcinterface
%
% There is an example document of this link type contained in
% the requirement demo directory to determine the path to the
% document invoke:
%
% >> which demo_req_1.abc

%  Copyright 1984-2005 The MathWorks, Inc.
%  $Revision: 1.1.4.3 $  $Date: 2007/01/21 11:56:15 $

    % Create a default (blank) requirement link type
    linkType = ReqMgr.LinkType;
    linkType.Registration = mfilename;

    % Label describing this link type
    linkType.Label = 'ABC  file (for demonstration)';

    % File information
    linkType.IsFile = 1;
    linkType.Extensions = {'.abc'};

    % Location delimiters
    linkType.LocDelimiters = '>@';
    linkType.Version = '';             % not needed

    % Uncomment the functions that are implemented below
```

```
    linkType.NavigateFcn = @NavigateFcn;
    linkType.ContentsFcn = @ContentsFcn;


function NavigateFcn(filename,locationStr)
  if ~isempty(locationStr)
      findId=0;
      switch(locationStr(1))
      case '>'
          lineNum = str2num(locationStr(2:end));
          openFileToLine(filename, lineNum);
      case '@'
          openFileToItem(filename,locationStr(2:end));
      otherwise
          openFileToLine(filename, 1);
      end
  end


function openFileToLine(fileName, lineNum)
  if lineNum > 0
      err = javachk('mwt', 'The MATLAB Editor');
      if isempty(err)
         editor = com.mathworks.mlservices.MLEditorServices;
         editor.openDocumentToLine(fileName, lineNum);
      end
  else
      edit(fileName);
  end


function openFileToItem(fileName, itemName)
  reqStr = ['Requirement:: "' itemName '"'];
  lineNum = 0;
  fid = fopen(fileName);
  i   = 1;
  while lineNum == 0
      lineStr = fgetl(fid);
      if ~isempty(strfind(lineStr, reqStr))
          lineNum = i;
```

```
        end;
        if ~ischar(lineStr), break, end;
        i = i + 1;
    end;
    fclose(fid);
    openFileToLine(fileName, lineNum);


function [labels, depths, locations] = ContentsFcn(filePath)
    % Read the entire M-file into a variable
    fid = fopen(filePath,'r');
    contents = char(fread(fid)');
    fclose(fid);

    % Find all the requirement items
    fList1 = regexpi(contents,'\nRequirement:: "(.*?)"','tokens');

    % Combine and sort the list
    items = [fList1{:}]';
    items = sort(items);
    items = strcat('@',items);

    if (~iscell(items) && length(items)>0)
       locations = {items};
       labels = {items};
    else
       locations = [items];
       labels = [items];
    end

    depths = [];
```

2 To register the custom link type ABC, type the following MATLAB command:

```
rmi register rmicustabcinterface
```

The ABC file type appears on the drop-down list of document types in the Requirements dialog box.

3 Create a text file with the `.abc` extension containing several requirement items marked by the `Requirement::` string. For your convenience, an example file is included in the installation. It is named `demo_req_1.abc` and is located in *`matlabroot`*`\toolbox\slvnv\rmidemos`. `demo_req_1.abc` contains the following content:

```
Requirement:: "Altitude Climb Control"

Altitude climb control is entered whenever:
|Actual Altitude- Desired Altitude | > 1500

Units:
Actual Altitude - feet
Desired Altitude - feet

Description:

When the autopilot is in altitude climb
control mode, the controller maintains a
constant user-selectable target climb rate.

The user-selectable climb rate is always a
positive number if the current altitude is
above the target altitude.  The actual target
climb rate is the negative of the user
setting.

<END "Altitude Climb Control">



Requirement:: "Altitude Hold"

Altitude hold mode is entered whenever:
|Actual Altitude- Desired Altitude | <
  30*Sample Period*(Pilot Climb Rate / 60)
```

```
Units:
Actual Altitude - feet
Desired Altitude - feet
Sample Period - seconds
Pilot Climb Rate - feet/minute

Description:

The transition from climb mode to altitude
hold is based on a threshold that is
proportional to the Pilot Climb Rate.

At higher climb rates the transition occurs
sooner to prevent excessive overshoot.

<END "Altitude Hold">



Requirement:: "Autopilot Disable"

Altitude hold control and altitude climb
control are disabled when autopilot enable
is false.

Description:

Both control modes of the autopilot
can be disabled with a pilot setting.

<END "Autopilot Disable">



Requirement:: "Glide Slope Armed"

Glide Slope Control is armed when Glide
Slope Enable and Glide Slope Signal
are both true.
```

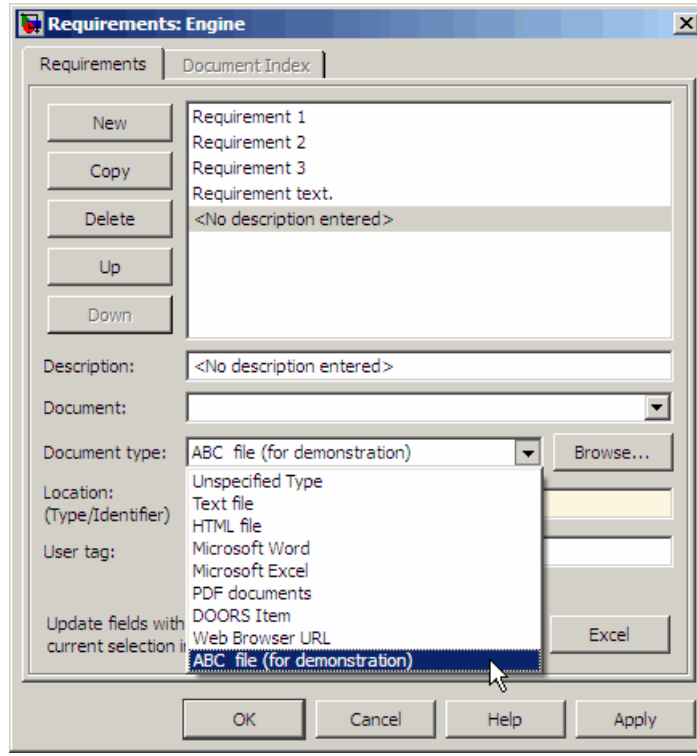
```
Units:
Glide Slope Enable - Logical
Glide Slope Signal - Logical

Description:

ILS Glide Slope Control of altitude is only
enabled when the pilot has enabled this mode
and the Glide Slope Signal is true.  This indicates
the Glide Slope broadcast signal has been
validated by the on board receiver.

<END "Glide Slope Armed">



Requirement:: "Glide Slope Coupled"

Glide Slope control becomes coupled when the control
is armed and (Glide Slope Angle Error > 0) and
 Distance < 10000

Units:
Glide Slope Angle Error - Logical
Distance - feet

Description:

When the autopilot is in altitude climb control
mode the controller maintains a constant user
selectable target climb rate.

The user-selectable climb rate is always a positive
number if the current altitude is above the target
altitude the actual target climb rate is the
negative of the user setting.

<END "Glide Slope Coupled">
```

4 Open the model `my_sf_car`.

5 Right-click the Engine block and select **Requirements > Edit/Add Links**.

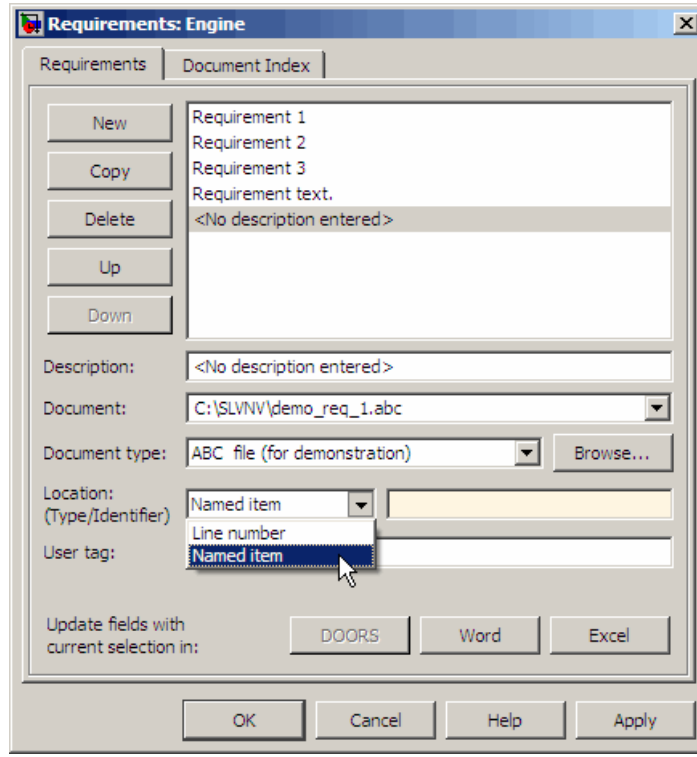
The Requirements dialog box opens.

6 Click **New** to add a new default requirement. The ABC file type is now available in the **Document type** drop-down list.



7 Set **Document type** to `ABC file (for demonstration)` and browse to the `demo_req_1.abc` file, or to the `.abc` requirements file that you created in step 3. The browser shows only the files with the `.abc` extension.

8 Define a particular location in the document. In this example, use a line number or a requirement name as the item identifier, so that the location delimiters in the `rmicustabcinterface` function are specified as '>@'. As a result of this parameter, the **Location** drop-down menu contains these two items whenever the document type is set to ABC file.



Creating a Document Index

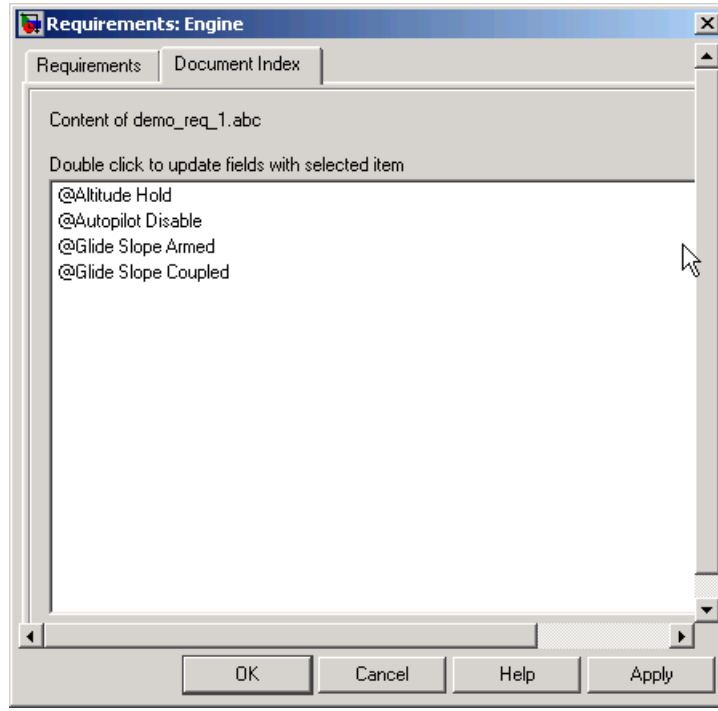
The example file format clearly defines the listed requirement items. To generate a document index, set the `ContentsFcn` to a valid function. The MATLAB M-code uses file I/O functions to read the contents into a MATLAB variable. The RMI uses the regular expression utility in the MATLAB software to extract a list of requirement items that it returns.

The following code generates an index for the ABC files.

```
function [labels, depths, locations] = ContentsFcn(filePath)
  % Read the entire M-file into a variable
  fid = fopen(filePath,'r');
  contents = char(fread(fid)');
  fclose(fid);
```

```
% Find all the functions
fList1 = regexpi(contents,'\nRequirement:: "(.*?)"','tokens');

% Combine and sort the list
items = [fList1{:}]';
items = sort(items);
items = strcat('@',items);

locations = [items];
labels = [items];
depths = [];
```

For example, for the `demo_req_1.abc` file described in "Creating a Custom Link Requirement Type" on page 2-34, this function generates the document index as shown in the following illustration.



Navigating to Simulink Models from External Documents

The RMI includes several functions that simplify creating navigation interfaces in external documents. The external application that displays your document must support an application programming interface (API) for communicating with the MATLAB software.

Providing Unique Object Identifiers

Whenever you create a requirement link for a Simulink or Stateflow object, a globally unique identifier is created for that object. This identifier identifies the object and does not change if you rename or move the object, or add or delete requirement links. The unique identifier is only used to resolve an object within a model. The identifier is globally unique and does not collide with identifiers in other models, unless the two models derive from the same original model. Unique object identifiers have formats such as:`GIDa_cd14afcd_7640_4ff8_9ca6_1490 4bdf2f0f`.

Using the rmiobjnavigate Function

The `rmiobjnavigate` function identifies the appropriate Simulink or Stateflow object, highlights that object, and brings the appropriate editor window to the front of the screen. When you navigate to a Simulink model from an external application, invoke this function. Internally this function creates a table of all the unique object identifiers within a model, which is used for efficient object lookup.

The first time you navigate to an item in a particular model, you might experience a slight delay while the software constructs the internal navigation table. You do not experience a long delay on subsequent navigation.

Determining the Navigation Command

Once you create a requirement link for a Simulink or Stateflow object, use the `rmi` function at the MATLAB prompt to find the appropriate navigation command string. The return value of the `navCmd` method is a string that navigates to the correct object when evaluated by the MATLAB software:

```
cmdString = rmi('navCmd', block);
```

Send this exact string to the MATLAB software for evaluation as part of navigating from the external application to the Simulink model.

Using the ActiveX Navigation Control

The Simulink Verification and Validation software includes a special Microsoft ActiveX control. You use this control to enable linking to Simulink models from Microsoft Word and Excel documents. You use this same control in any other application that supports ActiveX within its documents.

The control is derived from a push button and has the Simulink icon. There are two instance properties that define how the control works. The `tooltipstring` property is the string that is displayed in the ToolTip of the control. The `MLEvalCmd` property is the string that you pass to the MATLAB software for evaluation when the control is pushed.

Typical Code Sequence for Establishing Two-Way Links

When you create an interface to an external tool, you can automate the procedure for establishing links so that you do not need to update the dialog box fields manually. This type of automation is part of the selection-based linking that is implemented for certain built-in types, such as Microsoft Word and Excel documents.

To automate the procedure for establishing links:

1. Select a Simulink or Stateflow object and an item in the external document.
2. Invoke the link creation action either from a Simulink menu or command, or a similar mechanism in the external application.
3. Identify the document and current item using the scripting capability of the external tool. Pass this information to the MATLAB software and create a requirement link on the selected object using `rmi('createempty')` and `rmi('cat')`.
4. Determine the MATLAB navigation command string that you must embed in the external tool using the `navCmd` method:

   ```
   cmdString = rmi('navCmd',obj)
   ```

5 Create a navigation item in the external document using the scripting capability of the external tool. Set the MATLAB navigation command string in the appropriate property.

As an example of this type of automation, you can use the code for selection-based linking to the MicrosoftWord and Excel, and Telelogic DOORS software. The files are contained in *matlabroot*\toolbox\slvnv\reqmgt\private:

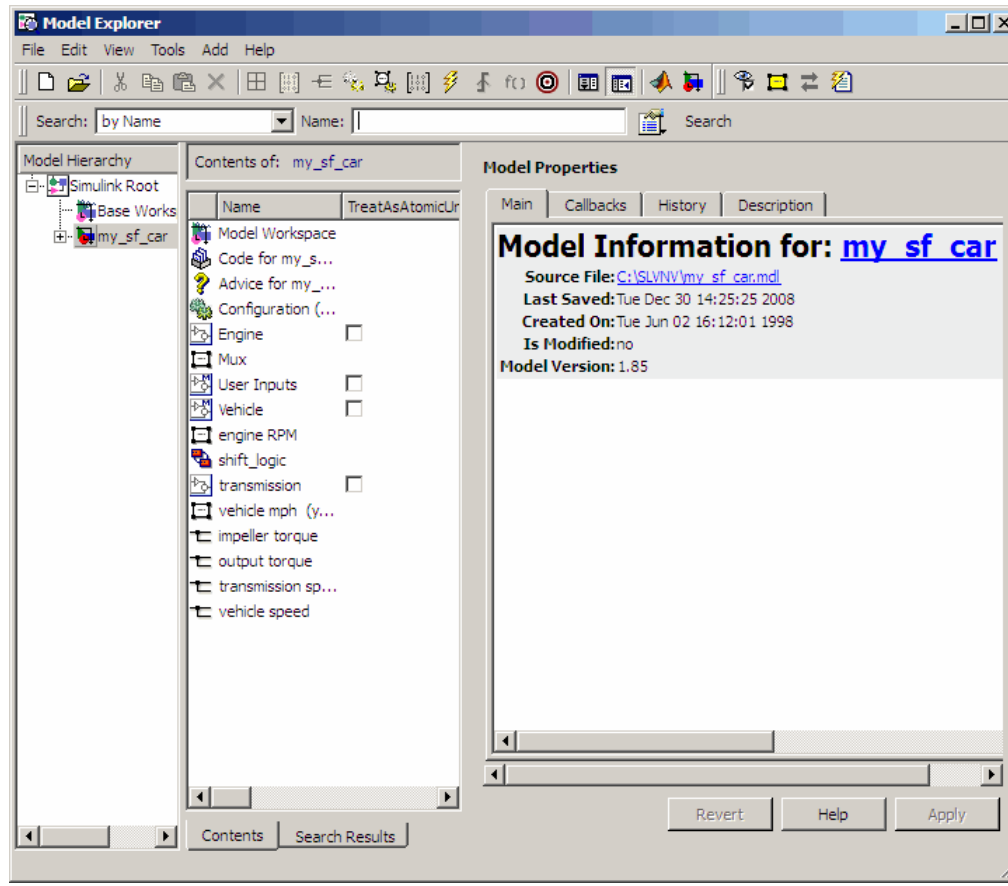
```
selection_link_doors.m
selection_link_excel.m
selection_link_word.m
```

Viewing Objects with Requirement Links

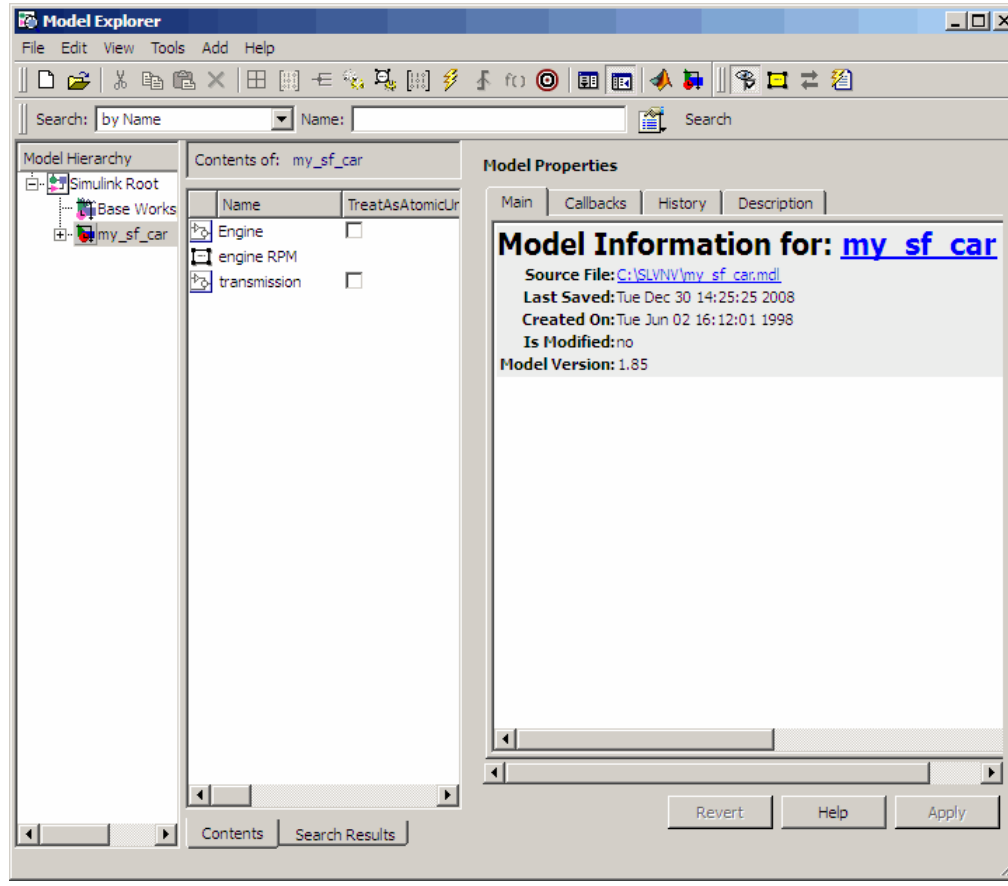
After you add requirements to blocks in a model, you can change the view in the Model Explorer to show only objects that have associated requirements:

1. Open the model, `my_sf_car`.
2. Select **View > Model Explorer**.

   The model and its elements appear in the Model Explorer window.



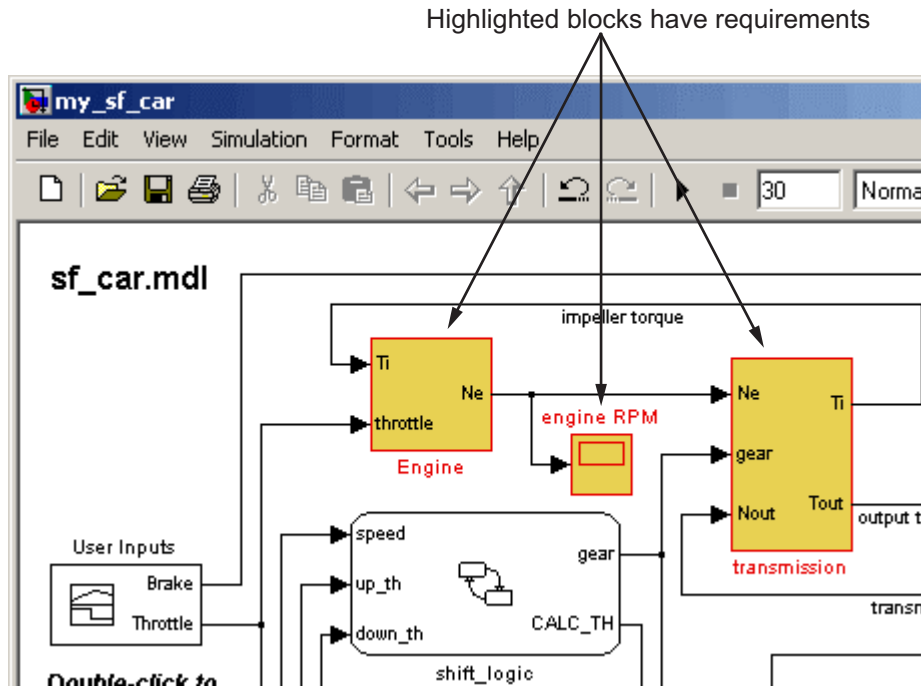
3 In the **Model Explorer** toolbar, select the Display Objects with Linked Requirements tool  .



4 To see all objects, click the Display Objects with Linked Requirements tool again.

You can highlight blocks of a Simulink model with associated requirements in a Simulink model window as follows:

1 In the **Model Explorer** toolbar, select the Highlight Items with Requirements on Model tool .



2 To clear the highlighting, click the Highlight Items with Requirements in Model tool again.

Generating a Requirements Report

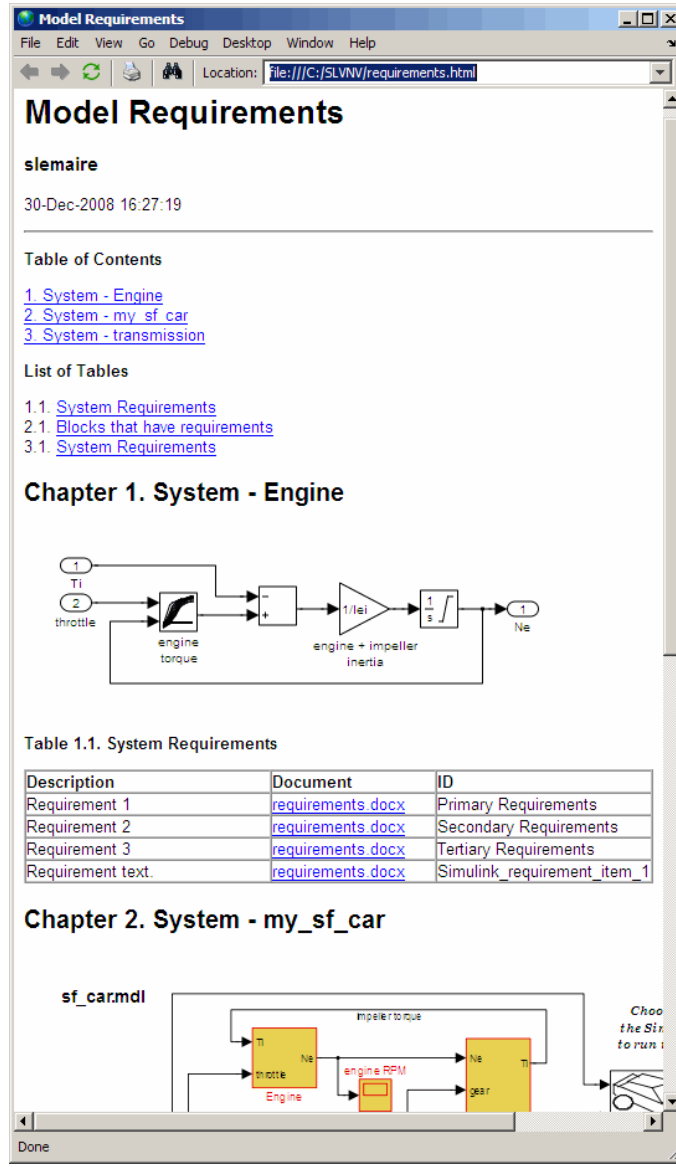
After you add requirements to a model, you can generate a report on all the requirements associated with the model and its blocks.

To generate a requirements report for the `my_sf_car` model:

1. Open the `my_sf_car`model.
2. Select

   Tools > Requirements > Generate Report.

   The RMI searches through all the blocks and subsystems in the model for associated requirements. It generates a complete report in HTML format with the default name `requirements.html`and displays it in your system Web browser.


Model Requirements
File Edit View Go Debug Desktop Window Help
Location: file:///C:/SLVNV/requirements.html
Model Requirements
slemaire
30-Dec-2008 16:27:19
Table of Contents
1. System - Engine
2. System - my_sf_car
3. System - transmission
List of Tables
1.1. System Requirements
2.1. Blocks that have requirements
3.1. System Requirements
Chapter 1. System - Engine
1
Ti
2
throttle
engine torque
1/Iei
engine + impeller inertia
1
Ne
Table 1.1. System Requirements
Description | Document | ID
Requirement 1 | requirements.docx | Primary Requirements
Requirement 2 | requirements.docx | Secondary Requirements
Requirement 3 | requirements.docx | Tertiary Requirements
Requirement text. | requirements.docx | Simulink_requirement_item_1
Chapter 2. System - my_sf_car
sf_car.mdl
impeller torque
Ti
throttle
Ne
Engine
engine RPM
Ne
gear
Ti
Done

Displaying the System Requirements in a Diagram

In this section...

About the System Requirements Block

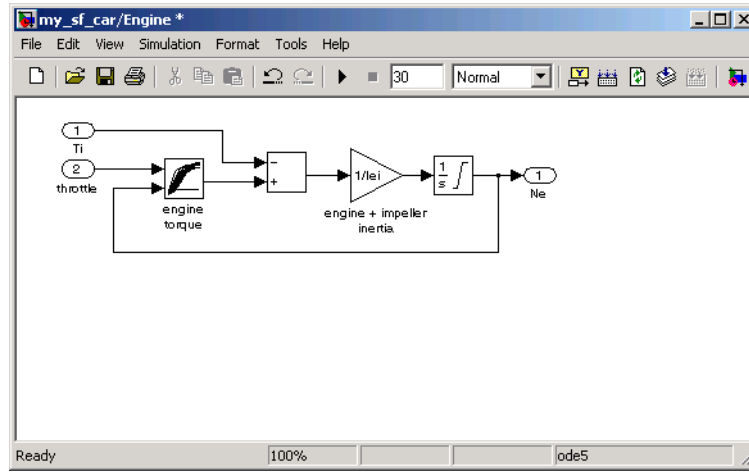
You can list all the requirements for a model or a subsystem directly on the Simulink diagram. You list the requirements by adding the System Requirements block from the Simulink Verification and Validation library to the diagram. You can place this block anywhere in a diagram. It is not connected to other Simulink blocks.

After you place the System Requirements block in a Simulink diagram, the requirements associated with the model or subsystem are automatically listed. Requirements associated with individual blocks in the diagram are not listed.

Adding the System Requirements Block

In "Adding Requirement Links to an Object" on page 2-7, you added requirement links to the Engine block of the model, `my_sf_car`. You can list these requirements in the block diagram of the Engine subsystem as follows:

1. Open the `my_sf_car`model.
2. Double-click the Engine block. The Engine subsystem diagram opens.



3 Click the Library Browser tool  .

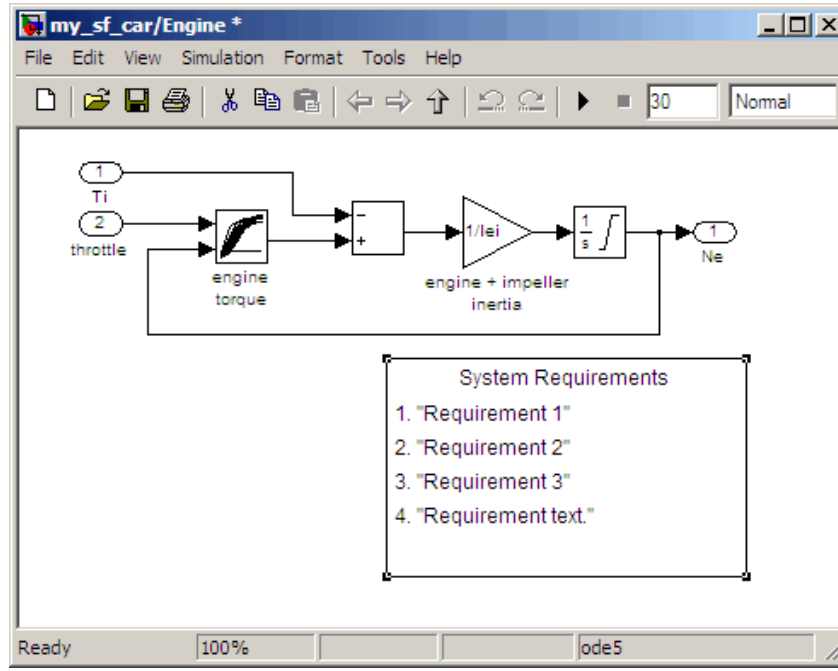
The Simulink Library Browser opens.

4 In the left pane of the Simulink Library Browser, select **Simulink Verification and Validation**.

The Simulink Verification and Validation library opens in the right pane of the Simulink Library Browser. It contains one block, System Requirements.

5 Drag the System Requirements block to an empty space in the Engine diagram.

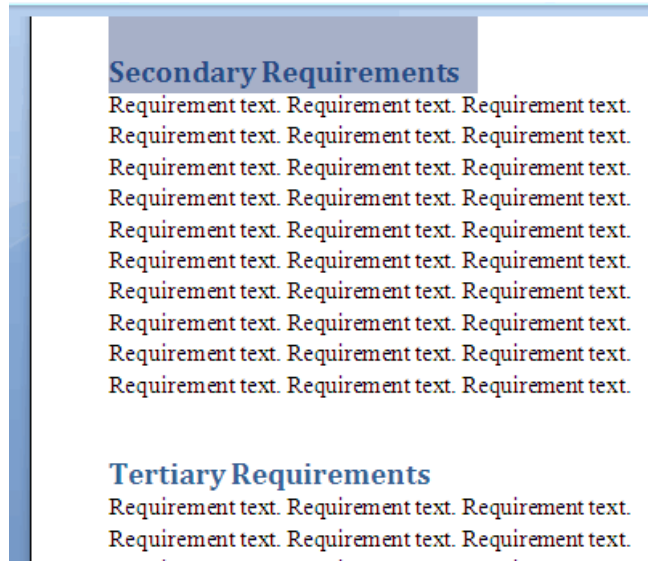
The block is automatically populated with the system requirements for the Engine diagram.



6 The System Requirements block automatically lists all the system requirements for the current model or subsystem. You cannot have more than one System Requirements block in a diagram.

Each of the listed requirements is an active link to the actual requirements document. For example, to access the document for the second requirement link, double-click `Requirement 2`.

The document, `requirements.docx`, opens in the Microsoft Word editor at the highlighted first occurrence of the text “Secondary Requirements”.

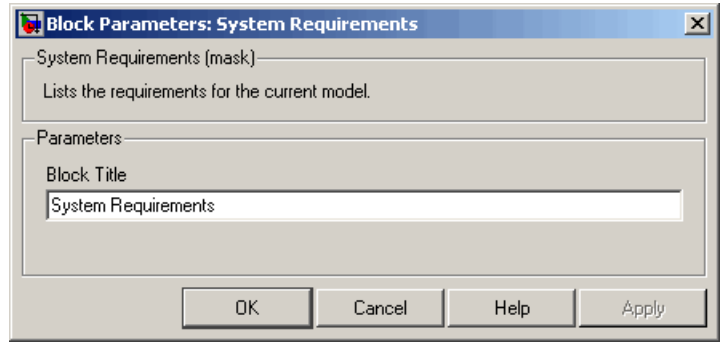


If a diagram includes a System Requirements block, that block automatically updates the listing as you add, modify, or delete requirements for the model or subsystem.

Renaming the System Requirements Block

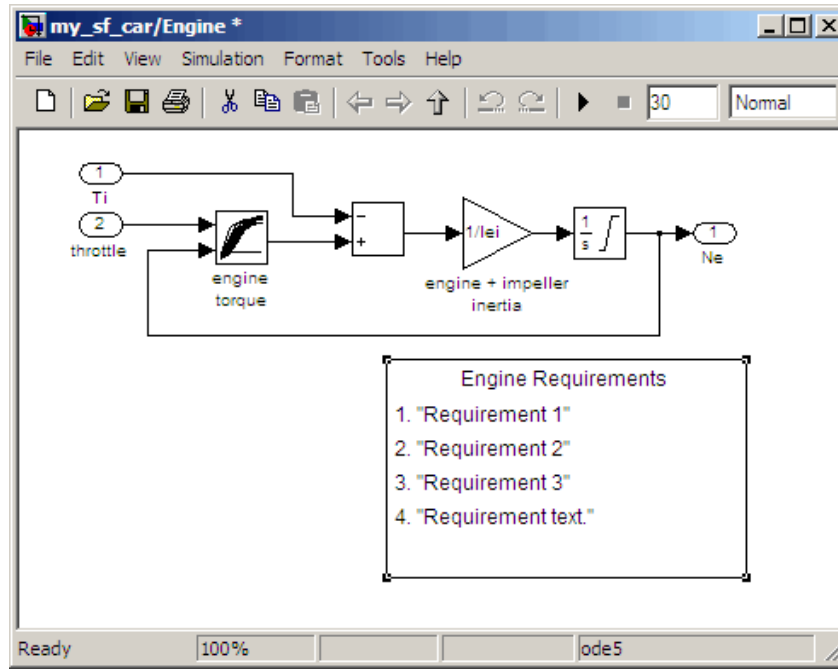
The list of the system requirements in a diagram appears under a default heading, `System Requirements`. You can change the heading by renaming the System Requirements block in the diagram:

1. Right-click the System Requirements block in the `my_sf_car/Engine` diagram.
2. Select **Mask Parameters**. The Block Parameters dialog box opens.



3 Type `Engine Requirements` in the **Block Title** field and click **OK**.

The new block heading updates in the diagram.

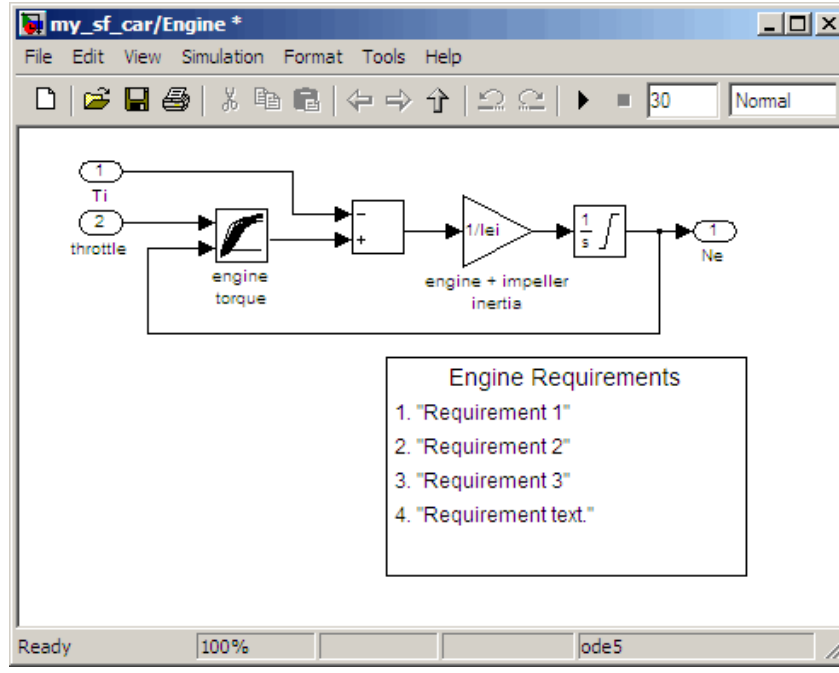


Changing Fonts for the System Requirements Block

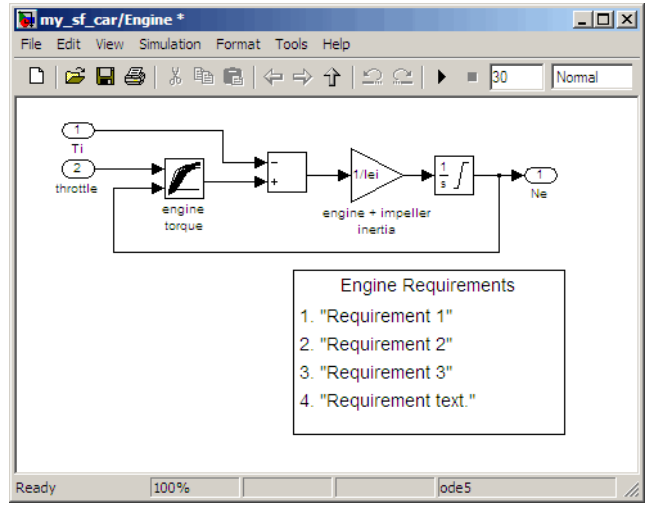
The System Requirements block is implemented using a set of empty subsystems. Occasionally, the appearance does not refresh correctly, for example, when you change the font style or size. You can easily fix this by double-clicking the top label for the block, which causes the entire block display to refresh.

To change the font used in the block:

1. Right-click the System Requirements block (now named Engine Requirements) in the `my_sf_car/Engine` diagram.
2. Select **Format > Font**. The Set Font dialog box opens.
3. Under **Size**, select 14, then click **OK**. The block display partially refreshes.



4. To refresh the entire block display, double-click the top label, `Engine Requirements`.


my_sf_car/Engine *
File Edit View Simulation Format Tools Help
30
Normal
1
Ti
2
throttle
engine
torque
1/Iei
engine + impeller
inertia
1
Ne
Engine Requirements
1. "Requirement 1"
2. "Requirement 2"
3. "Requirement 3"
4. "Requirement text."
Ready
100%
ode5

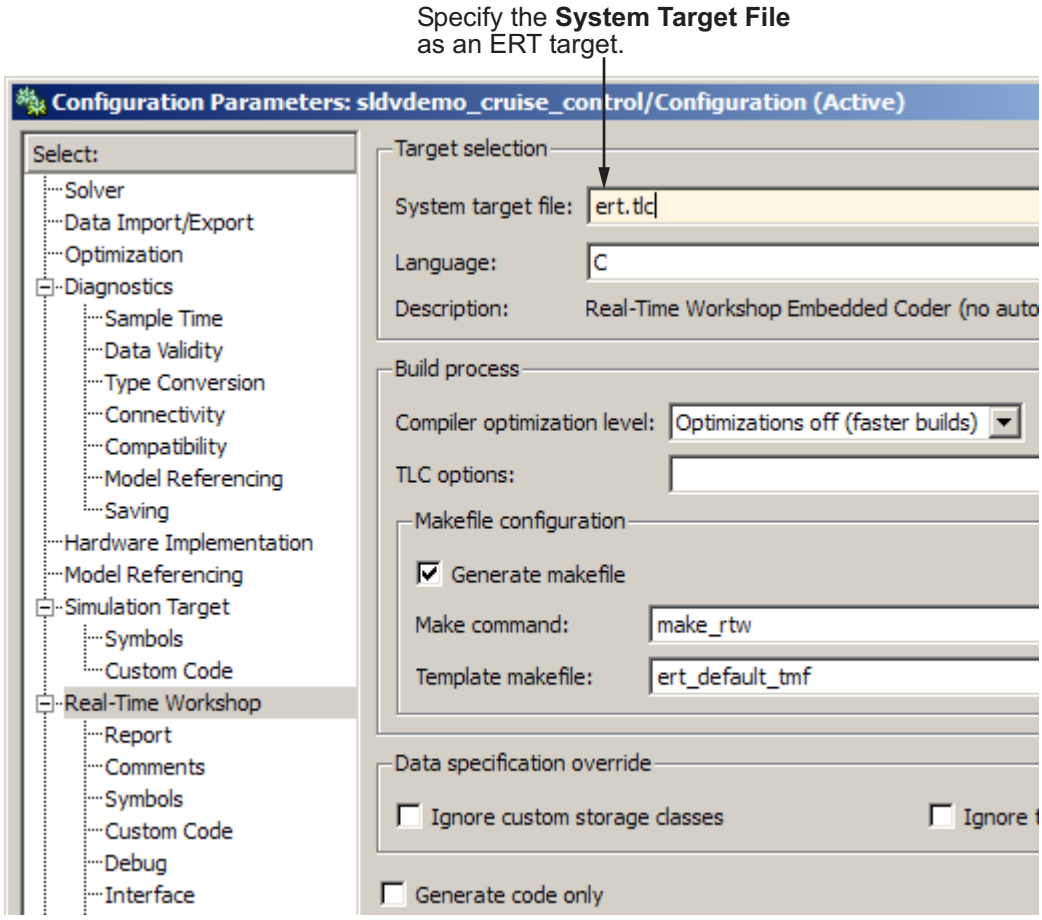
Including Requirements with Generated Code

After you finish simulating your model and verifying its performance against the requirements, consider using it to generate code for an embedded real-time application. The Simulink Verification and Validation software can include the requirements that you assign to Simulink blocks in generated code for Embedded Real-Time (ERT) targets of the Real-Time Workshop® Embedded Coder™ software.

To specify that the generated code of an ERT target include requirements:

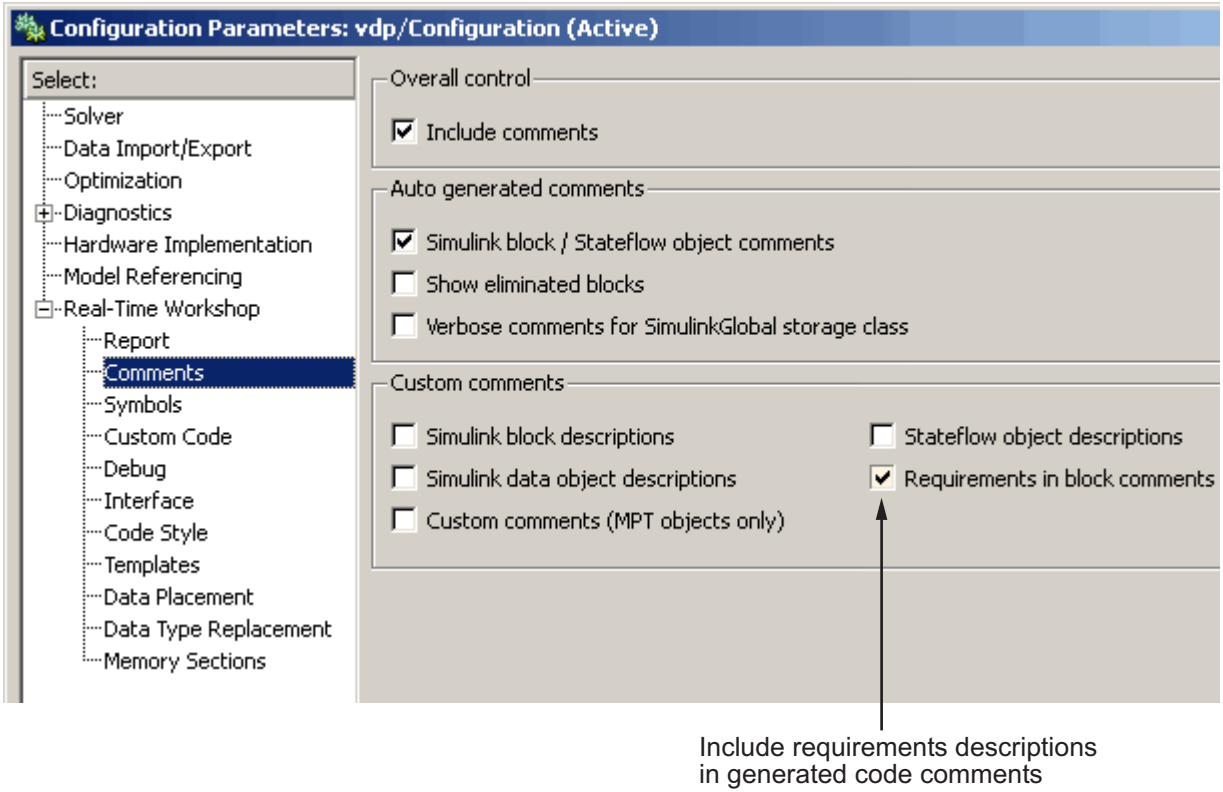
1. Open your model.
2. Select **Simulation > Configuration Parameters**.
3. In the **Select** pane of the Configuration Parameters dialog box, select the **Real-Time Workshop** node.

   The currently configured system target must be an ERT target.



4 In the **Select** pane, under **Real-Time Workshop**, select **Comments**.

5 In the **Custom comments** section on the right, select the **Requirements in block comments** check box.



Generated code includes requirement descriptions in the following locations.

Model Element	Requirement Description Location
Model	In the main header file, `<model>.h`
Nonvirtual subsystems	At the call site for the subsystem
Virtual subsystems	At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem has no nonvirtual parent, requirement descriptions are located in the main header file for the model, `<model>.h`.
Nonsubsystem blocks	In the generated code for the block

3

Managing Model Requirements with DOORS Software

The Requirements Management Interface for Telelogic DOORS software associates DOORS requirements with model objects. To learn how to use these applications together, see the following sections:

What Is the Requirements Management Interface for DOORS Software?

Telelogic DOORS software is a requirements management application that captures, tracks, and manages user requirements. The Requirements Management Interface (RMI) is a special interface between your Simulink model and the DOORS software.

Configuring the Requirements Management Interface for DOORS Software

In this section...

Before You Begin

Telelogic DOORS software is a requirements management application for capturing, tracking, and managing requirements. If you plan to use DOORS software with the Requirements Management Interface (RMI), you must install some additional files to establish communication between the DOORS and Simulink software. The sections that follow discuss installation and configuration procedures for a variety of situations.

Installing DOORS Software Before RMI

If DOORS software is installed before you install the RMI and run the setup script, as described in "Configuring the Requirements Management Interface" on page 2-3, no additional installation for your DOORS software is necessary. The setup script automatically copies all the necessary files to the correct location.

Installing DOORS Software After RMI

If you install DOORS software after you install the RMI, run the setup script again, as described in "Configuring the Requirements Management Interface" on page 2-3.

Upgrading DOORS Software

If you upgrade your DOORS software installation after installing the RMI, run the setup script again, as described in "Configuring the Requirements Management Interface" on page 2-3.

If you upgrade from Version 7.1 to 8.0 of the DOORS software, follow these additional steps:

1. Navigate to the directory `Telelogic\DOORS_8.0\lib\dxl\startupFiles`.
2. Open the file `copiedFromDoors7.dxl` with a text editor.
3. Comment out the line:

   ```
   #include <addins/dmi/dmi.inc>
   ```

   It should now look like this:

   ```
   //#include <addins/dmi/dmi.inc>
   ```
4. Save and close the file.
5. Start the DOORS and MATLAB software.
6. Run the setup script.

Manual Installation for DOORS Software

Normally, the setup script automatically copies all the files to the correct location. However, in some cases the script might fail because of file permissions in your DOORS software installation. If this happens, you have to manually install additional files, as described in the following procedure:

1. Close the DOORS software if it is running.
2. Copy the following files from *`matlabroot`*`\toolbox\slvnv\reqmgt` to the `<`*`doors`*`>\lib\dxl\addins` directory:

   ```
   addins.idx
   addins.hlp
   ```

<doors> represents the top-level directory where the DOORS software is installed. Replace any existing versions of the files if they have not been modified; otherwise, merge their contents.

3 Copy the following files from *matlabroot*\toolbox\slvnv\reqmgt to the <*doors*>\lib\dxl\addins\dmi directory.

```
dmi.hlp
dmi.idx
dmi.inc
runsim.dxl
selblk.dxl
```

Replace any existing versions of these files.

4 Open the file <*doors*>\lib\dxl\startup.dxl, and add the following `include` statement in the user-defined files section:

```
#include <addins/dmi/dmi.inc>
```

Starting the Requirements Management Interface for DOORS Software

Use this procedure to start the Requirements Management Interface for Telelogic DOORS software. Do this prior to synchronizing the model with the DOORS software and linking objects to DOORS requirements.

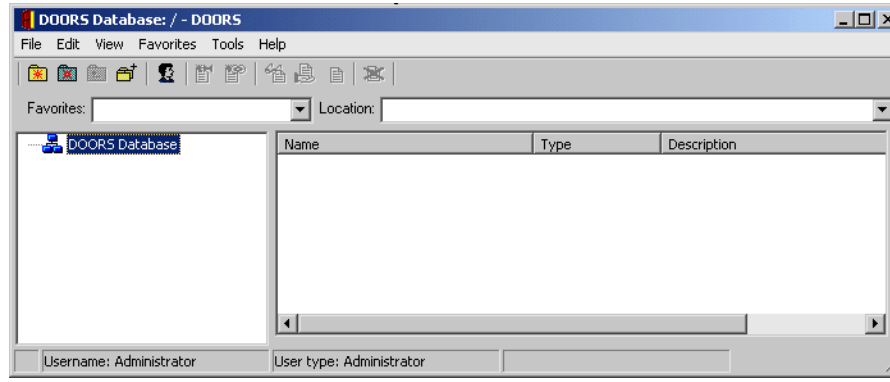
1. Start the MATLAB software on your DOS or UNIX system with the following command:

   ```
   \...\matlab.exe /automation
   ```

   The MATLAB software starts up minimized with a default *`matlabroot`*`\bin` path. This mode of operation is necessary to navigate between an object mapping in the DOORS software and its source object in the Simulink model. If this type of navigation is not needed, open your MATLAB software in default mode.

2. Start your DOORS software.

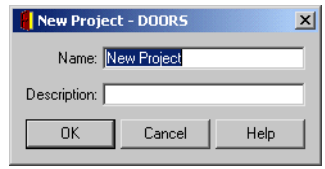
   The DOORS Database window appears.



You must have a DOORS project open in order to use the Requirements Management Interface. If you do not have a project to open, create and open one as follows:

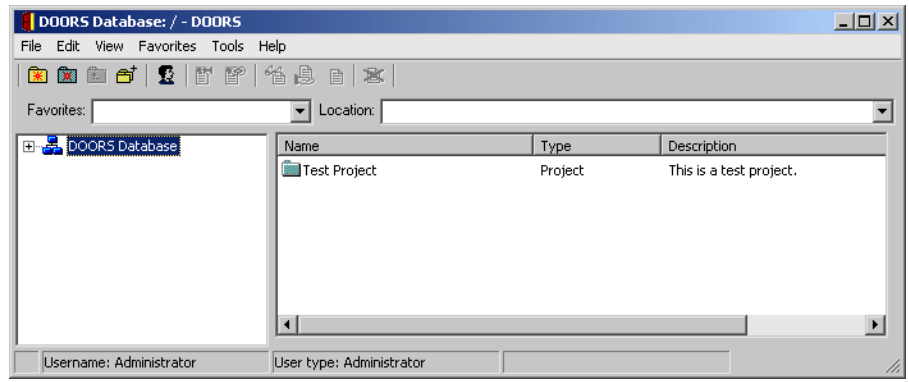
1 Right-click the `DOORS Database` node in the left pane and, from the resulting menu, select **New > Project**.

The New Project dialog box appears.

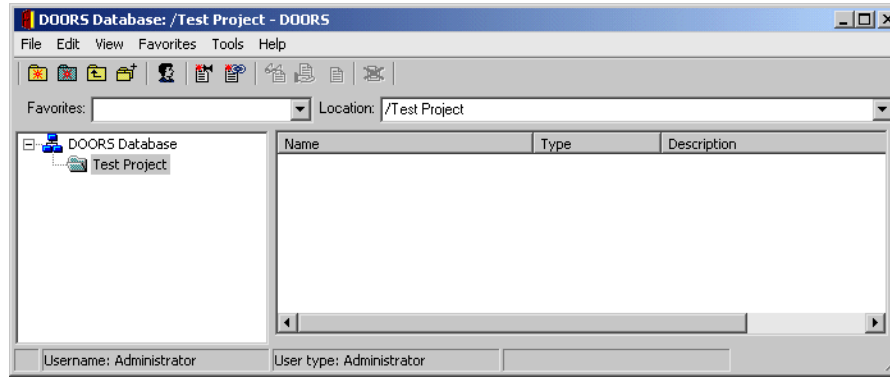


2 Enter the name `Test Project` and the description `This is a test project.` and click **OK**.

The new project appears in the right pane of the dialog box.



3 In the right pane, double-click the project to open it. The project opens.


DOORS Database: /Test Project - DOORS
File Edit View Favorites Tools Help
Favorites:
Location: /Test Project
DOORS Database
Test Project
Name
Type
Description
Username: Administrator
User type: Administrator

Linking Objects to DOORS Requirements

In this section...

About Linkages Between a Simulink Model and DOORS Software

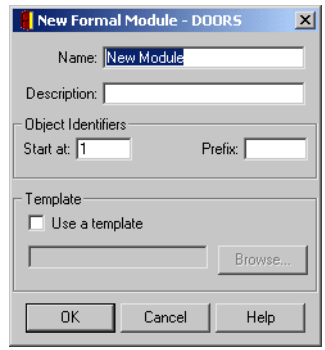
The Requirements Management Interface for Telelogic DOORS software lets you link from Simulink or Stateflow objects directly to DOORS requirements. This linking mechanism is similar to selection-based linking for Microsoft Word and Excel documents, described in "Selection-Based Linking" on page 2-20. That is, it provides two-way links by creating a special navigation object in the DOORS software, which allows you to navigate from the DOORS requirement to the associated object in the Simulink or Stateflow diagram. The sections that follow describe how to link DOORS requirements to objects in your Simulink or Stateflow diagram.

Creating a DOORS Requirement Object

Use the following procedure to create a DOORS requirement object in a formal module.

1 In the main DOORS window, from the **File** menu, select **New > Formal Module** to create a new formal module.

The New Formal Module dialog box appears.



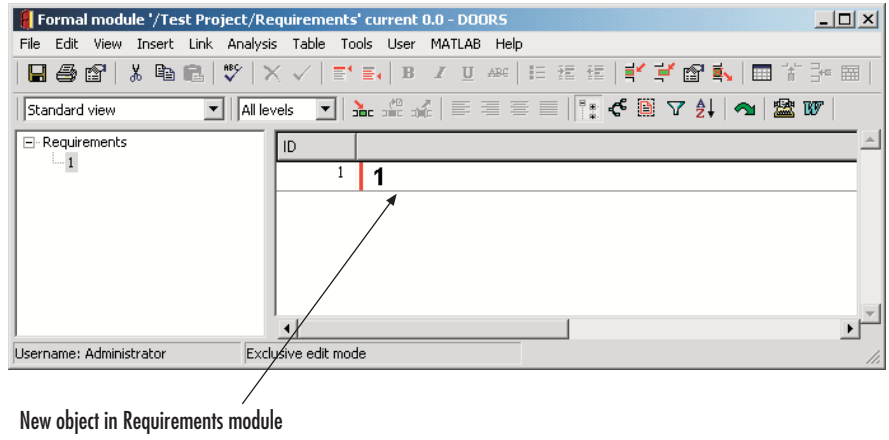
2 In the **Name** text field, enter the name `Requirements` and click **OK**.

The new formal module `Requirements` appears listed in the main DOORS window. A window for `Requirements` is already open, but not in focus.

3 In the main DOORS window, double-click the `Requirements` module to bring it in focus.

4 In the formal module window `Requirements`, select **Insert > Object**.

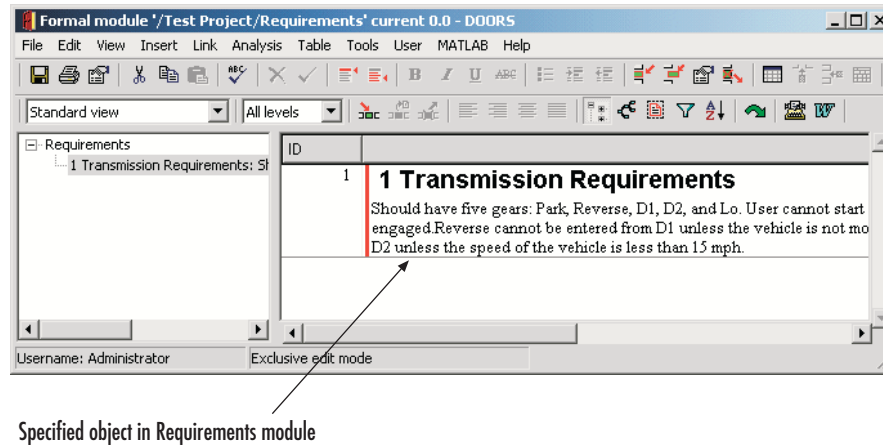
A new object appears in the formal module.



5 Right-click the object in the right pane and, from the resulting context menu, select **Properties**.

6 In the resulting Object properties dialog box, enter the **Heading** `Transmission Requirements`, some text for **Object Text**, and select **OK**.

You should now see an object similar to the following in the `Requirements` formal module.



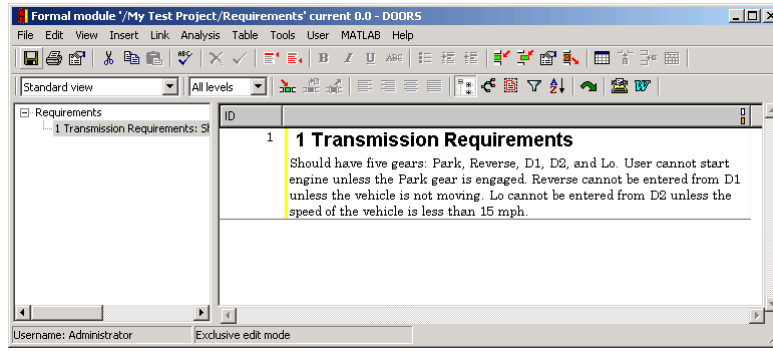
Linking a Simulink or Stateflow Object to a DOORS Requirement

In "Creating a DOORS Requirement Object" on page 3-9, you created a `Transmission Requirements` object in the `Requirements` formal module in the DOORS software. Now use the following procedure to link the transmission block in the `sf_car` model to this DOORS requirement.

1 In the MATLAB Command Window, type `sf_car` at the MATLAB prompt to open the demo model `sf_car.mdl`.

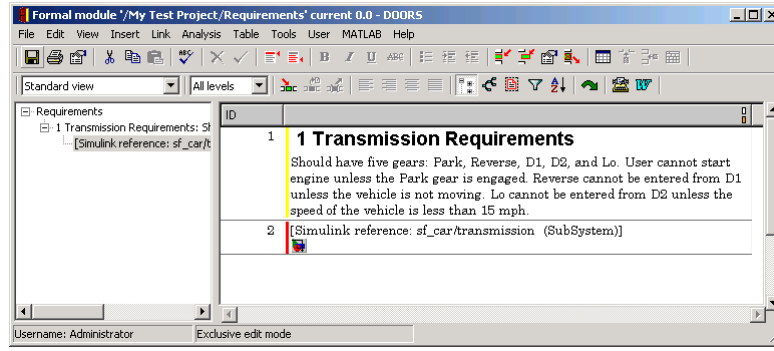
2 In the formal module window `Requirements`, select the Transmission Requirement node in the left pane.

Note *Always* select the requirement node by moving the mouse and left-clicking. Do *not* use the up and down arrow keys on the keyboard to select a requirement node.



3 In the Simulink diagram, right-click the transmission block and, from the resulting pop-up menu, select **Requirements > Add link to current DOORS object**.

The Requirements Management Interface adds the link to the DOORS requirement object.



4 Save the DOORS module.

5 Save the Simulink model as `sf_car_doors.mdl`.

The Requirements Management Interface uses the DOORS absolute number and the unique module number to identify items in the DOORS software. This ensures that the correct item is identified even if the module is renamed or the items in the module are rearranged.

You can also use the Requirements dialog box to create links to DOORS objects. Set the **Document type** field to `DOORS Item` and click **Browse**. The Requirements Management Interface opens the DOORS database. Browse to the desired module and specify the DOORS item number.

Synchronizing a DOORS Module with the Simulink Model

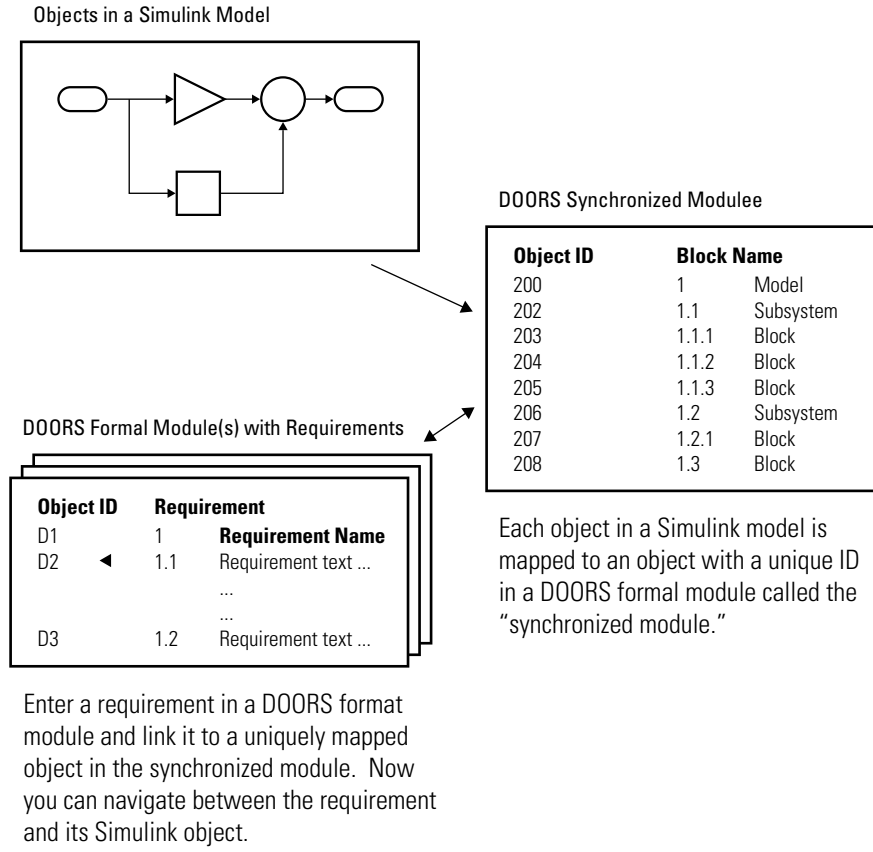
In this section...

About Module Synchronization

The sections that follow show you how to create a synchronized module and link objects with requirements in the Telelogic DOORS software. Keep in mind the following synchronization rules:

- Synchronization is optional.
- You can create requirement links before or after you synchronize, in any order.
- The synchronized module captures requirement information from the model into the DOORS database, enabling further analysis and reporting.

The following diagram illustrates the synchronization process.



Note The Requirements Management Interface and DOORS software both use the term *object*, but each uses the term differently. In the Requirements Management Interface, and in this document, the term object refers to a Simulink model, a Simulink block, a Stateflow block, and elements of a Stateflow diagram. In the DOORS software, object refers to each numbered element in the synchronized formal module for the objects in a Simulink model. The DOORS software assigns each of these objects a unique object identifier. In this document, these objects are referred to as DOORS objects.

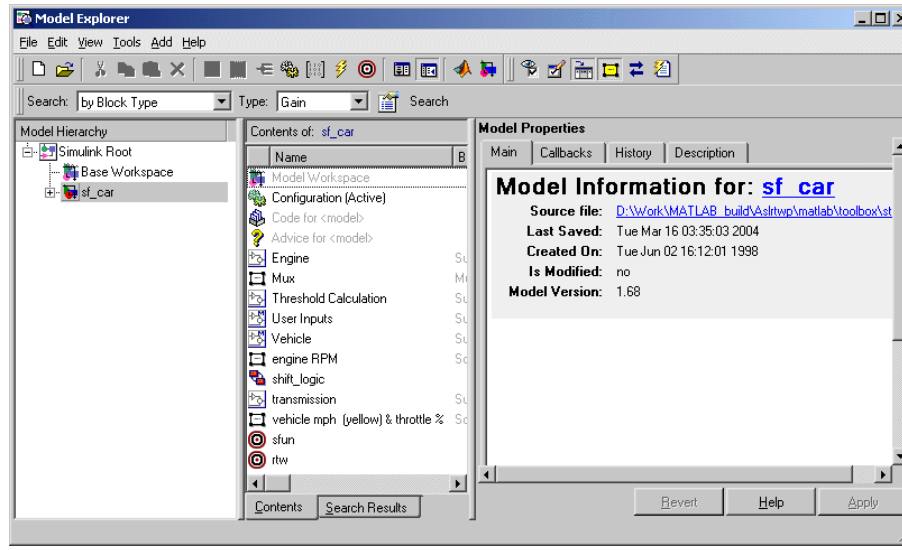
Synchronizing a Model with the DOORS Software

In "Starting the Requirements Management Interface for DOORS Software" on page 3-6, you start the MATLAB software, start the DOORS software, and open a DOORS project. Begin the process of mapping DOORS requirements to a Simulink model by first synchronizing the model with the open DOORS project. Synchronization maps a hierarchical representation of a Simulink model's blocks and Stateflow objects to a formal module in a DOORS project. Later, you use this formal module to add requirements.

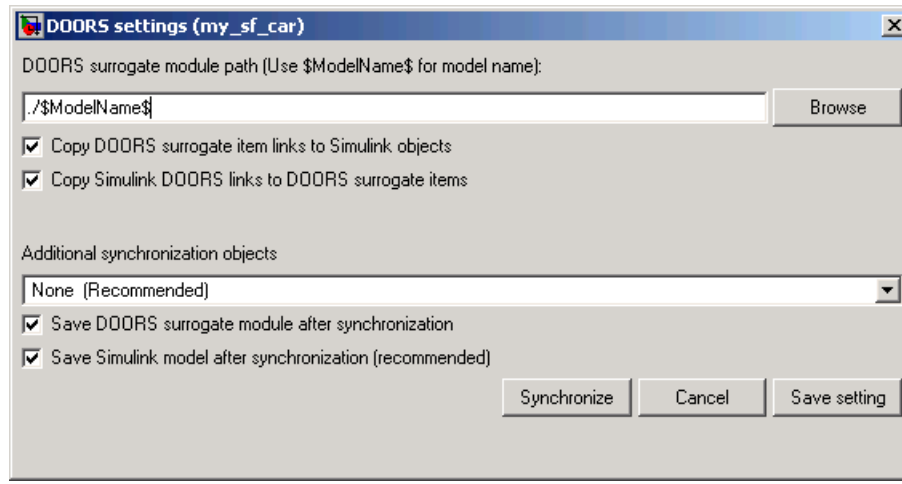
Use the following procedure to synchronize a Simulink model with the DOORS software:

1. In the MATLAB Command Window, type `sf_car` at the MATLAB prompt to open the demo model `sf_car.mdl`.

2. In the Simulink model, from the **View** menu, select **Model Explorer**.

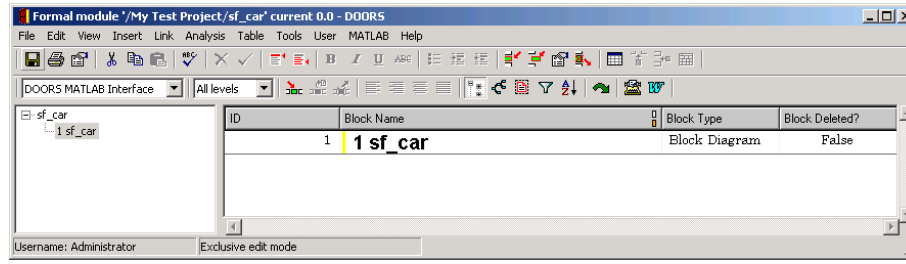


3. Select the Synchronize Requirements with DOORS tool  in the Model Explorer window. The DOORS settings dialog box opens.



4 Click **Synchronize**.

Synchronizing creates and opens a DOORS formal module for the model.



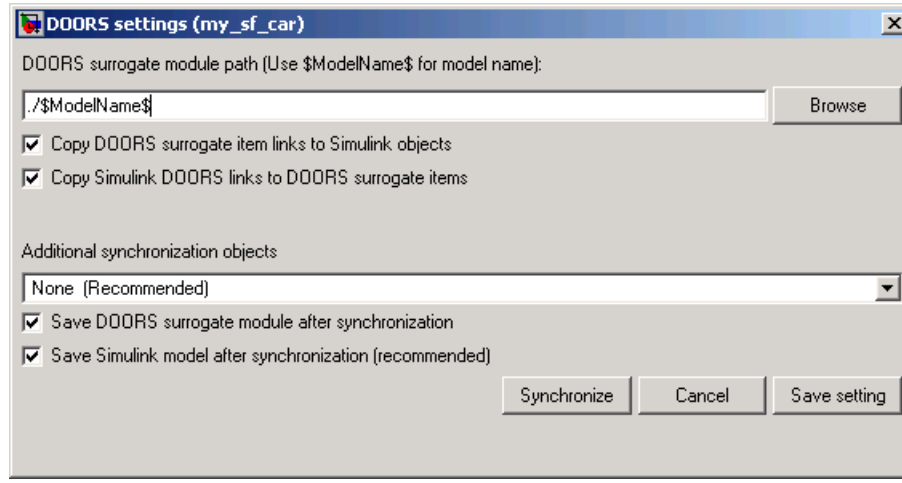
Notice that by default the DOORS formal module contains only one synchronized object, which corresponds to the top-level diagram. To include all the model blocks in the DOORS formal module, use the following procedure, “Customizing the Level of Synchronization Detail” on page 3-17.

Customizing the Level of Synchronization Detail

The DOORS surrogate module always contains the model objects that have DOORS requirement links and objects that were previously synchronized. You can choose a desired detail level to make the surrogate better reflect the model. Additional synchronization objects improve the surrogate detail at the expense of slower synchronization.

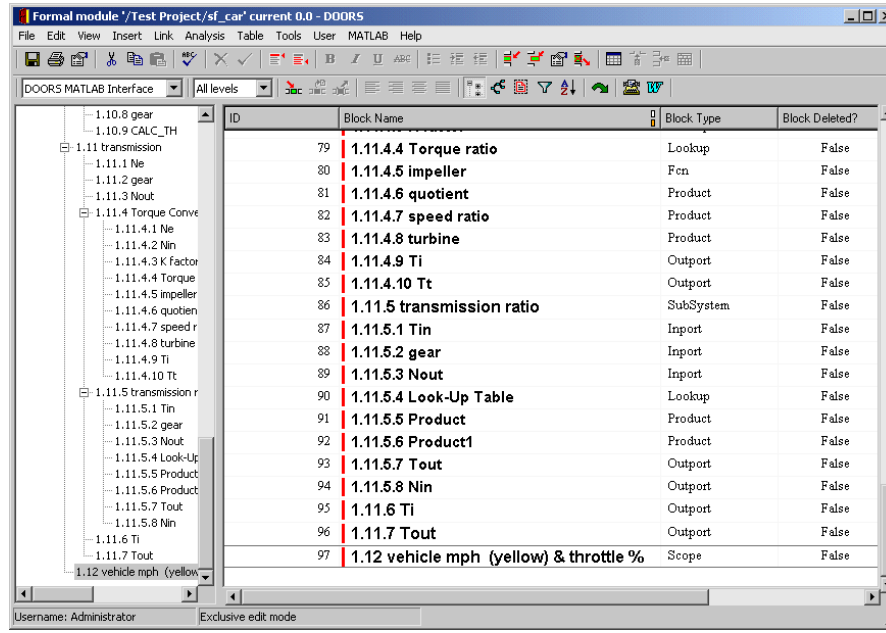
To include all the model blocks in the DOORS formal module, use the following procedure.

1. Open the `sf_car` model.
2. From the **Tools** menu in the Simulink window select **Requirements > Synchronize with DOORS**. The DOORS settings dialog box opens.



   Another way to access this dialog box is to select the Synchronize Requirements with DOORS tool  in the Model Explorer window.

3. From the drop-down list in the **Additional synchronization objects** pane, select `Complete  All blocks, subsystems, states, and transitions`.
4. Click **Synchronize**. The DOORS formal module for the model appears.

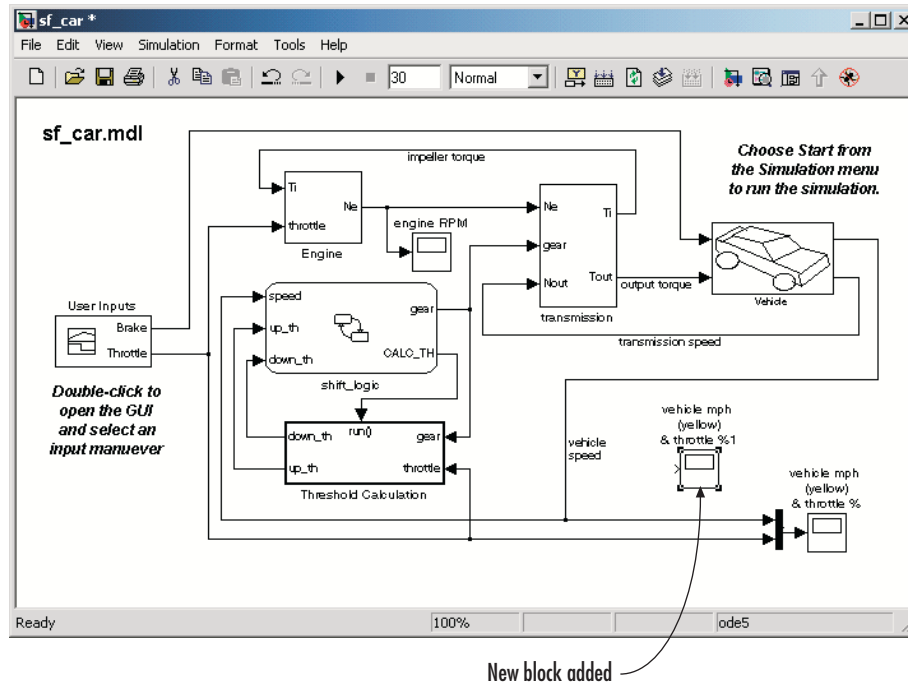


Notice the following:

- The formal module is named `sf_car` in the title bar, after the model.
- The left pane displays a node for each synchronized object. All nodes are expanded and the pane is scrolled to the bottom.
- The right pane displays a DOORS object for each model object, which consists of the model object title only. It is also scrolled to the bottom.
- Each DOORS object has a unique identifier displayed in the **ID** column. For example, the identifier for the DOORS object for the Product block `turbine` in the preceding figure is 83.
- Each DOORS object has a hierarchical identifier displayed in the **Block Name** column, which represents its relationship to other objects in the `engine` model. The hierarchical identifier of each block begins with 1, the hierarchical identifier for the model `sf_car` that contains them.
- For each DOORS object, there is a **Block Type** description that identifies each object as a particular block or a subsystem.

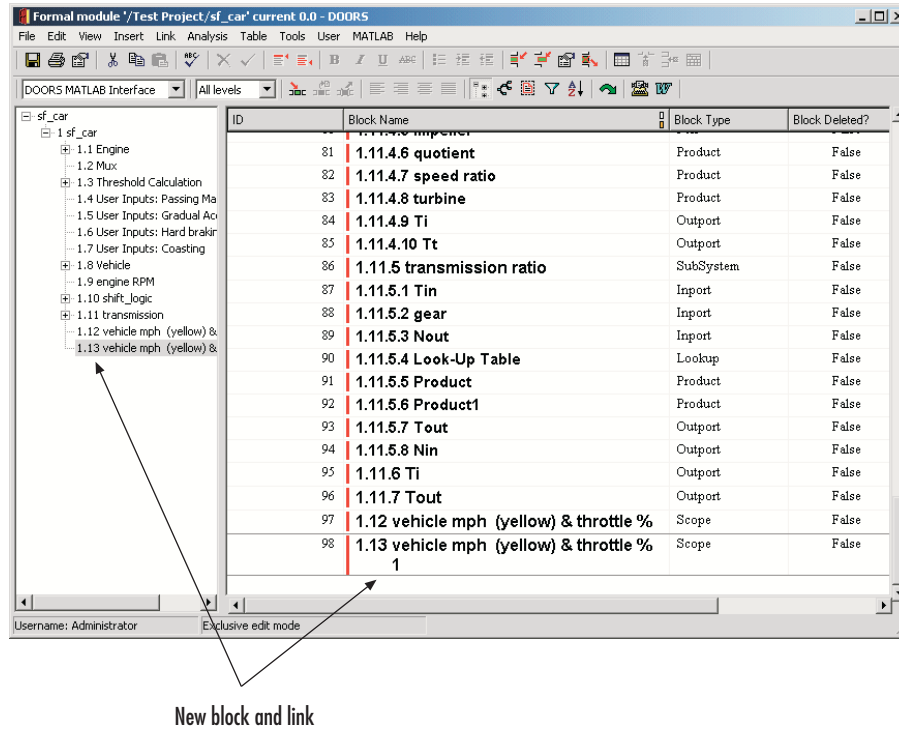
- You can add additional information columns to the right pane with the Insert Column tool in the DOORS toolbar.

5 In the Simulink model, right-click and drag a copy of the Scope block.



6 Select the Synchronize Requirements with DOORS tool again.

The synchronized module is updated with the new block.



Note The Requirements Management Interface does not detect model changes made after a synchronization. It is up to you to synchronize a changed model with the DOORS formal module.

7 In the Simulink model, delete the added Scope block and resynchronize.

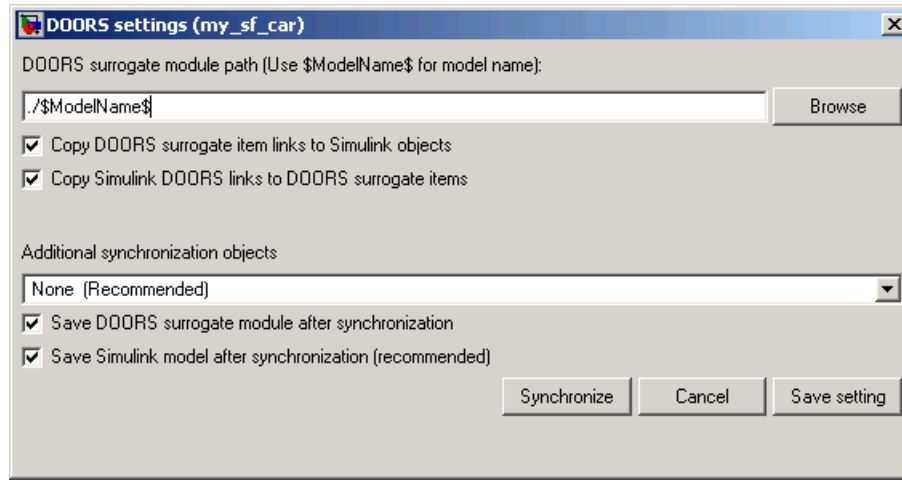
The deleted block appears at the bottom of the list of objects in the formal module and its entry in the **Block Deleted** column is `True`. If you want, you can delete this entry by right-clicking the line and selecting **Delete**. Otherwise, the module records the former presence of the deleted block.

8 Before you close the DOORS project, save the synchronized module in the DOORS software.

Customizing the DOORS Synchronization Settings

The DOORS settings dialog box lets you control not only the level of synchronization detail, but also the actions that the Requirements Management Interface performs upon synchronization.

1 From the **Tools** menu in the Simulink window select **Requirements > Synchronize with DOORS**. The DOORS settings dialog box opens.

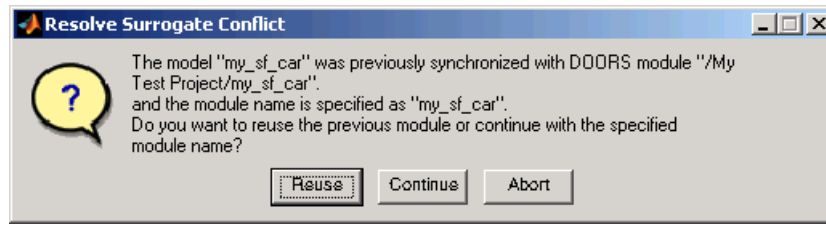


Another way to access this dialog box is to select the Synchronize Requirements with DOORS tool  in the Model Explorer window.

The **DOORS surrogate module path** field identifies the module within the DOORS database. You can specify a module with either a relative path (starting with `./`) or a full path (starting with `/`). Relative paths are appended to the current DOORS project. Absolute paths must specify a project and a module name.

After you synchronize a model, the Requirements Management Interface automatically updates the **DOORS surrogate module path** field with the actual full path. It also saves the unique module identifier with the module, to identify when the surrogate is renamed.

If you select a new module path, or if the surrogate module is renamed, the Resolve Surrogate Conflict dialog box appears when you click **Synchronize**.



It gives you the options to reuse the previous module, to continue with the specified path, or to abort synchronization.

2 Use the following options in the DOORS settings dialog box to customize your synchronization settings:

- **Copy DOORS surrogate item links to Simulink objects** — If this check box is selected, at the time of synchronization the Requirements Management Interface copies all the requirement links created from the surrogate module items into the appropriate Simulink model objects.
- **Copy Simulink DOORS links to DOORS surrogate items** — If this check box is selected, at the time of synchronization the Requirements Management Interface copies all the requirement links created directly from the Simulink model into the appropriate surrogate module items.

  Keeping both these check boxes selected ensures that your requirement link information is completely synchronized.
- **Additional synchronization objects** — Lets you select the level of synchronization detail, as described in “Customizing the Level of Synchronization Detail” on page 3-17.
- **Save DOORS surrogate module after synchronization** — If this check box is selected, the DOORS formal modules are automatically saved upon synchronization. If you clear the check box, you will have to save them manually.
- **Save Simulink model after synchronization (recommended)** — If this check box is selected, the Simulink model is automatically saved upon synchronization. It is recommended that you use this option.

3 After you select the desired configuration, click **Save Settings**.

Linking Requirements to the DOORS Synchronized Module

After you create or resynchronize a synchronized module, you can add requirements for its objects in another DOORS formal module. Each requirement is then linked to its DOORS object in the synchronized module. This establishes recognizable requirements in the Requirements Management Interface.

In "Creating a DOORS Requirement Object" on page 3-9, you created a `Transmission Requirements` object in the `Requirements` formal module in the DOORS software.

Now use the following procedure to add this requirement to the synchronized module you created for the `sf_car` model in "Synchronizing a DOORS Module with the Simulink Model" on page 3-14:

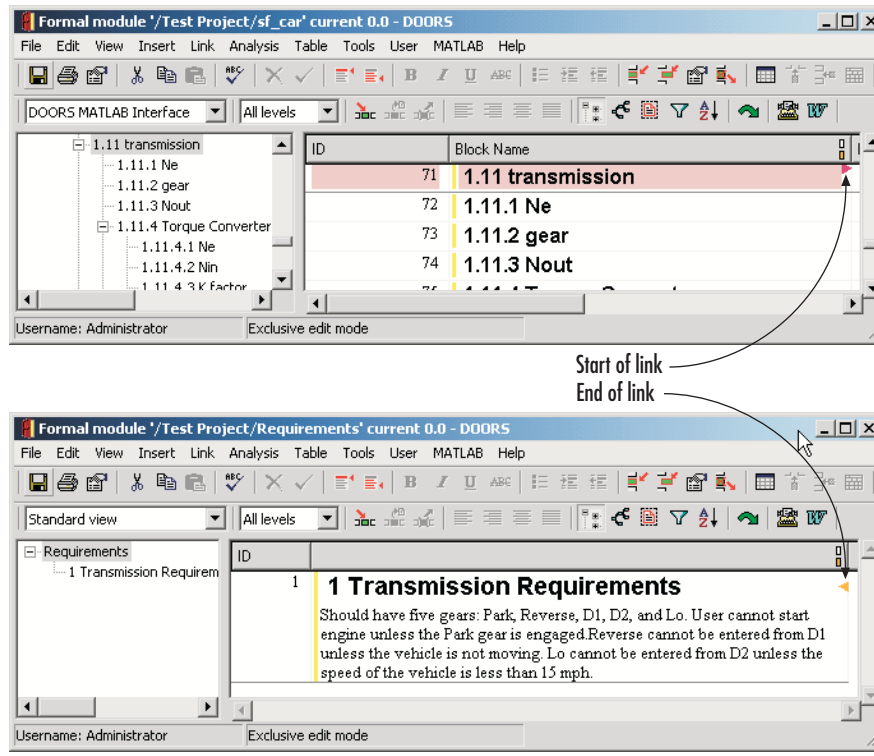
1 Open the `Requirements` formal module in the DOORS software.

2 In the main DOORS window, open the synchronized module `sf_car` and scroll down to the `transmission` object.

3 Right-click the `transmission` object and select **Link > Start Link** from the resulting context menus.

4 In the `Requirements` formal module window, right-click the `Transmission Requirements` object and select **Link > Make Link from Start** from the resulting context menus.

   A link now exists between the `transmission` object in the synchronized module and the `Transmission Requirements` object in the `Requirements` module. The presence of the link is indicated by a right-facing arrow for the `transmission` object in the synchronized module and a left-facing arrow in the `Transmission Requirements` object in the `Requirements` module.



The requirement you install in this section is an example of an official DOORS requirement for the Requirements Management Interface. You can navigate between the object in the synchronized module and its DOORS requirement by right-clicking one of the arrows and selecting from the resulting pop-up menu. You can also establish more links from the object to other requirements. Later on, when you display Simulink objects with DOORS requirements in “Navigating Between Model Objects and DOORS Requirements” on page 3-26, these are the requirements that the Requirements Management Interface detects.

Navigating Between Model Objects and DOORS Requirements

In this section...

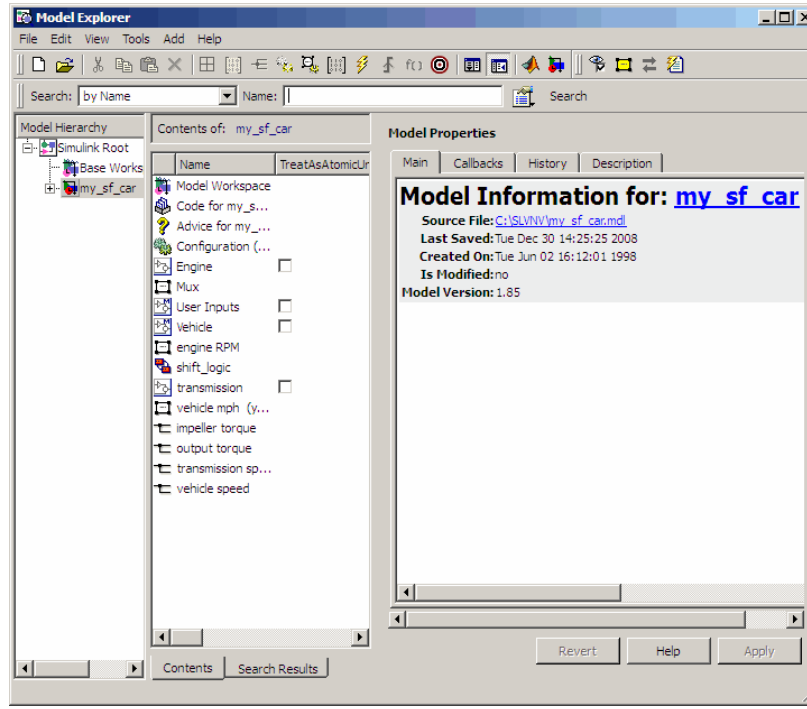
Viewing Model Elements with Requirements

It is sometimes helpful to distinguish model objects with requirements from those without requirements in a single glance. The Requirements Management Interface lets you see model elements with requirements linked to the synchronized module both in the Simulink window and in the Model Explorer.

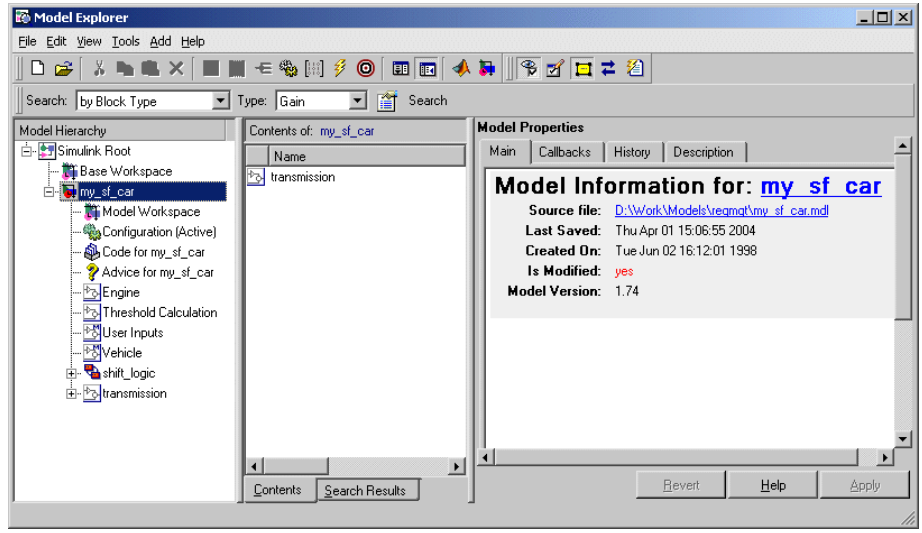
Use the following procedure to display only those model elements with requirements:

1 In the Simulink model, from the **View** menu, select **Model Explorer**.

The Model Explorer window appears with the model highlighted in the **Model Hierarchy** pane.



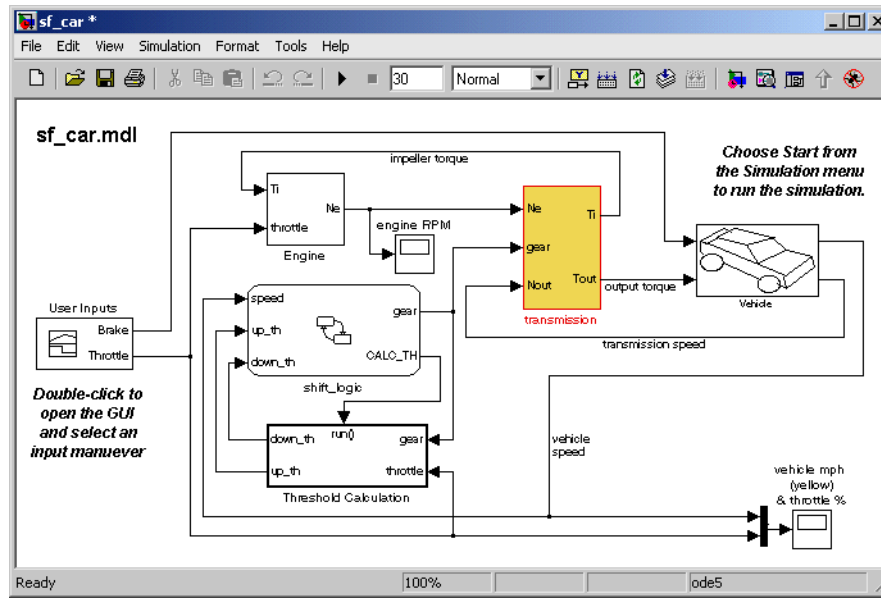
2 Select the Display Objects with Linked Requirements tool  in the Model Explorer toolbar.



The Model Explorer displays only the `transmission` object, which you added requirements to in "Linking Objects to DOORS Requirements" on page 3-9.

3 Select the Highlight Items with Requirements on Model tool  in the Model Explorer toolbar.

The transmission block in the Simulink model is highlighted.

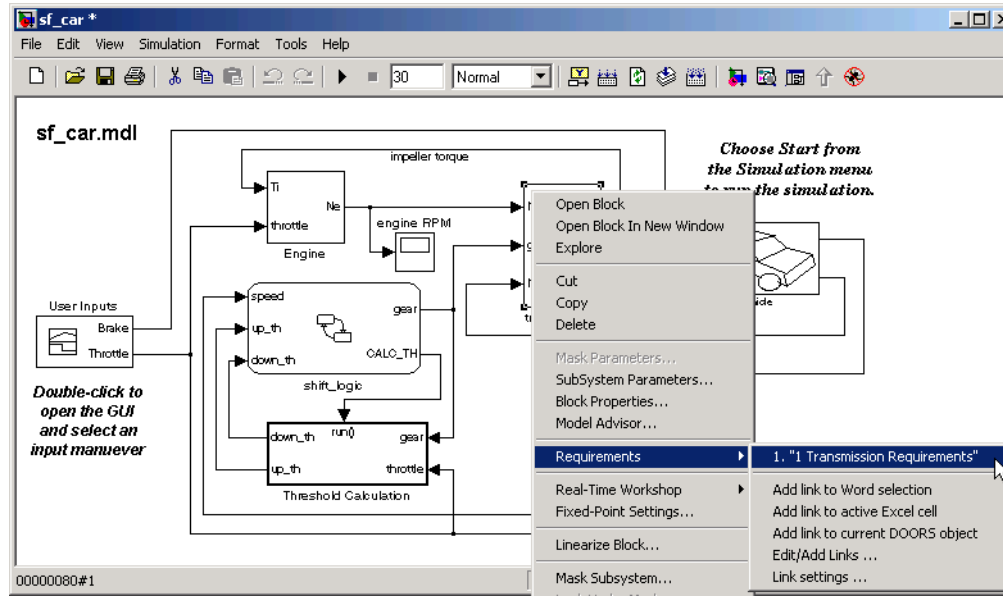


Navigating from a Simulink Model to DOORS Requirements

If you create requirement links directly from the Simulink or Stateflow object, you can navigate directly from the object to the DOORS requirement. In “Linking Objects to DOORS Requirements” on page 3-9, you create a link from the transmission block in the Simulink model to the Transmission Requirements DOORS object.

Use the following procedure to navigate from the transmission block in the Simulink model to its associated DOORS requirement:

1. In the Simulink software, open the model `sf_car_doors`.
2. Right-click the transmission block and, from the resulting pop-up menu, select **Requirements > 1. “Transmission Requirements”**.



The `Requirements` formal module window opens scrolled to the Transmission Requirements link.

Navigating Through the Synchronized Module

If you use the synchronized module to create links to DOORS requirements, as described in "Linking Requirements to the DOORS Synchronized Module" on page 3-24, then you can navigate between Simulink objects and DOORS requirements by using the synchronized module as an intermediary. You first navigate to the unique object in the synchronized module from its object in the Simulink model or the Model Explorer. From the synchronized module, you then access requirements for each object through the linking process in the DOORS software.

Use the following procedure to navigate from a Simulink object to the object mapped in the synchronized module:

1 In the Simulink model, right-click a block with requirements.

A pop-up menu appears.

2 In the pop-up menu, select **Requirements > DOORS Surrogate Item**.

   If the synchronized module is closed, it opens and the mapped object is highlighted. If the synchronized module is already open, only the mapped object is highlighted.

3 Access individual requirements in the synchronized module.

   You can access individual requirements by right-clicking the arrows that appear in the **Block Name** column for each mapped object with requirements and making a requirement selection.

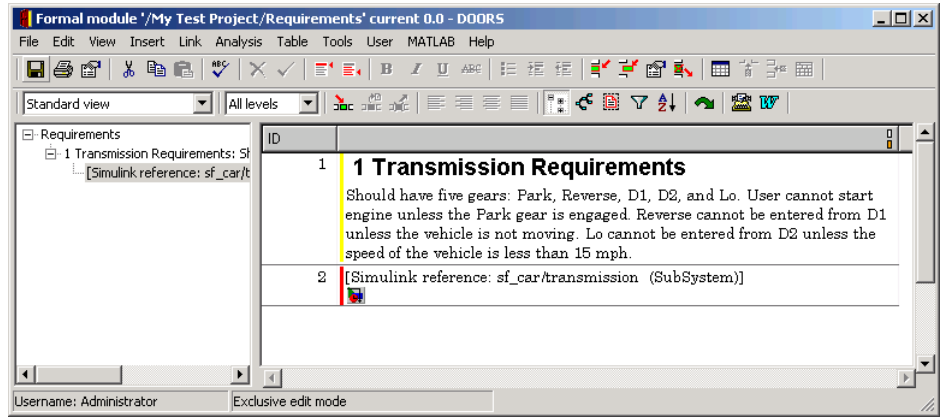
Navigating from a DOORS Requirements to the Simulink Model

If you create two-way requirement links directly from the Simulink or Stateflow object, you can navigate from the DOORS requirement directly to the associated object in the Simulink or Stateflow diagram. In "Linking Objects to DOORS Requirements" on page 3-9, you create a link from the transmission block in the Simulink model to the Transmission Requirements DOORS object.

Use the following procedure to navigate from the Transmission Requirements DOORS object to the transmission block in the Simulink model:

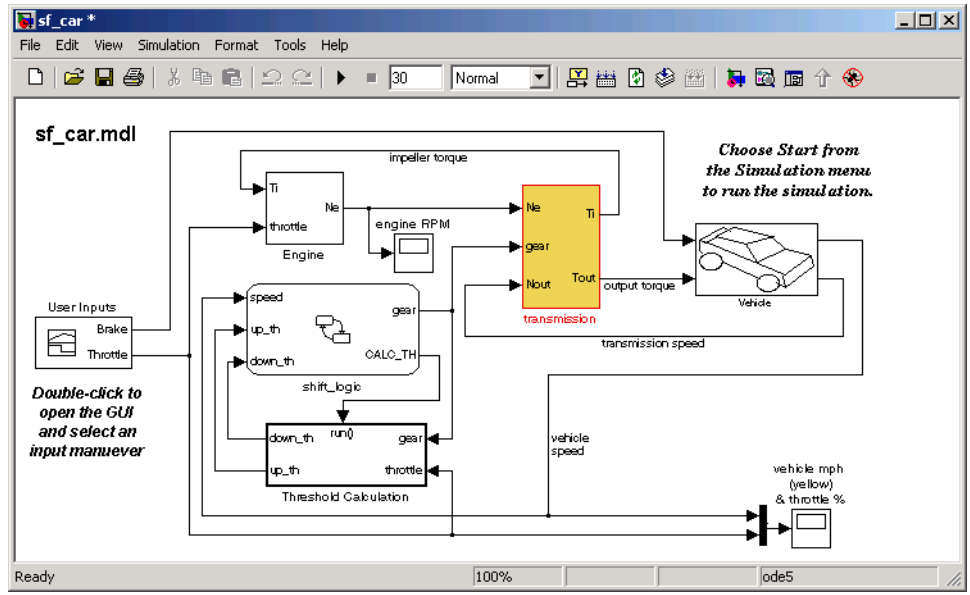
1 In the DOORS software, open the formal module window `Requirements`.

2 Select the Simulink Reference sub-node of the Transmission Requirements node in the left pane.



3 From the **MATLAB** menu in the formal module window `Requirements`, select **Select item**.

The transmission block in the Simulink diagram is highlighted.



Navigating Through the Synchronized Module

In the DOORS software, you can navigate from a requirement in a formal module to its mapped object in the synchronized module through the left-facing arrows in the **Block Name** column for each requirement. This brings focus to the synchronized module with the owning object selected.

You can navigate from an object in the synchronized module to its Simulink object as follows:

1. In the DOORS synchronized module, click an object in either the left or right pane to select it.
2. From the **MATLAB** menu, choose **Select item**.

   The object opens in its native diagram as follows:

   - For a Simulink object, the model window of the subsystem containing the selected object opens with that block or subsystem selected. All parent Simulink blocks are selected as well, so that you can reach the object from any higher-level object.
   - For a Stateflow object, the diagram containing the selected object opens with the object highlighted.

Note Although the **MATLAB** menu and **Select item** feature appear in all DOORS formal modules, you can use them only in a synchronized formal module.

If the DOORS **Block Deleted** status for the object is `True`, you cannot navigate to the object.

4

Managing Model Verification Blocks

You use Model Verification blocks throughout your model to monitor individual signals relative to limits that you impose on them. Use Model Verification blocks in conjunction with the Verification Manager tool in the Signal Builder block to carefully construct simulation tests for your model from a single location.

Using Model Verification Blocks

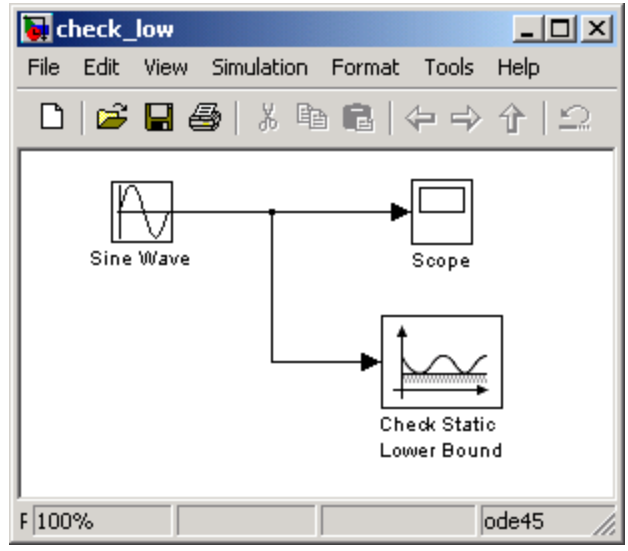
You use Model Verification blocks throughout your model to monitor its signals. You can set a verification block to assert when its signal leaves the specified limit or range. During simulation, when the signal crosses the limit, the verification block can

- Stop simulation and bring immediate focus to that part of the model
- Report the limit encounter with a logical signal output of its own, which can be true if the limit is not encountered and false if the limit is encountered

To see a complete list of all Model Verification blocks and references for each, see the "Model Verification" category in the Simulink Block Reference documentation.

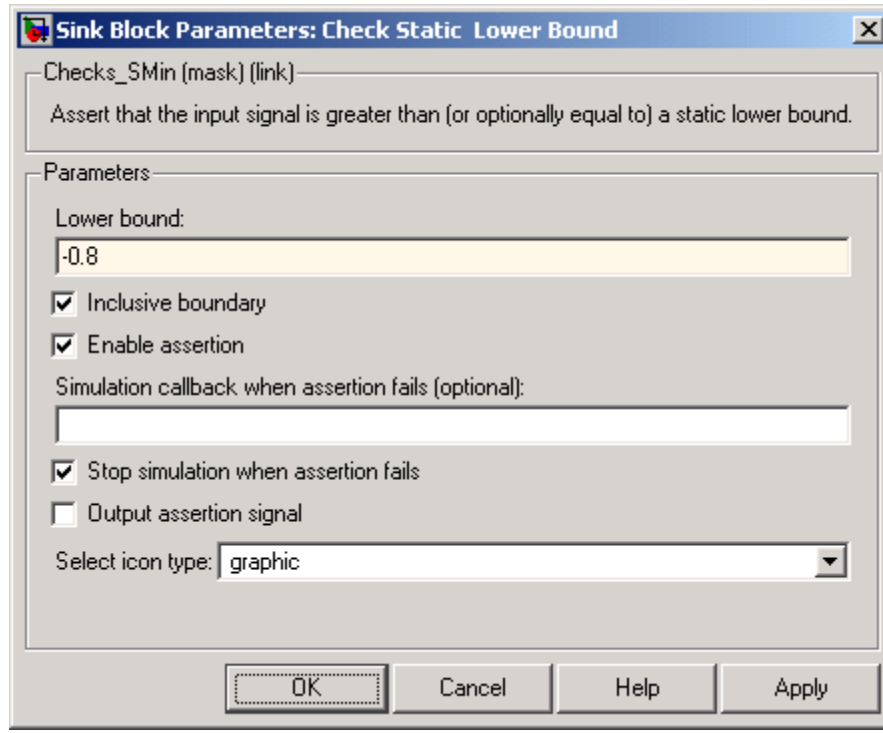
In the following example, a Check Static Lower Bound verification block is used to stop simulation when a signal from a Sine Wave block crosses its lower bound limit:

1 Attach a Check Static Lower Bound verification block to the signal from a Sine Wave block, as shown in the following schematic.



2 Set the model to run for 2 seconds while the Sine Wave block outputs a signal with an amplitude of 1 and a frequency of `pi` radians per second.

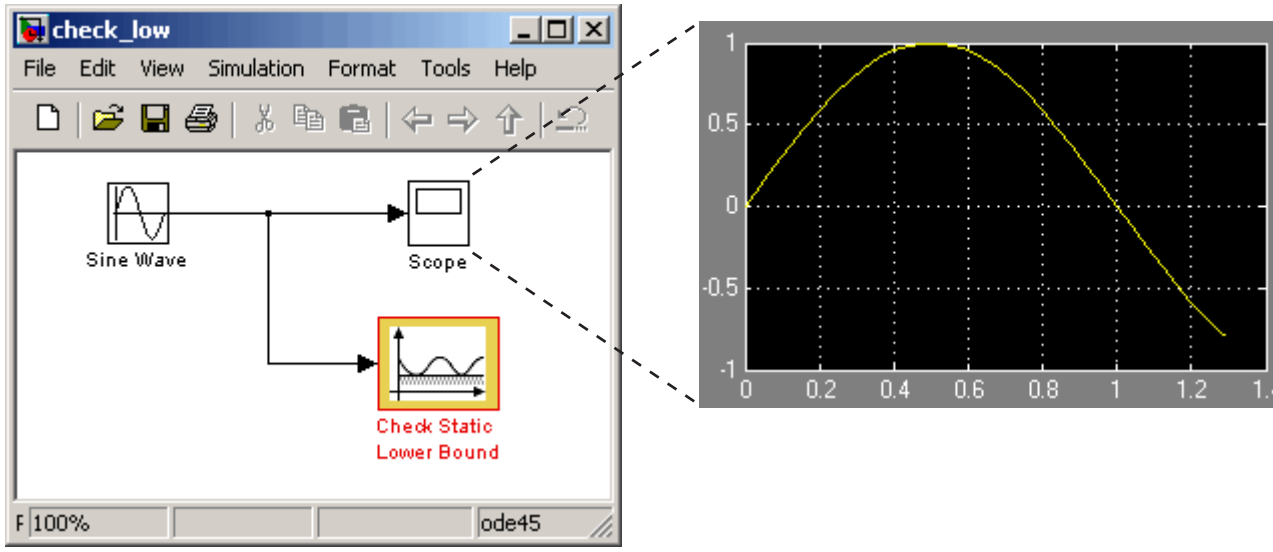
3 Open the Check Static Lower Bound block and set the parameters as follows:



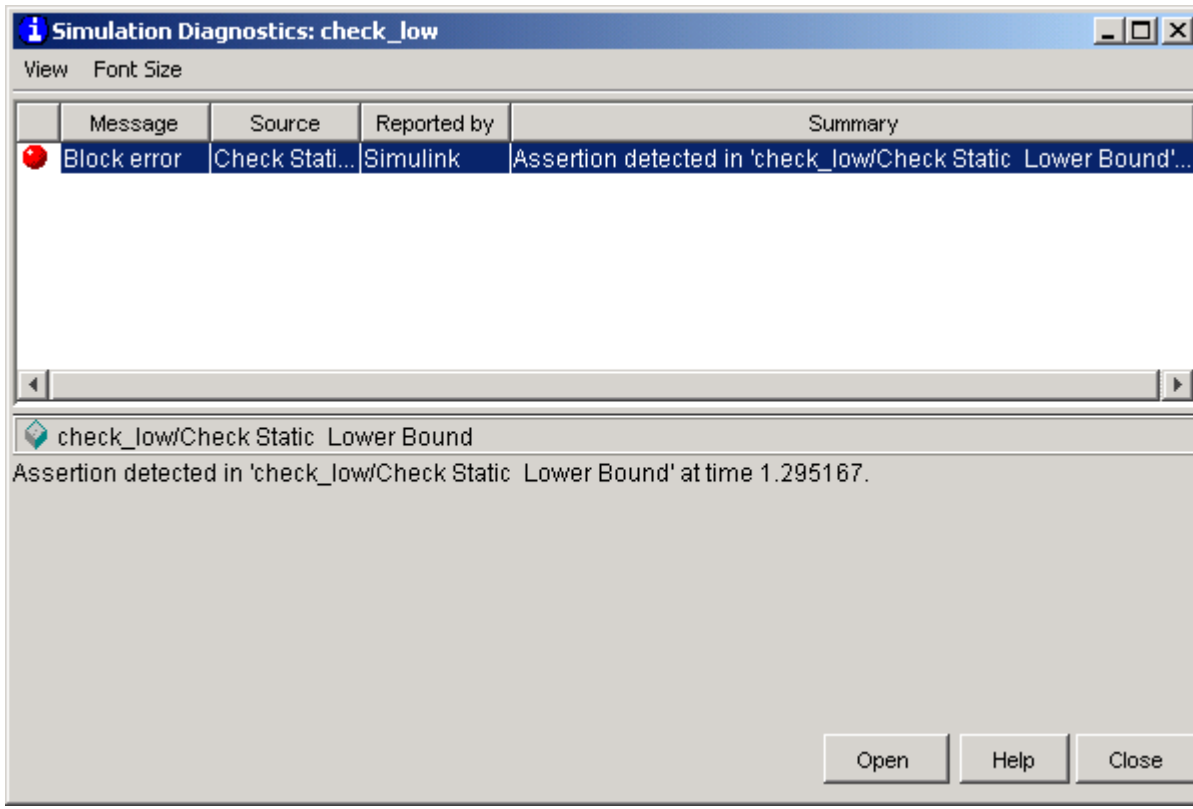
A verification block is enabled for an assertion when the **Enable assertion** check box is selected (this is the default setting). According to the preceding property settings, the Check Static Lower Bound block is set to detect a signal value of -0.8 or lower. If this signal is detected, simulation is stopped.

4 Run the simulation.

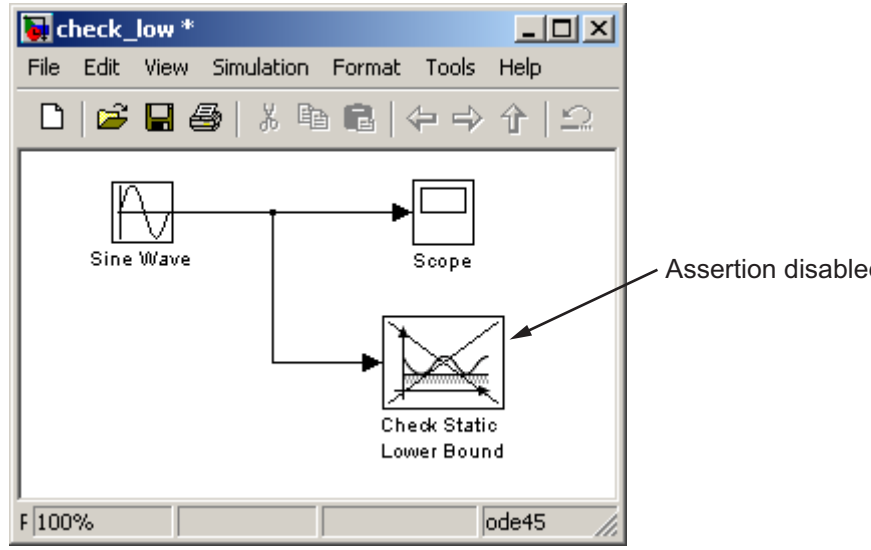
The model stops simulating after 1.295 seconds, when the output is -0.8, as shown. This brings focus to the asserting verification block, which is highlighted.



The stop in simulation is also accompanied by the following status diagnostic message.



5 You can disable the block from asserting its limit by clearing the **Enable assertion** check box, which has the following effect on the block's appearance in the model.


check_low *
File Edit View Simulation Format Tools Help
Sine Wave
Scope
Check Static
Lower Bound
100%
ode45
Assertion disabled

Using the Verification Manager

In this section...

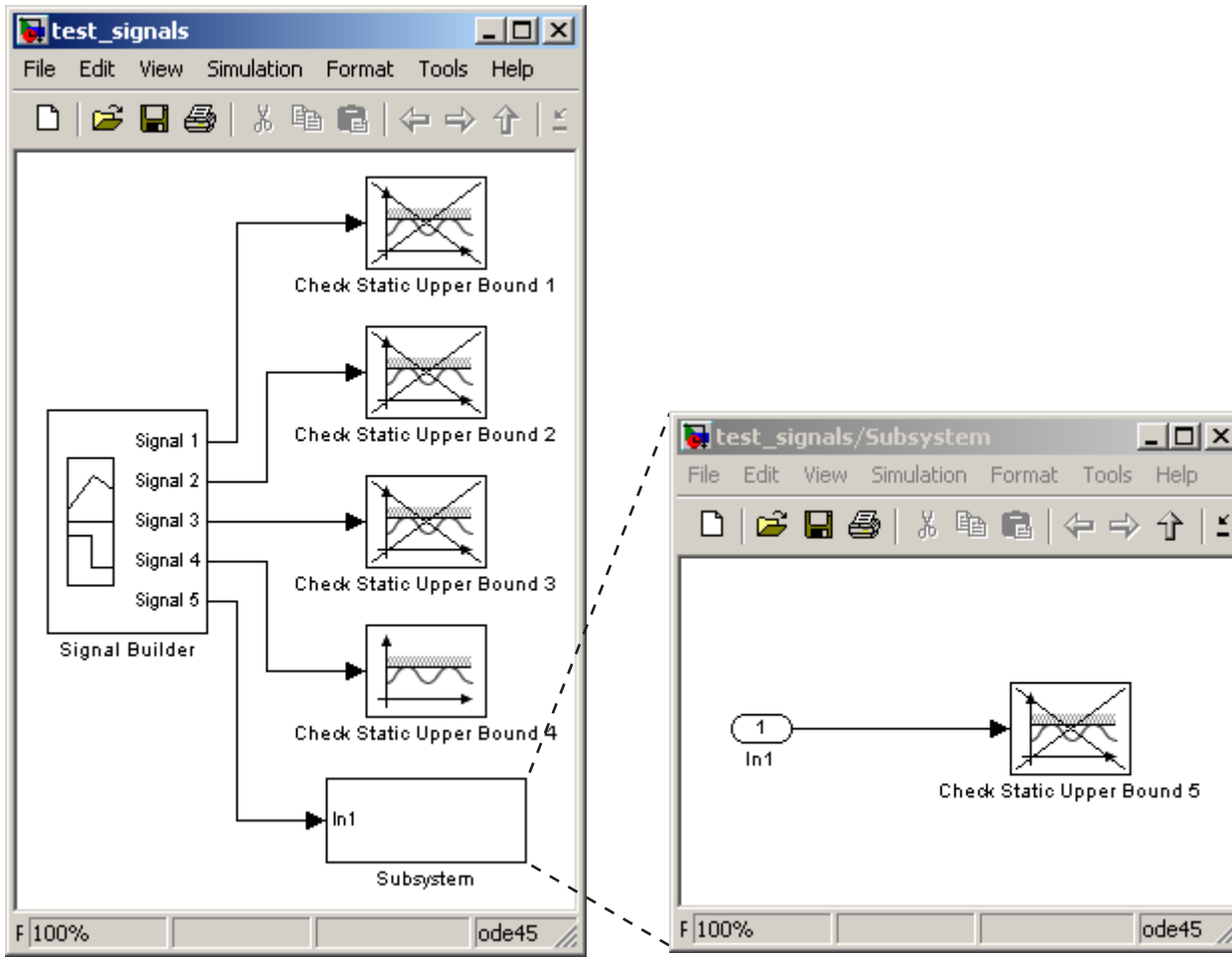
What Is the Verification Manager?

The Verification Manager is a graphical interface that appears in the Signal Builder dialog box. The tool allows you to manage from a central location all the Model Verification blocks in your model. The sections that follow describe how to access the Verification Manager for the purpose of enabling or disabling Model Verification blocks in a Simulink model.

Opening the Verification Manager

In this topic you create a Simulink model that you use to examine the Verification Manager in the following steps:

1 Create the following example model in the Simulink software.

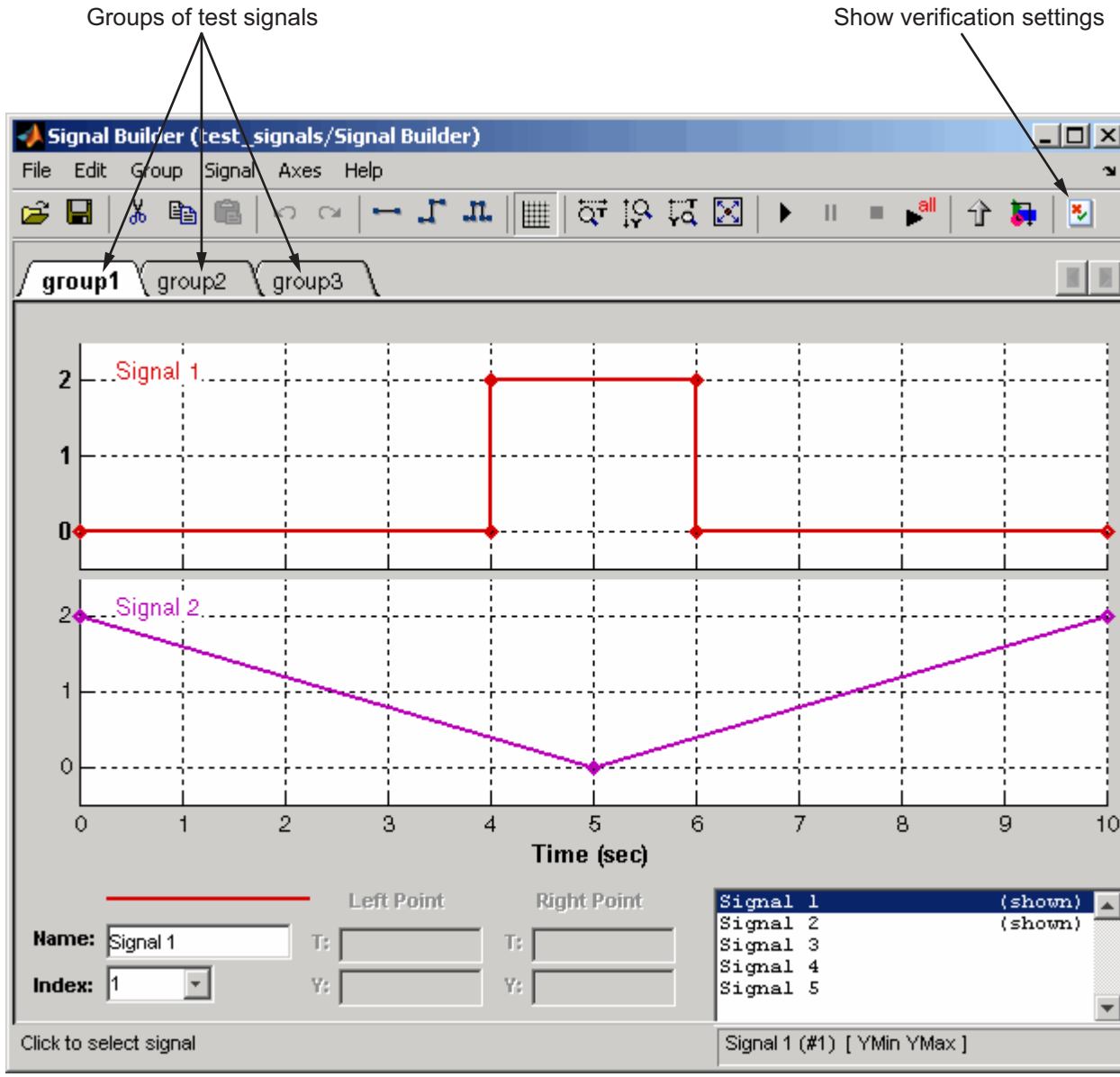


Typically, a Signal Builder block provides test signals for an entire model from one location. The example model contains a Signal Builder block feeding five test signals to Model Verification blocks. Signals 1 through 4 are sent directly to Check Static Upper Bound Model Verification blocks. The fifth signal is sent to a subsystem that contains a Check Static Upper Bound verification block.

Each Check Static Upper Bound verification block is set to assert for an upper bound of 1 (property **Upper bound** = 1). Blocks 1, 2, 3, and 5 appear

crossed out because they are disabled (property **Enable assert** is cleared). Block 4 is enabled (property **Enable assert** is checked).

2 Double-click the Signal Builder block in the preceding model to open its Signal Builder dialog box.

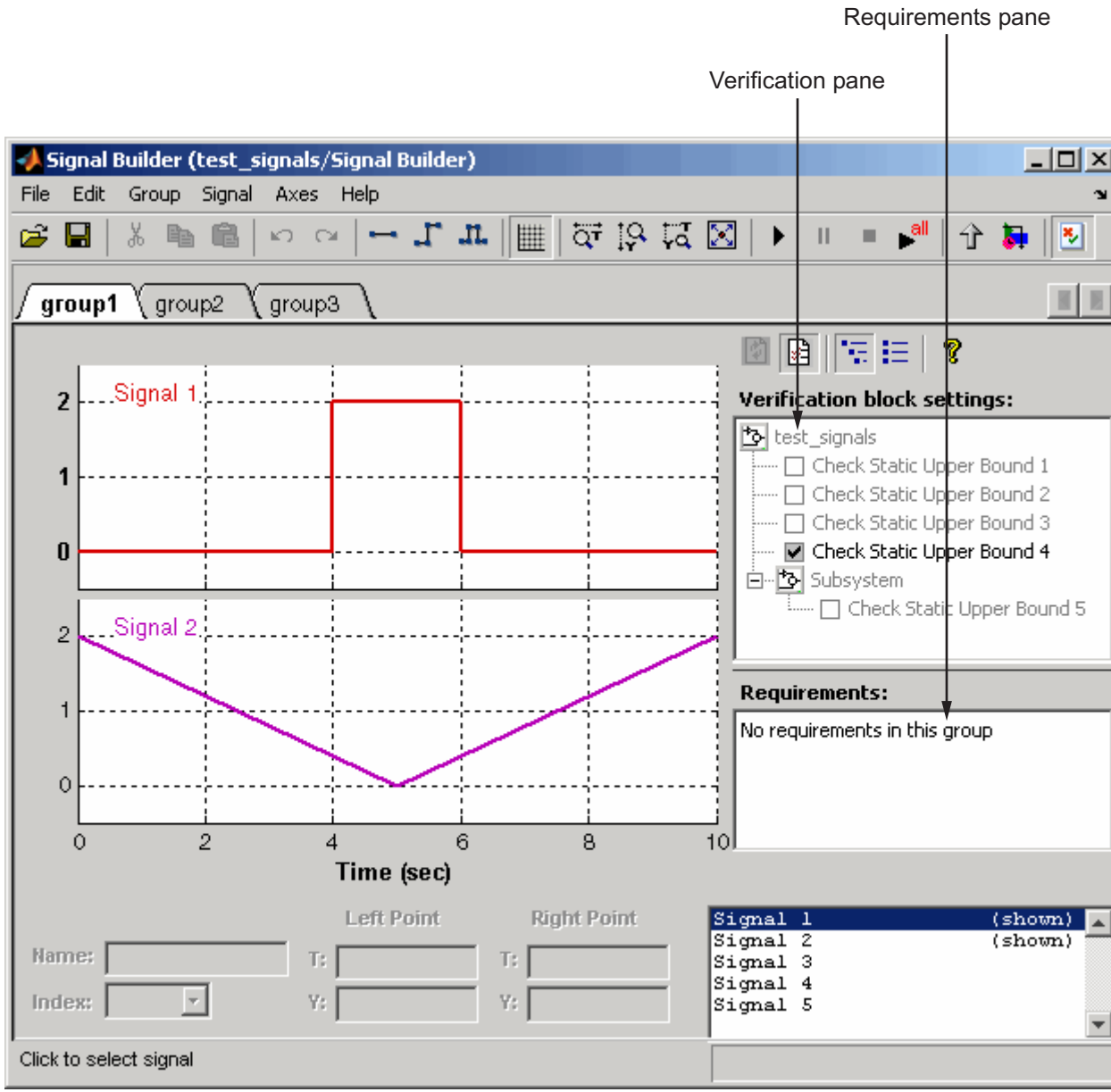


The Signal Builder dialog box displays tabbed pages for three groups of signal values. Each group contains independent values for all five signals.

However, only a subset of the signals is displayed for each group. For example, **group1** displays signals 1 and 2. For more information on the Signal Builder block, see "Working with Signal Groups" in the Simulink documentation.

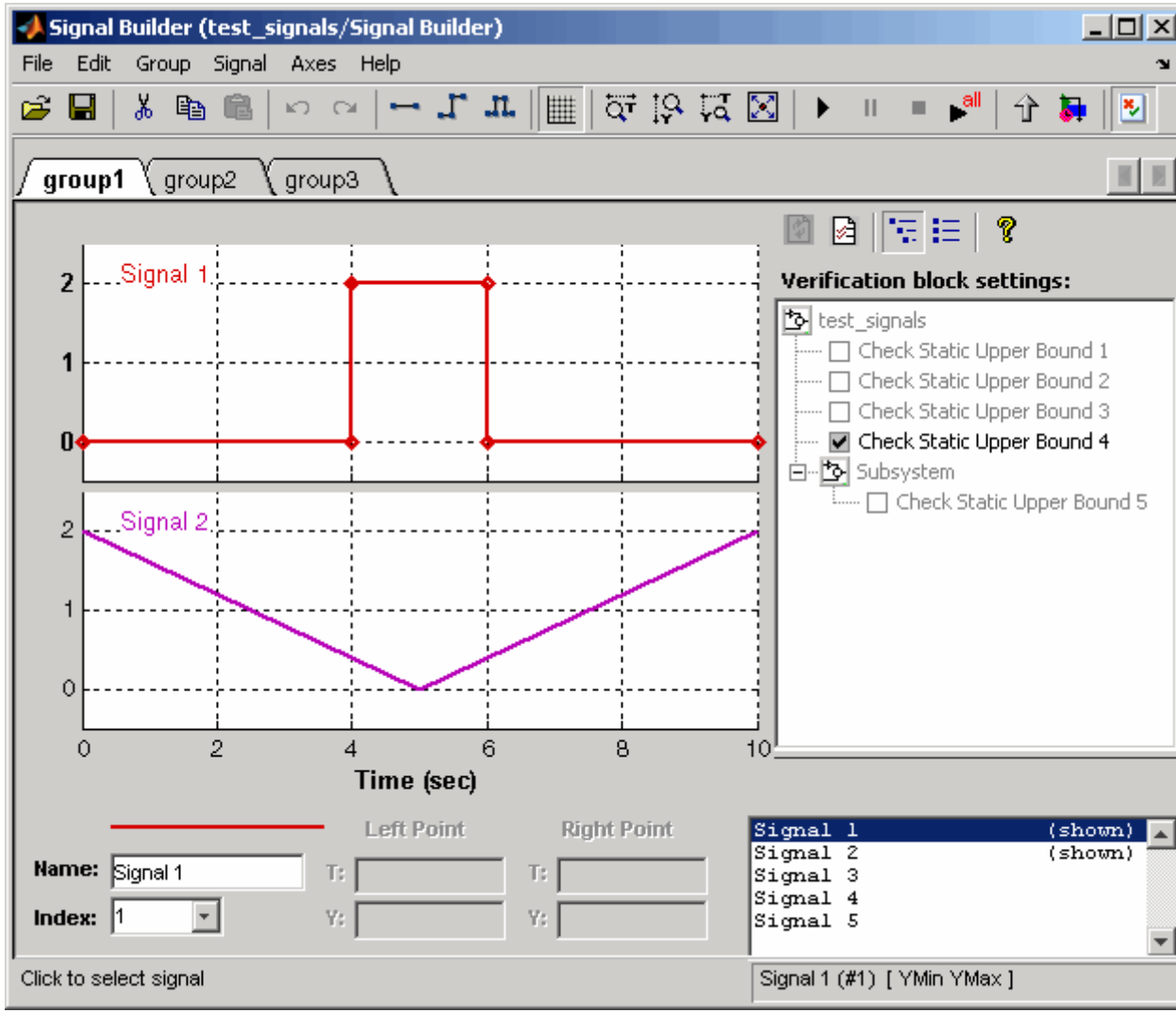
3 In the Signal Builder dialog toolbar, select the Show Verification Settings tool  .

The **Verification block settings** pane and the **Requirements** pane appear as shown.


Requirements pane
Verification pane
Signal Builder (test_signals/Signal Builder)
File Edit Group Signal Axes Help
group1 group2 group3
Signal 1
Signal 2
Time (sec)
Verification block settings:
test_signals
Check Static Upper Bound 1
Check Static Upper Bound 2
Check Static Upper Bound 3
Check Static Upper Bound 4
Subsystem
Check Static Upper Bound 5
Requirements:
No requirements in this group
Left Point
Right Point
Name:
Index:
Signal 1 (shown)
Signal 2 (shown)
Signal 3
Signal 4
Signal 5
Click to select signal

By default, the **Verification block settings** pane lists all Model Verification blocks for the model, grouped by subsystem. The **Requirements** pane lists the requirements document links for the current signal group. See "Managing Verification Requirements" on page 4-24 for details on adding requirement document links in the Signal Builder dialog box. For now, delete the **Requirements** pane in the next step.

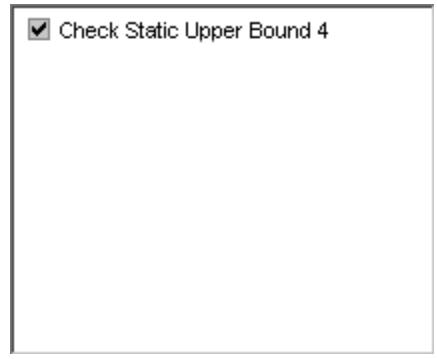
4 Just above the **Verification block settings** pane, select  to close the **Requirements** pane.



The example **Verification block settings** pane displays five Model Verification blocks. Four are in the top level of the model, and one is in a subsystem.

5 Select the List Enabled Verifications tool  in the **Verification block settings** toolbar.

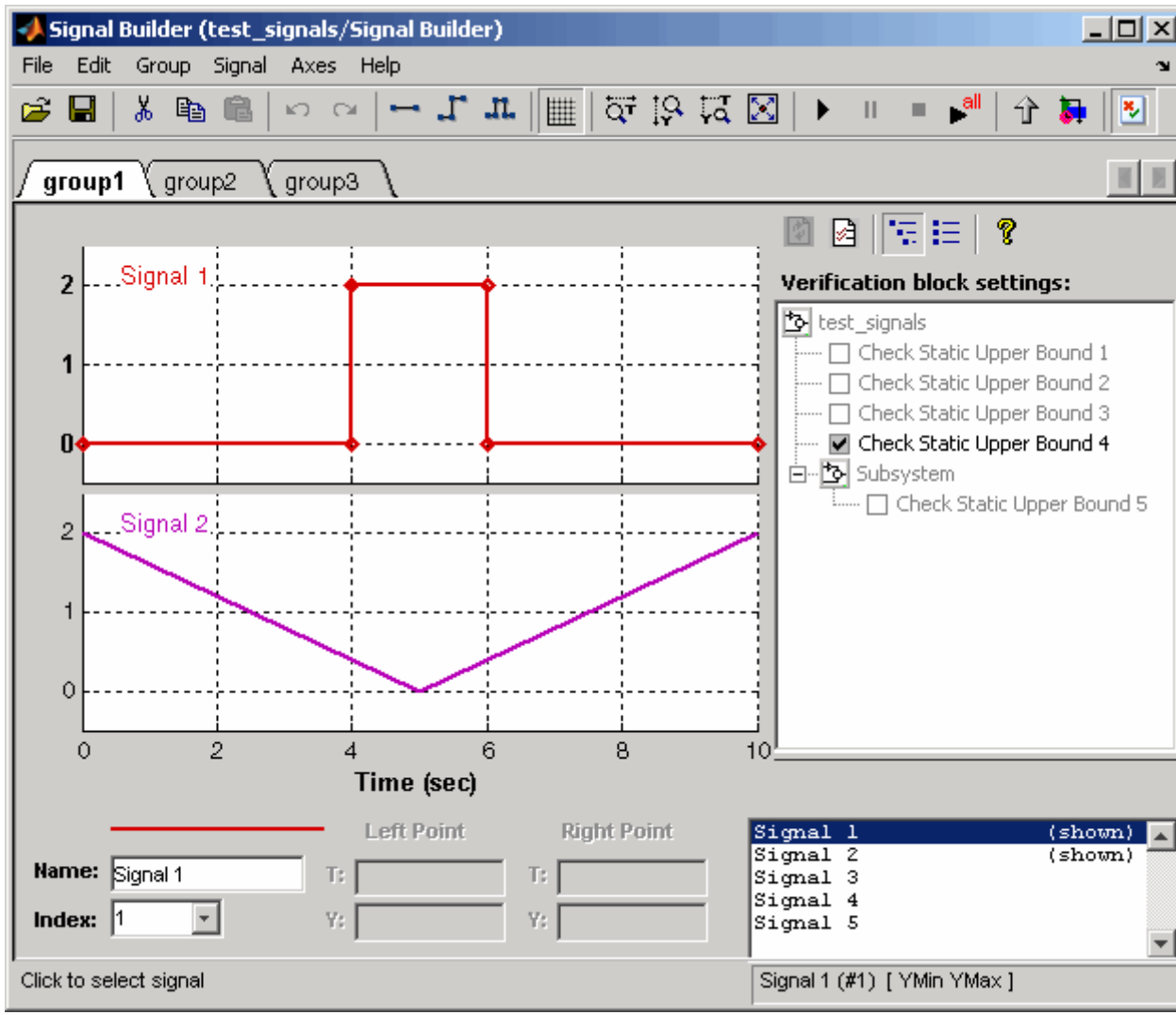
The **Verification block settings** pane now shows only the enabled Model Verification blocks for the current group, as shown.



6 Select the Show Verification Block Hierarchy tool  to list all Model Verification blocks for the current group again.

Enabling and Disabling Model Verification Blocks with the Verification Manager

In this section you use the Verification Manager to selectively enable and disable Model Verification blocks in group tests. In “Opening the Verification Manager” on page 4-7, you open the Verification Manager in the Signal Builder, as shown.



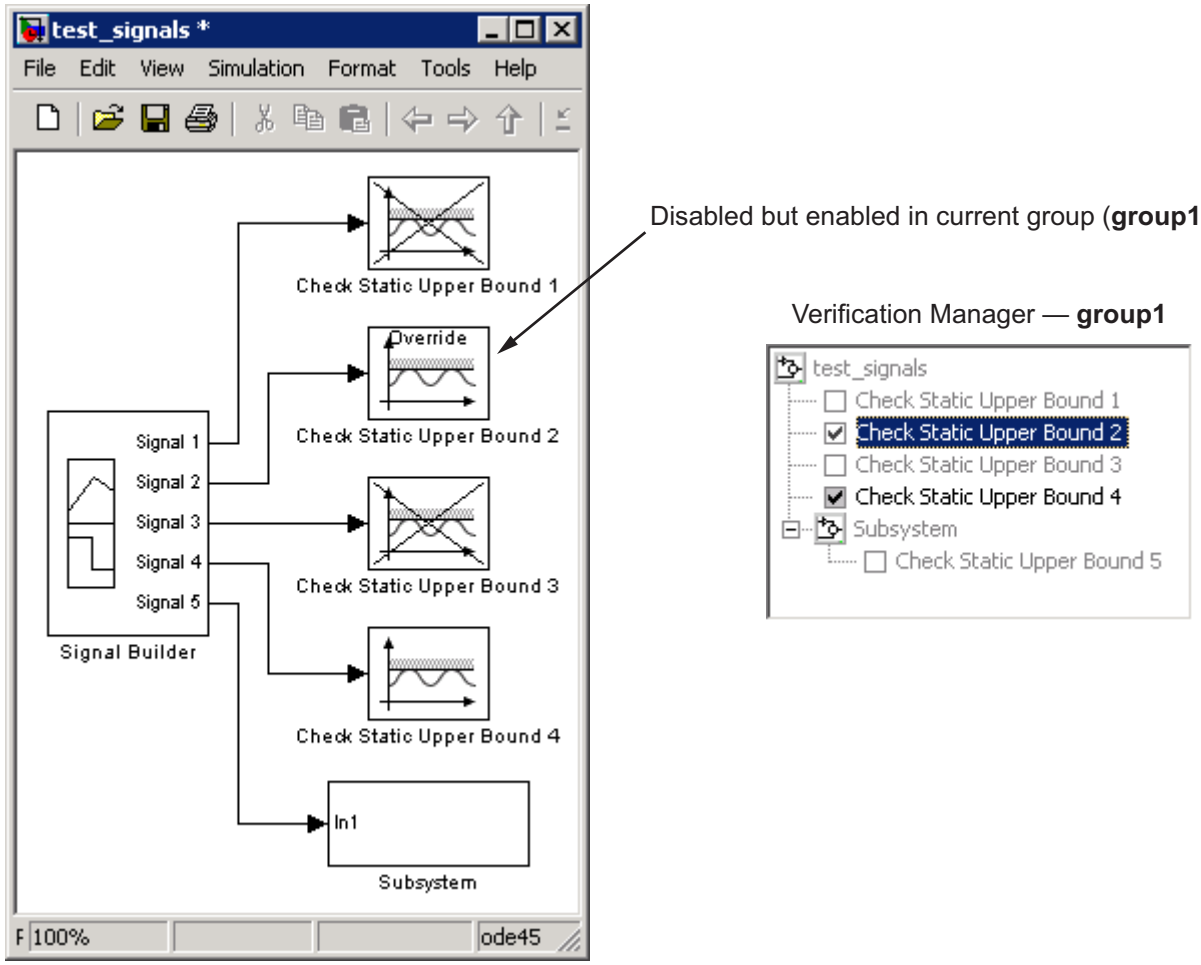
The **Verification block settings** pane in the preceding example lists the Model Verification blocks in the model. Each verification block has a preceding status node that indicates whether its assertion is enabled or disabled and whether that setting applies universally or to the active group. The preceding status node can be one of the following.

Node	Status
☐	Verification block is disabled for this group. Click to enable for current group.
☑	Verification block is enabled for the current group. Click to disable for current group.
☑	Verification block is enabled for all test groups.

Use the Verification Manager to enable or disable model verification blocks in the `test_signals` model you created in "Opening the Verification Manager" on page 4-7, as follows:

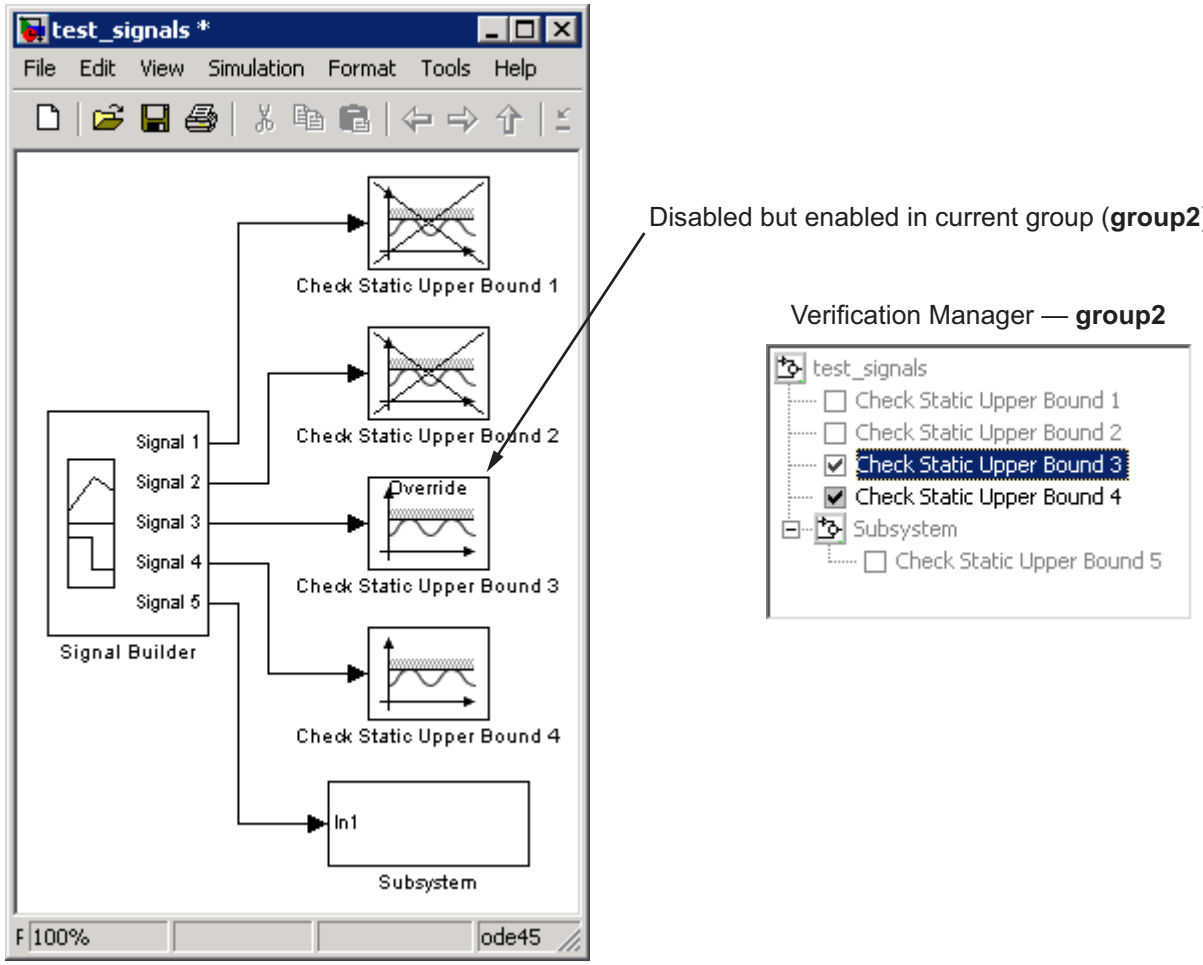
1. In the Verification Manager, click the empty check box next to the Check Static Upper Bound 2 node to enable it for the current group (**group1**).

   Enabling a disabled block in the **Verification block settings** pane leads to the following change in block appearance in the model.



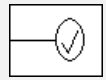
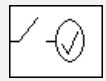
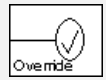
Because it is enabled in the current group, the Check Static Upper Bound 2 block gains an **Override** label and loses its cross-out. The meaning behind the change in appearance becomes clearer when another group is selected.

2 In the Signal Builder dialog box, select the **group2** tab and click the empty check box next to the Check Static Upper Bound 3 block to enable it for the current group (**group2**).



The Check Static Upper Bound 3 block loses its cross to indicate that it is enabled for the current group. However, Check Static Upper Bound 2 gains a cross because it is enabled in another group, but not this one.

The change in appearance of the Check Static Upper Bound blocks in the preceding steps is exemplary of the change in appearance of every other Model Verification block except the Assertion block. The change in appearance of the Assertion block is summarized in the following table:

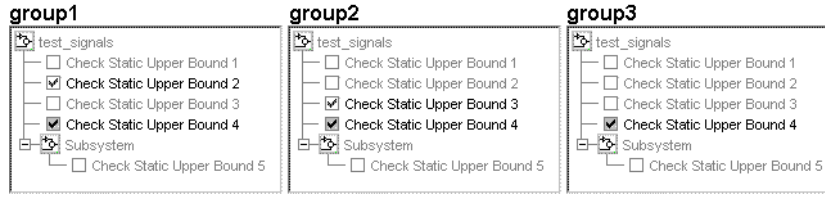
Assertion Block	Description
	Enabled for all groups
	Disabled in current group
	Enabled in current group

Using Enabling and Disabling Tools in the Verification Manager

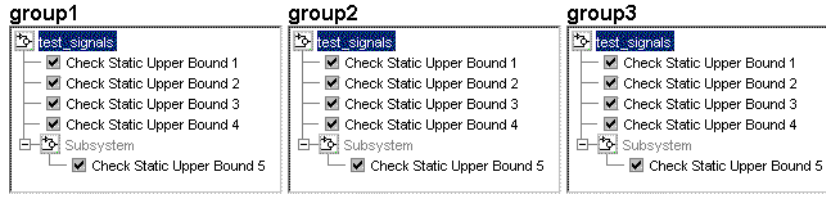
If you have many verification blocks, it is tedious to enable and disable blocks individually. For this reason, the Verification Manager lets you enable and disable blocks through selections from a context menu. These selections vary with the node as follows:

Node	Context Menu Selections
	• Contents enable for all groups • Contents enable by group • Contents group enable • Contents group disable
	• Block enable by group
	• Block enable for all groups • Block group enable
	• Block enable for all groups • Block group disable

As an example, assume that the following groups are defined in the Verification Manager for a model with five Model Verification blocks.

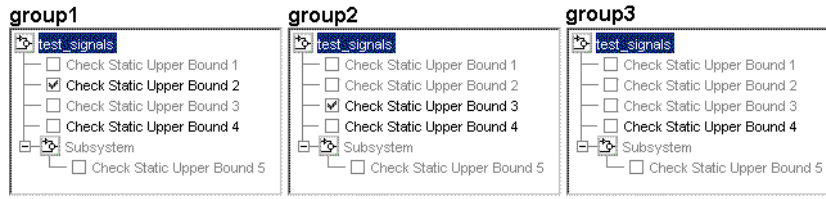


1 Right-click the `test_signals` node and select **Contents enable for all groups**.



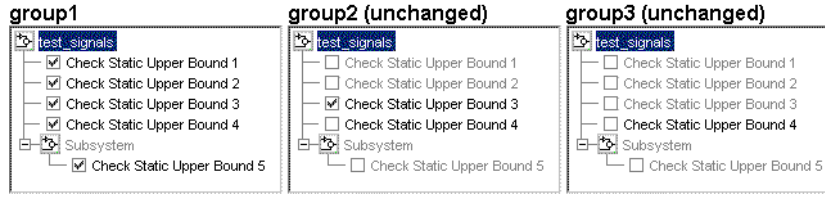
Applying the **Contents enable for all groups** selection to the model node enables all contained Model Verification blocks, for all test groups, in all contained subsystems.

2 Right-click `test_signals` and select **Contents enable by group**.



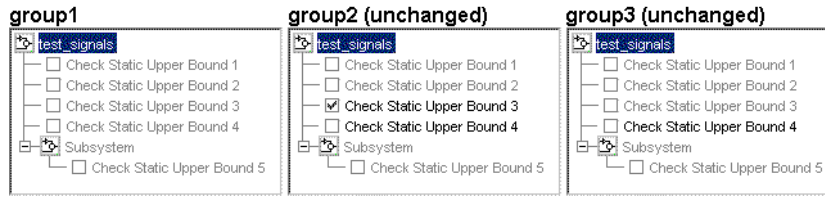
Applying the **Contents enable by group** selection to the model node restores the previous individually enabled/disabled settings for each block in each group.

3 Right-click `test_signals` and select **Contents group enable**.



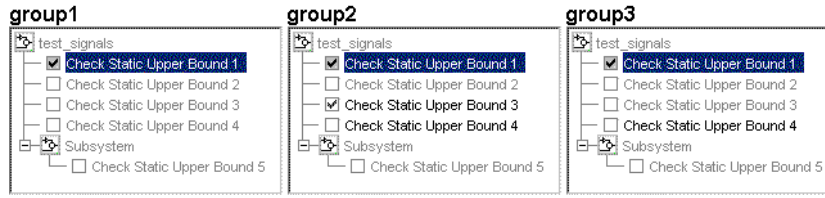
Applying **Contents group enable** to the `test_signals` model node in **group1** individually enables all contained blocks for **group1**, but leaves the other groups untouched.

4 Right-click `test_signals` and select **Contents group disable**.



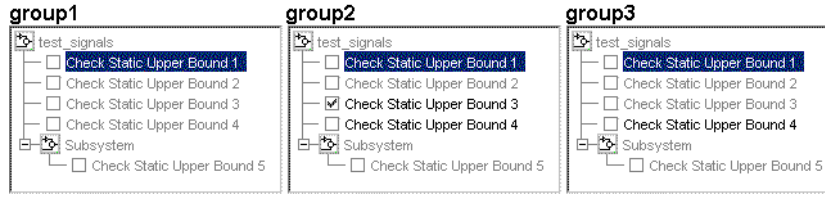
Applying **Contents group disable** to the `test_signals` model node in **group1** individually disables all contained blocks for **group1**, but leaves the other groups untouched.

5 Right-click `Check Static Upper Bound 1` and select **Block enable for all groups**.



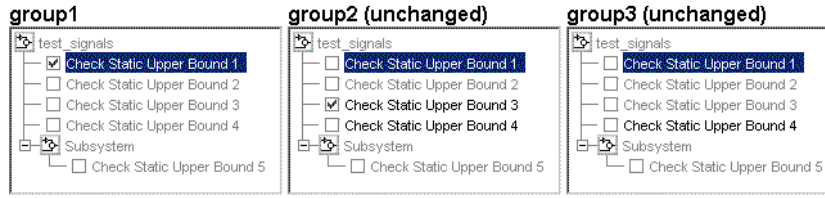
Applying **Block enable for all groups** to the individual **group1** block node for `Check Static Upper Bound 1` in **group1** enables this block for all groups.

6 Right-click `Check Static Upper Bound 1` and select **Block enable by group**.



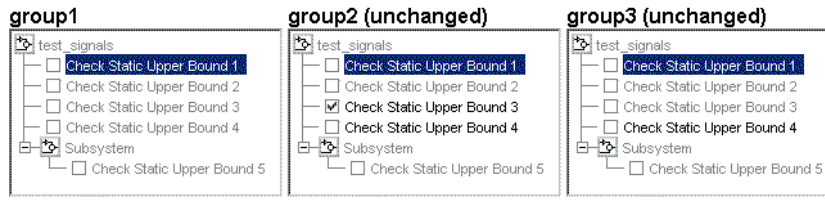
Applying **Block enable by group** to the individual **group1** block node for `Check Static Upper Bound 1` in **group1** restores the previous individually enabled/disabled state to this block for all groups. This lets you enable or disable this node individually for each group.

7 Right-click `Check Static Upper Bound 1` and select **Block group enable**.



Applying **Block group enable** to the individual **group1** block node for `Check Static Upper Bound 1` in **group1** enables this block for this group only. This is equivalent to selecting the empty check box in **group1** for this node.

8 Right-click `Check Static Upper Bound 1` and select **Block group disable**.



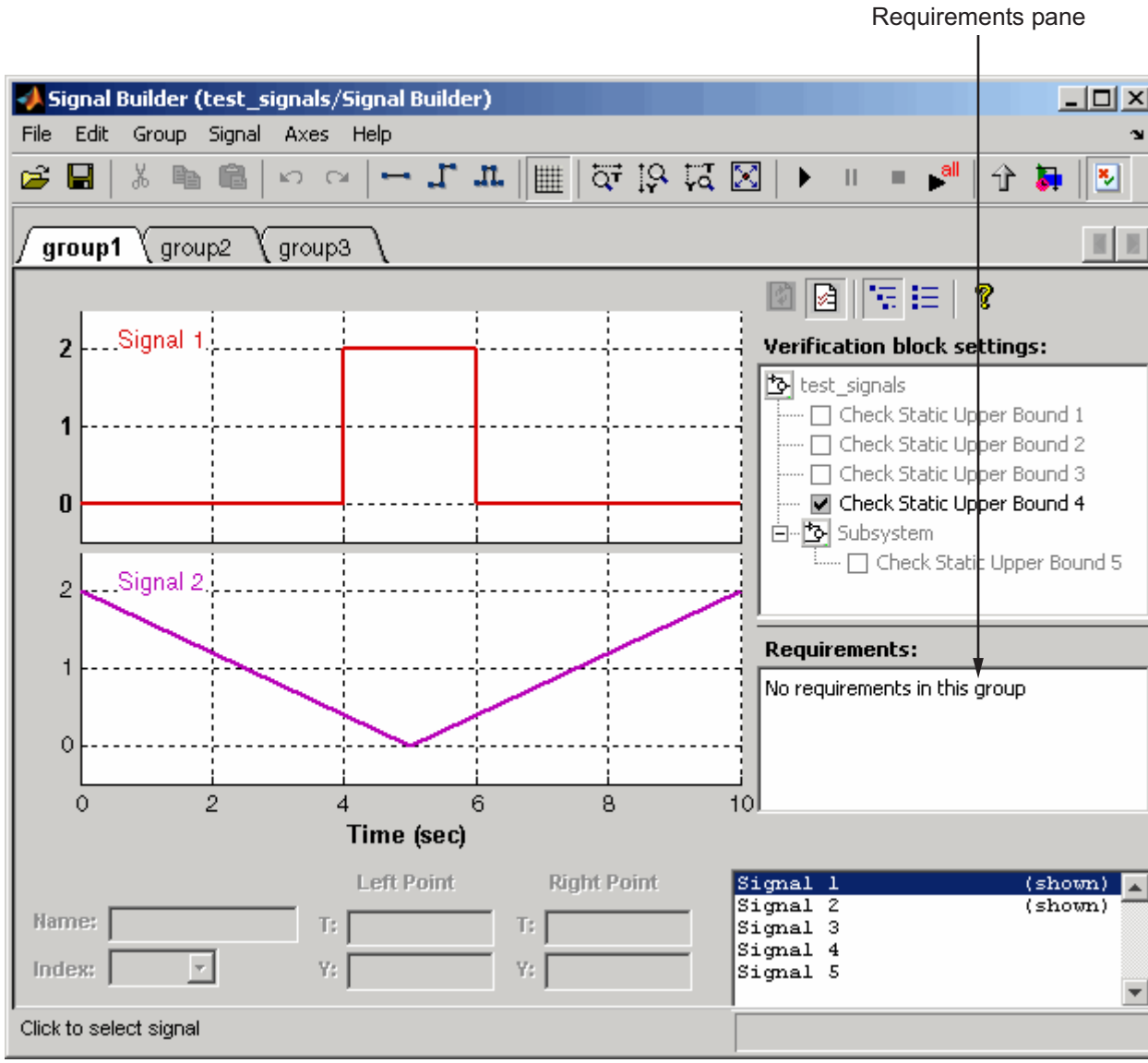
Applying **Block group disable** to the individual block node for `Check Static Upper Bound 1` in **group1** disables this block for this group only. This is equivalent to clearing the check box for this node.

Managing Verification Requirements

In "Using the Verification Manager" on page 4-7, you learn how to use the Verification Manager to manage Model Verification blocks along with signal group tests in a Simulink model. The combination of test groups and their schedules of enabled and disabled Model Verification blocks is used to verify the correct behavior for your Simulink model. In this section you learn how to link the requirements to this combination that specify correct behavior.

You can link requirements documents to individual verification blocks just as you can for any Simulink block. See "Adding Requirement Links to an Object" on page 2-7 for details on linking requirements documents to individual Simulink blocks.

You can link requirements documents to test groups and their scheduled Model Verification blocks through the **Requirements** pane of the Verification Manager in the Signal Builder. By default, when you display the Verification Manager in the Signal Builder window, the **Requirements** pane appears, as shown.

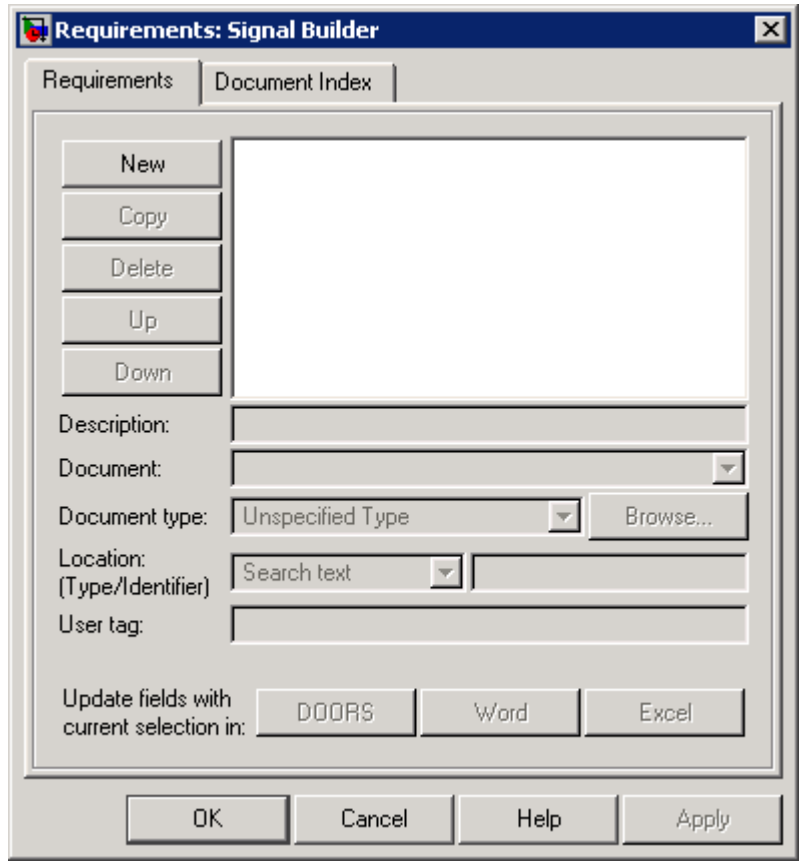


1 Right-click anywhere in the **Requirements** pane.

A pop-up menu appears.

2 From the pop-up menu, select **Edit/Add Links**.

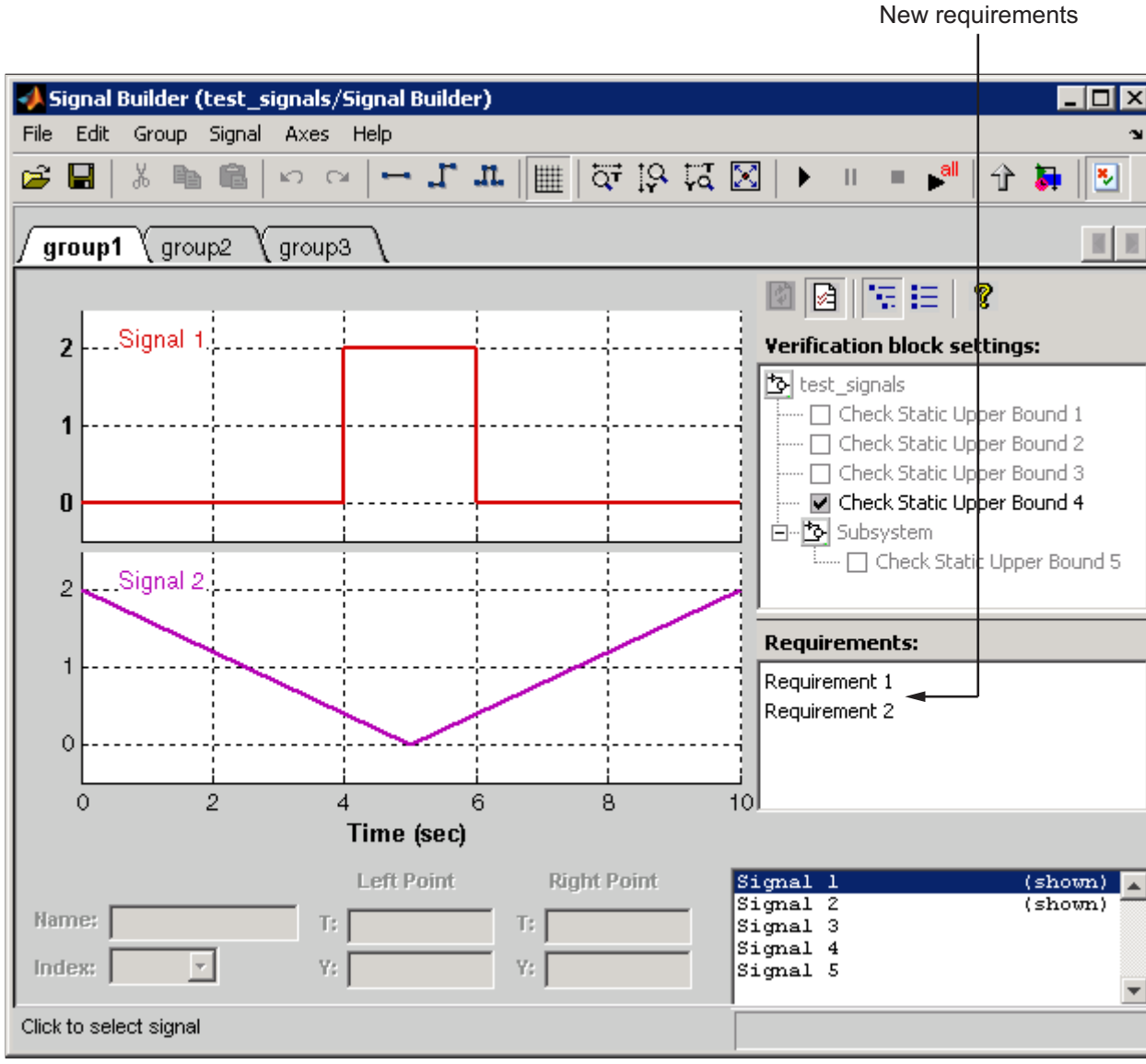
The Requirements dialog box appears, as shown.



You can also access the Requirements dialog box for a Signal Builder block by right-clicking it in the Simulink model and selecting **Requirements > Edit/Add Links**.

3 Add links to requirements documents as described in steps 4 through 9 of "Adding Requirement Links to an Object" on page 2-7.

The descriptions for the links that you add appear in the **Requirements** pane, as shown.



4 Right-click a requirement link and select **View** to view the requirements document in its native editor.

5 Right-click a requirement link and select **Delete** to delete it.

5

Using Model Coverage

Model coverage helps you validate your model tests by measuring how thoroughly the model objects are tested. The following sections describe Simulink Verification and Validation tools that measure and display model coverage for the model.

Introduction to Model Coverage

In this section...

What Is Model Coverage?

Model coverage determines the extent to which a model test case exercises simulation pathways through a model. The percentage of pathways that a test case exercises is called its *model coverage*. Model coverage is a measure of how thoroughly a test tests a model. Model coverage helps you validate your model tests.

How Model Coverage Works

Model coverage works by analyzing the execution of blocks that directly or indirectly determine simulation pathways through your model. If a model includes Stateflow charts, the tool also analyzes the states and transitions of those charts. During a simulation run, the tool records the behavior of the covered blocks, states, and transitions. At the end of the simulation, the tool reports the extent to which the run exercised potential simulation pathways through each covered block.

For an example of a model coverage report, along with descriptions of the coverages it contains, see "Understanding Model Coverage Reports" on page 5-26. Before you do, you might need to review the types of coverages that model coverage performs in "Types of Model Coverage" on page 5-2.

Types of Model Coverage

Simulink Verification and Validation software performs several types of coverage analysis, depending on the coverage options you select.

- "Cyclomatic Complexity" on page 5-3

Cyclomatic Complexity

Cyclomatic complexity is a measure of the structural complexity of a model. It approximates the McCabe complexity measure for code generated from the model. The McCabe complexity measure is slightly higher due to error checks that the model coverage analysis does not consider.

Model coverage uses the following formula to compute the cyclomatic complexity of an object (such as a block, chart, or state):

$$c = \sum_{1}^{N} (o_n - 1)$$

In this formula, N is the number of decision points that the object represents and o_n is the number of outcomes for the nth decision point. The tool adds 1 to the complexity number computed by this formula for atomic subsystems and Stateflow charts.

Decision Coverage (DC)

Decision coverage analyzes elements that represent decision points in a model, such as a Switch block or Stateflow states. For each item, decision coverage determines the percentage of the total number of simulation paths through the item that the simulation actually traversed.

Condition Coverage (CC)

Condition coverage analyzes blocks that output the logical combination of their inputs (for example, the Logic block) and Stateflow transitions. A test case achieves full coverage if it causes each input to each instance of a logic block in the model and each condition on a transition to be true at least

once during the simulation and false at least once during the simulation. Condition coverage analysis reports whether the test case fully covered the block for each block in the model.

Modified Condition/Decision Coverage (MC/DC)

Modified condition/decision coverage analysis by the Simulink Verification and Validation software extends the decision coverage and condition coverage capabilities. It analyzes blocks that output the logical combination of their inputs (for example, the Logic block) and Stateflow transitions to determine the extent to which the test case tests the independence of logical block inputs and transition conditions:

- A test case achieves full coverage for a block if, for every input, there is a pair of simulation times when changing only that input causes a change in the block's output.
- A test case achieves full coverage for a transition if, for each condition on the transition, there is at least one time when a change in the condition triggers the transition.

Because the Simulink Verification and Validation modified condition/decision coverage does not guarantee full decision or condition coverage, you can achieve 100% modified condition/decision coverage *without* achieving 100% decision coverage.

Some Simulink blocks support model condition/decision coverage, some blocks support only condition coverage, and some blocks support only decision coverage. The table "Blocks That Receive Model Coverage" on page 5-5 lists which blocks can receive which types of model coverage. For example, the Logic block can receive condition coverage and modified condition/decision coverage, but not decision coverage.

To achieve 100% modified condition/decision coverage for your model, as defined by the DO-178B standard, collect coverage for all of the following coverage metrics in the Coverage Settings dialog box:

- **Condition Coverage**
- **Decision Coverage**
- **MCDC Coverage**

If you run the test cases independently and accumulate all the coverage results, you can determine if your model adheres to the modified condition/decision coverage standard.

For more information about the DO-178B standard, see "DO-178B Checks" on page 11-4.

Lookup Table Coverage (LUT)

Lookup table coverage examines blocks, such as the Lookup Table block, that output the result of looking up one or more inputs in a table of inputs and outputs, interpolating between or extrapolating from table entries. Lookup table coverage records the frequency that table lookups use each interpolation interval. A test case achieves full coverage if it executes each interpolation and extrapolation interval at least once. For each lookup table block in the model, the coverage report displays a colored map of the lookup table that indicates where each interpolation was performed.

Note Configure lookup table coverage only at the start of a simulation. If you tune a parameter that affects lookup table coverage at run time, the coverage settings for the affected block are not updated.

Blocks That Receive Model Coverage

The following table lists the Simulink blocks analyzed by the tool and the type of coverage analysis for each block.

Block	Decision	Condition	MC/DC	LUT
1D Lookup				•
2D Lookup				•
ND Lookup				•
Interpolation Using Prelookup				•
ND Direct Lookup				•
Abs	•			

Block	Decision	Condition	MC/DC	LUT
Combin. Logic	•	•		
Discrete-Time Integrator (when saturation limits are enabled)	•			
Embedded MATLAB Function	•	•	•	
Fcn (Boolean operators only)		•		
For	•			
If	•			
Logic		•	•	
MinMax	•			
Model	•	•	•	•
Multiport Switch	•			
Rate Limiter	• (Relative to slew rates)			
Relay	•			
Saturation	•			
Stateflow (see note below)	•	•	•	
Subsystem	•	•	•	
Switch	•			
SwitchCase	•			
While	•			

Note Model coverage provides decision coverage for Stateflow states, events, and state temporal logic decisions. It also provides decision, condition, and MCDC coverage for Stateflow transitions. For more information, see "Understanding Model Coverage for Stateflow Charts" in the Stateflow documentation.

Analyzing Model Coverage

In this section...

Model Coverage Analysis Workflow

To develop effective tests with model coverage:

1. Develop one or more test cases for your model. (See "Creating and Running Test Cases" on page 5-8.)
2. Run the test cases to verify that the model behavior is correct.
3. Analyze the coverage reports produced by the Simulink Verification and Validation software.
4. Using the information in the coverage reports, modify the test cases to increase their coverage or add new test cases that cover areas not covered by the current set of test cases.
5. Repeat the preceding steps until you are satisfied with the coverage of your test set.

Note The Simulink Verification and Validation software comes with an online demonstration of model coverage to validate model tests. To run the demo, enter `simcovdemo` at the MATLAB prompt.

Creating and Running Test Cases

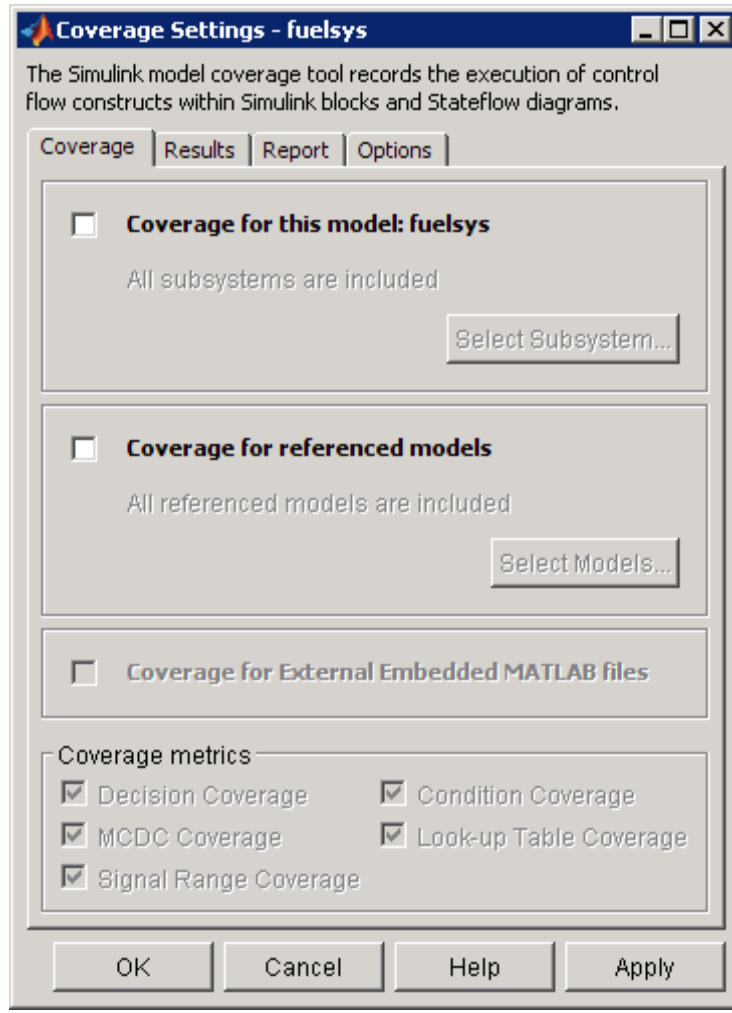
Model coverage provides two MATLAB commands, `cvtest` and `cvsim`, for creating and running test cases. The `cvtest` command creates test cases that the `cvsim` command runs. (See "Creating Tests with cvtest" on page 5-62 and "Running Tests with cvsim" on page 5-64.)

You can also run the coverage tool interactively as follows:

1 Open the `fuelsys` Simulink model.

2 In the Simulink model window, select **Tools > Coverage Settings**.

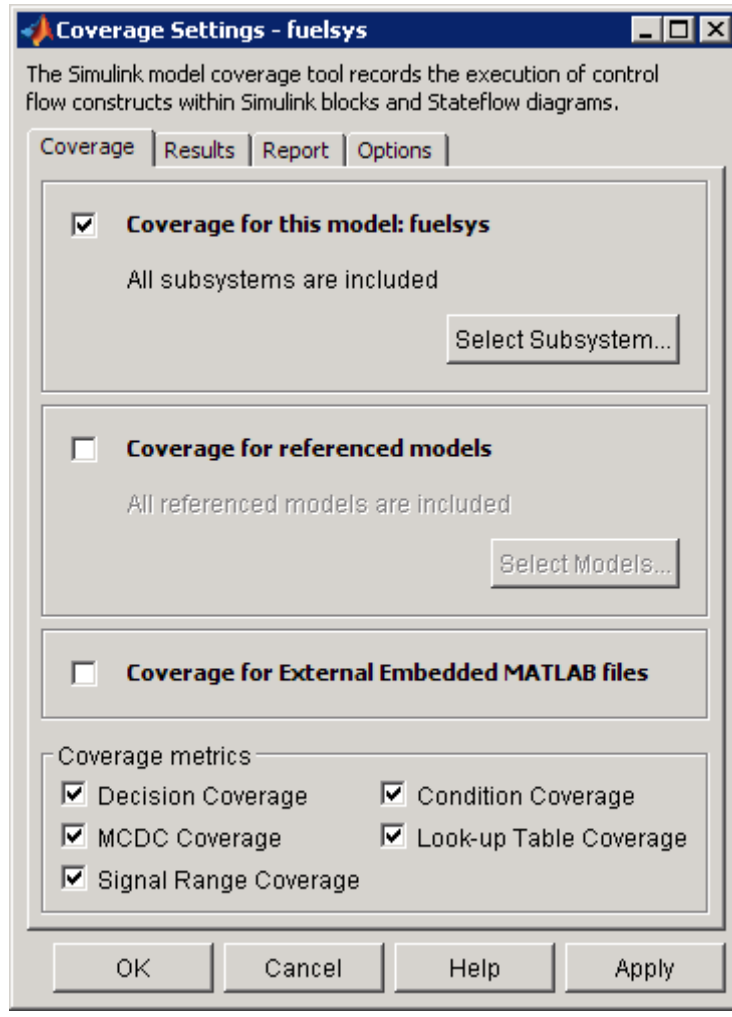
The Coverage Settings dialog box Coverage tab is displayed by default.



3 Select **Coverage for this model**.

Selecting this option enables:

- The **Select Subsystem** button
- The **Coverage for External Embedded MATLAB Files** option
- Check boxes in the **Coverage metrics** section
- Fields on the other tabs of the Coverage Settings dialog box



4 In the **Coverage metrics** group box, select the coverages you want to appear in the coverage report.

   For a complete inventory of coverage selections in all four tabs of the Coverage Settings dialog box, see "Model Coverage Reporting Options" on page 5-12.

5 Click **OK** to close the dialog box.

6 In the Simulink model window, select **Start > Simulation** or click the **Start** button on the Simulink toolbar to start simulating the model.

   If you specify to report model coverage, the Simulink Verification and Validation software saves coverage data for the current run in the workspace object `covdata` and cumulative coverage data in `covCumulativeData` by default. This data appears in an HTML report that opens in a browser window at the end of simulation.

Note You cannot run simulations if you select both model coverage reporting and acceleration options. The Simulink Verification and Validation software clears the model coverage reporting option if you select acceleration mode.

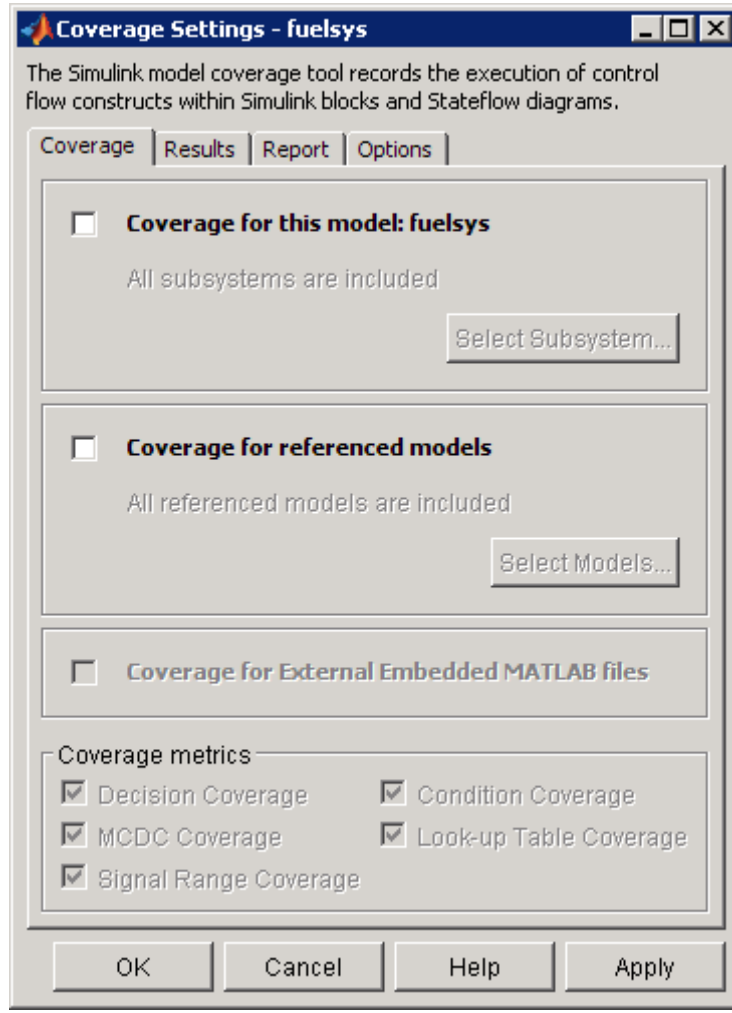
Block reduction optimization and conditional branch input optimization are disabled when you perform coverage analysis because they interfere with coverage recording.

Model Coverage Reporting Options

In this section...

Coverage Settings Dialog Box

Before starting a model coverage analysis, you must specify model coverage reporting options. In a Simulink model window, select **Tools > Coverage Settings**. The Coverage Settings dialog box appears, with the **Coverage** pane displayed.



The following sections describe the settings for each tab of the Coverage Settings dialog box.

Coverage Tab

In the Coverage Settings dialog box, on the **Coverage** tab, select the model coverages calculated during simulation.

Coverage for this model

Causes the Simulink Verification and Validation software to gather and report the model coverages that you specify during simulation. When you select the **Coverage for this model** option, the **Select Subsystem** button and the **Coverage metrics** section of the **Coverage** pane become available.

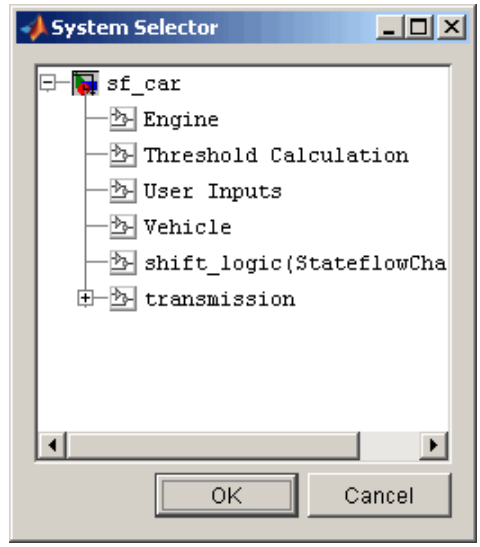
Select Subsystem

Specifies the subsystem for which the Simulink Verification and Validation software gathers and reports coverage data. When you select the **Coverage for this model** option, the software, by default, generates coverage data for the entire model.

To restrict coverage reporting to a particular subsystem:

1. In the Coverage Settings dialog box, on the**Coverage** tab, click **Select Subsystem**.

   The System Selector dialog box appears.



2. Select the subsystem for which you want to enable coverage reporting and click **OK**.

Coverage for referenced models

Causes the Simulink Verification and Validation software to gather and report the model coverages that you specify for referenced models during simulation. When you select the **Coverage for referenced models** option, the **Select Models** button and the **Coverage metrics** section of the **Coverage** tab become available.

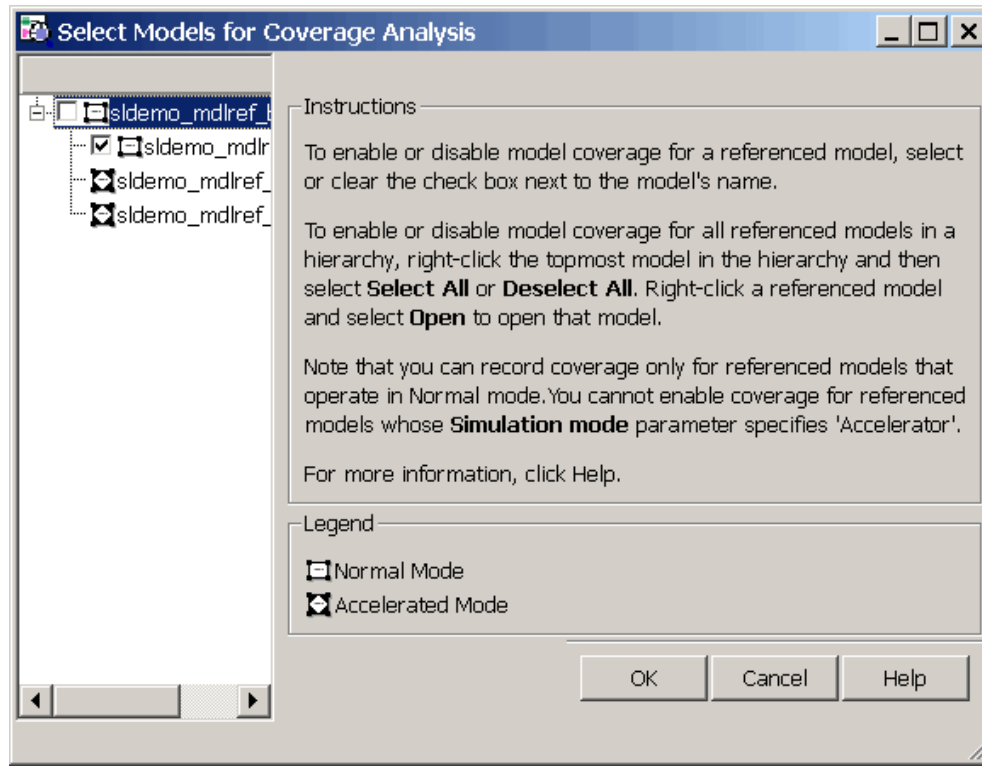
Select Models

Specifies the referenced models for which the Simulink Verification and Validation software gathers and reports coverage data. When you select the **Coverage for referenced models** option, the software, by default, generates coverage data for all referenced models.

To enable coverage reporting for particular referenced models:

1. In the **Coverage** pane of the Coverage Settings dialog box, click **Select Models**.

   The Select Models for Coverage Analysis dialog box appears.



2 Select the referenced models for which you want coverage reporting and click **OK**.

Note The Simulink Verification and Validation software provides model coverage support only for referenced models that operate in Normal mode. The software cannot record coverage for Model blocks whose **Simulation mode** parameter specifies `Accelerator`.

Coverage for External Embedded MATLAB Files

Enables coverage for any external M-file functions that Embedded MATLAB™ functions call in your model. The Embedded MATLAB functions may be defined in an Embedded MATLAB Function block or in a Stateflow chart.

You must select **Coverage for this model** or **Coverage for referenced models** to be able to select the **Coverage for External Embedded MATLAB Files** option.

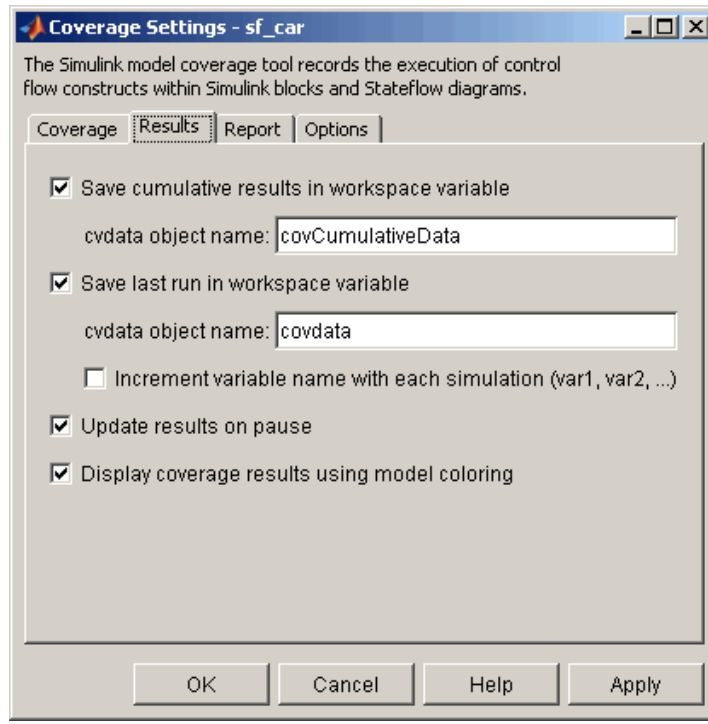
Coverage metrics

Select the types of test case coverage analysis that you want the tool to perform (see "Types of Model Coverage" on page 5-2). The Simulink Verification and Validation software gathers and reports the selected types of coverage for the subsystem, model, and referenced models that you specified elsewhere on the **Coverage** tab.

Note To specify different types of coverage analysis for each of the referenced models in a hierarchy, use the `cv.cvtestgroup` and `cvsimref` functions. For more information, see "Using Model Coverage Commands for Referenced Models" on page 5-69.

Results Tab

In the Coverage Settings dialog box, on the **Results** pane, select the destination of model coverage results.



Save Cumulative Results in Workspace Variable

Causes model coverage to accumulate and save the results of successive simulations in the workspace variable that you specify in the **cvdata object name** field. (By default, the `cvdata` object name is `covCumulativeData`.) The coverage running total in the workspace variable is updated with new results at the end of each simulation.

Save Last Run in Workspace Variable

Causes model coverage to save the results of the last simulation run in the workspace variable that you specify in the **cvdata object name** field below. (By default, the `cvdata` object name is `covdata`.)

Increment Variable Name with Each Simulation

Causes the Simulink Verification and Validation software to increment the name of the coverage data object variable used to save the last run with each simulation. This prevents the current simulation run from overwriting the results of the previous run.

Update Results on Pause

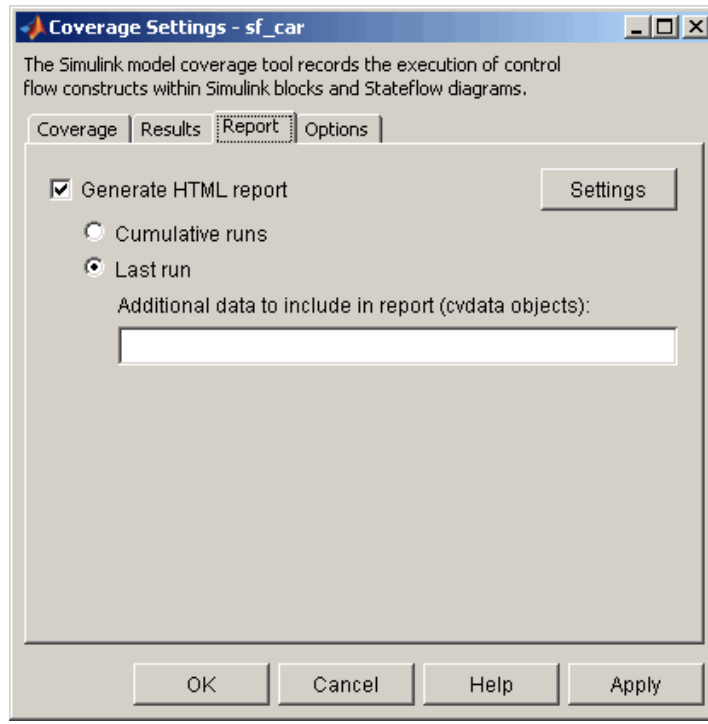
Causes the HTML model coverage report to appear with model coverage results recorded up to the pause point when you pause during simulation for the first time. When you resume simulation and later pause or stop simulation, the model coverage report reappears in updated form with coverage results up to the current pause or stop time.

Display Coverage Results Using Model Coloring

Causes coloring of Simulink blocks according to their level of model coverage, after simulation. Blocks highlighted in light green received full coverage during testing. Blocks highlighted in light red received incomplete coverage. In addition, model coverage results for each block are available in context-sensitive form. See “Colored Simulink Diagram Coverage Display” on page 5-57.

Report Tab

in the Coverage Settings dialog box, on the **Report** tab, select the model coverage test sessions (runs) reported by model coverage .

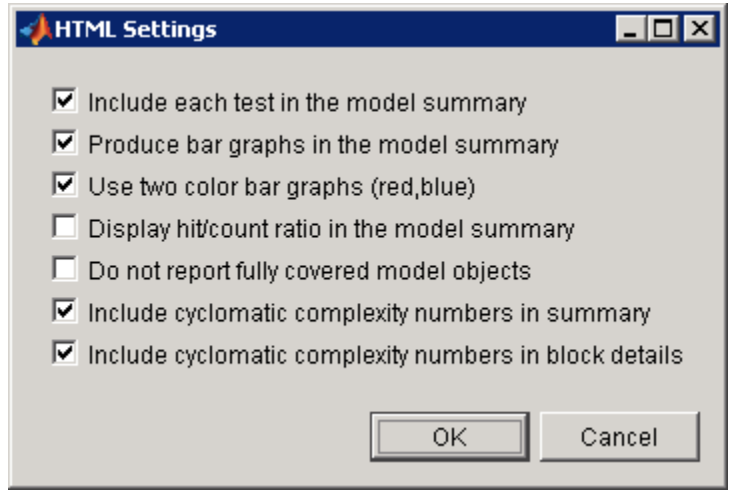


Generate HTML Report

Causes the Simulink Verification and Validation software to create an HTML report containing the coverage data. The software displays the report in the MATLAB Help browser at the end of the simulation. Click the **Settings** button to select various reporting options (see "Settings" on page 5-20).

Settings

In the Coverage Settings dialog box, on the **Report** tab, click **Settings** to open the HTML Settings dialog box. In the HTML Settings dialog box, choose the desired model coverage report options.



Include each test in the model summary. When you select this option, the model hierarchy table at the top of the HTML report includes columns listing the coverage metrics for each test. If you do not select this option, the model summary reports only the total coverage.

Produce bar graphs in the model summary. Causes the model summary to include a bar graph for each coverage result. The bar graphs provide a visual representation of the coverage.

Use two color bar graphs (red, blue). Causes the report to use red and blue bar graphs instead of black and white. The color graphs might not print well in black and white.

Display hit/count ratio in the model summary. Reports coverage numbers as both a percentage and a ratio, for example, 67% (8/12).

Do not report fully covered model objects. Causes the coverage report to include only model objects that the simulation does not cover fully. This option is useful when you are developing tests, because it reduces the size of the generated reports.

Include cyclomatic complexity numbers in summary. Includes the cyclomatic complexity (see "Types of Model Coverage" on page 5-2) of the model and its top-level subsystems and charts in the report summary. A cyclomatic complexity number shown in boldface indicates that the analysis considered the subsystem itself to be an object when computing its complexity. This occurs for atomic and conditionally executed subsystems as well as Stateflow Chart blocks.

Include cyclomatic complexity numbers in block details. Includes the cyclomatic complexity metric in the block details section of the report.

Cumulative Runs

Display the coverage results from successive simulations in the report. For more information about this report, see "Save Cumulative Results in Workspace Variable" on page 5-18.

If you select the **Save cumulative results in workspace variable** check box on the **Results** tab, a coverage running total is updated with new results at the end of each simulation. However, if you change model or block settings between simulations that are incompatible with settings from previous simulations and affect the type or number of coverage points, the cumulative coverage resets.

You can make cumulative coverage results persist between MATLAB sessions by using `cvsave` to save results to a file at the end of the session and `cvload` to load the results at the beginning of the session. The `cvload` parameter `RESTORETOTAL` must be `1` in order to restore cumulative results.

When you save the coverage results to a file using `cvsave` and a model name argument, the file also contains the cumulative running total. When you load that file back into the coverage tool using `cvload`, you can select whether you want to restore the running total from the file.

When you restore a running total from saved data, the saved results are reflected in the next cumulative report that is generated. If a running total already exists when you restore a saved value, the existing value is overwritten.

Whenever you report on more than one single simulation, the coverage displayed for truth tables and lookup-table maps is based on the total coverage of all the reported runs. In the case of a cumulative report, this includes all the simulations where cumulative results were stored.

You can also calculate cumulative coverage results at the command line via the + operator. The following script demonstrates this usage:

```
covdata1 = cvsim(test1);
covdata2 = cvsim(test2);
cvhtml('cumulative_report', covdata + covdata2);
```

Last Run

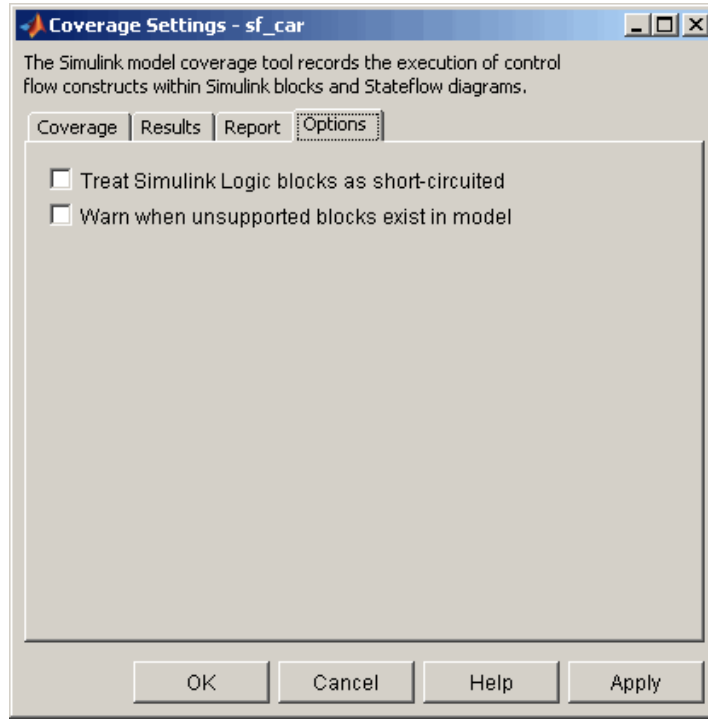
Display only the results of the most recent simulation run in the report.

Additional Data to Include in Report

Specify names of coverage data from previous runs to include in the current report along with the current coverage data. Each entry causes a new set of columns to appear in the report.

Options Tab

In the Coverage Settings dialog box, on the **Options** tab, select important options for model coverage reports.



Treat Simulink Logic Blocks as Short-Circuited

Applies only to condition and MC/DC coverage. If selected, coverage analysis treats Simulink logic blocks as though they short-circuit their input. In other words, the coverage tool treats such a block as if the block ignores remaining inputs when the previous inputs alone determine the block's output. For example, if the first input to a Logical Operator block whose **Operator** parameter specifies `AND` is false, MC/DC coverage analysis ignores the values of the other inputs when determining MC/DC coverage for a test case.

Select this option if you plan to generate code from a model and want the MC/DC coverage analysis to approximate the degree of coverage that your test cases would achieve for the generated code (most high-level languages short-circuit logic expressions).

Note A test case that does not achieve full MC/DC coverage for non-short-circuited logic expressions might achieve full coverage for short-circuited expressions.

Warn When Unsupported Blocks Exist in a Model

Select this option if you want the tool to warn you at the end of the simulation if the model contains blocks that require coverage analysis but are not currently covered by the tool.

Understanding Model Coverage Reports

In this section...

Types of Coverage Reports

In the Coverage Settings dialog box, on the **Report** tab, if you enable the **Generate HTML report** option, the Simulink Verification and Validation software creates one or more model coverage reports after it completes a simulation.

Report Type	Description	HTML Report File Name
"Model Coverage Reports" on page 5-27	The software creates this report whenever you generate coverage reports for a model. This report provides coverage information for all model elements, including the model itself.	*model_name*_cov.html
"Model Summary Reports" on page 5-49	The software creates this report when the top-level model includes Model blocks or calls one or more external M-files. This report provides links to coverage results for all referenced models and external M-files in the model hierarchy.	*model_name*_summary_cov.html

Report Type	Description	HTML Report File Name
"Model Reference Coverage Reports" on page 5-50	In addition to the model coverage report for the top-level model, the software creates this report for each referenced model in the model hierarchy. This report has the same format as the model coverage report.	*reference_model_name*_cov.html
"External M-File Coverage Reports" on page 5-50	In addition to the model coverage report for the top-level model, this report provides detailed coverage information about any external M-file that the model calls. There is one report for each M-file called.	*M-file_name*_cov.html
"Subsystem Coverage Reports" on page 5-54	If you select a subsystem in the Coverage Settings dialog box, the model coverage report includes only coverage results for that subsystem.	*model_name*_cov.html, where *model_name* is the name of the top-level model

Model Coverage Reports

The Simulink Verification and Validation software always creates a model coverage report for the top-level model named *model_name*_cov.html. The model coverage report contains several sections:

- "Coverage Summary" on page 5-28
- "Details" on page 5-29
- "Decisions Analyzed" on page 5-34
- "Conditions Analyzed" on page 5-35
- "MC/DC Analysis" on page 5-36

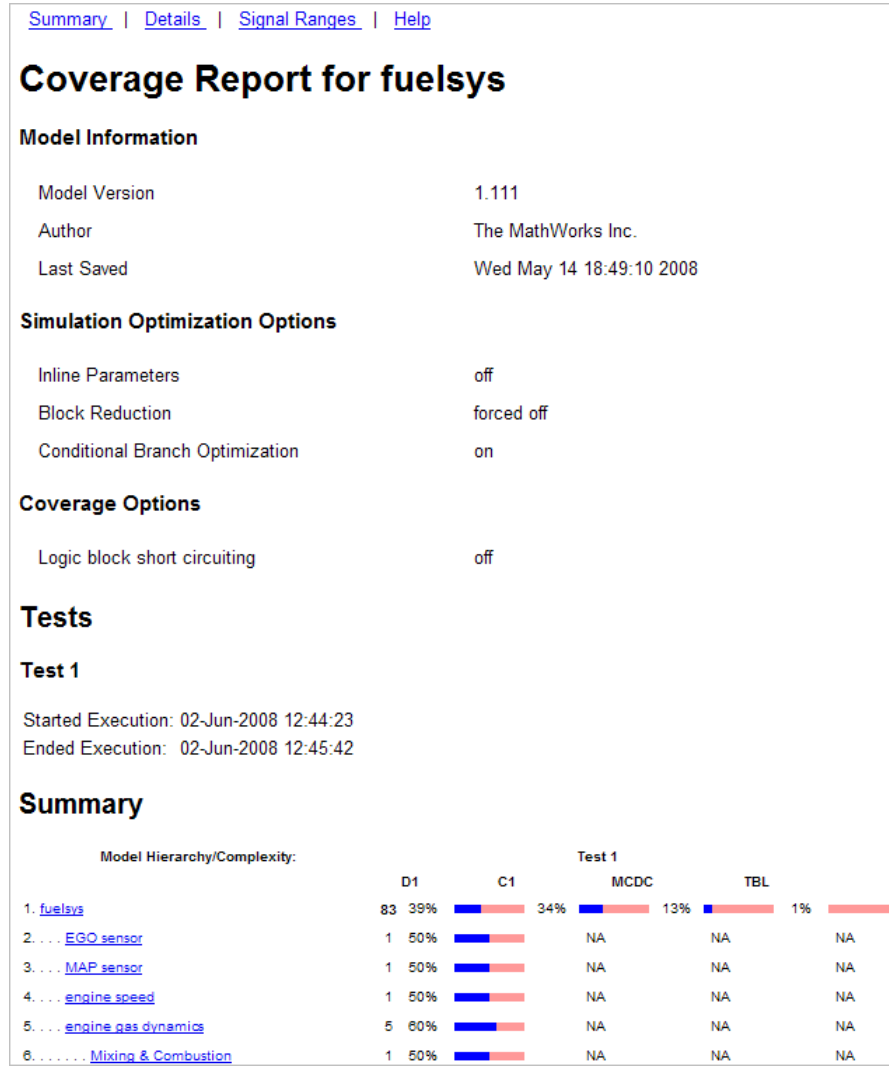
Coverage Summary

The coverage summary section contains basic information about the model being analyzed:

- **Model Information**
- **Simulation Optimization Options**
- **Coverage Options**

The coverage summary has two subsections:

- **Tests** — The simulation start and stop time of each test case and any setup commands that preceded the simulation. The heading for each test case includes any test case label specified using the `cvtest` command.
- **Summary** — Summaries of the subsystem results. To see detailed results for a specific subsystem, click the subsystem name in the Summary subsection.



Summary | Details | Signal Ranges | Help

Coverage Report for fuelsys

Model Information

Model Version	1.111
Author	The MathWorks Inc.
Last Saved	Wed May 14 18:49:10 2008

Simulation Optimization Options

Inline Parameters	off
Block Reduction	forced off
Conditional Branch Optimization	on

Coverage Options

Logic block short circuiting	off

Tests

Test 1

Started Execution: 02-Jun-2008 12:44:23
Ended Execution: 02-Jun-2008 12:45:42

Summary

Model Hierarchy/Complexity:	Test 1								
	D1		C1		MCDC		TBL		
1. fuelsys	83	39%		34%		13%		1%	
2. . . . EGO sensor	1	50%		NA		NA		NA	
3. . . . MAP sensor	1	50%		NA		NA		NA	
4. . . . engine speed	1	50%		NA		NA		NA	
5. . . . engine gas dynamics	5	60%		NA		NA		NA	
6. . . . . Mixing & Combustion	1	50%		NA		NA		NA	

Details

The Details section reports the model coverage results in detail. Each section of the detailed report summarizes the results for the metrics used to test each object in the model:

You can also access a model element's Details subsection as follows:

1. Right-click a Simulink element.
2. In the pop-up menu, select **Coverage > Report**.

   The model coverage report appears at the applicable Details subsection.

Model Details. The Details section contains a summary of results for the model as a whole, followed by a list of elements. Subsections for each subsystem and chart follow. Click the model element name to see its coverage results.

The following graphic shows the Details section for the `fuelsys` model.

Details:

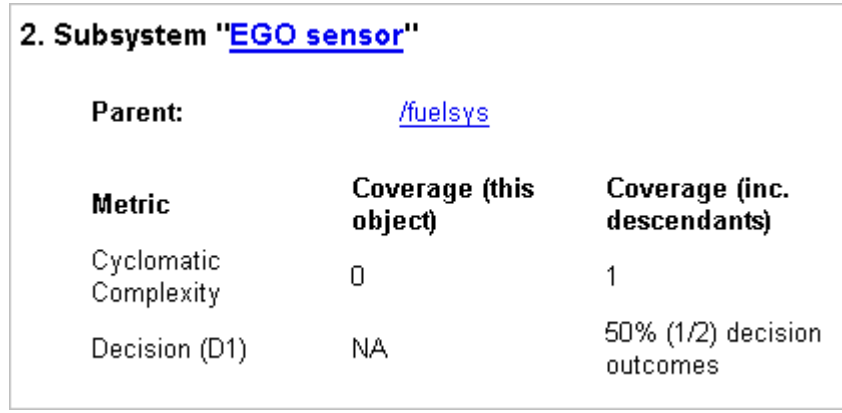
1. Model "fuelsys"

Child Systems: EGO sensor, MAP sensor, engine speed, engine gas dynamics, fuel rate controller, speed sensor, throttle command, throttle sensor

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	**1**	**83**
Decision (D1)	NA	39% (53/135) decision outcomes
Condition (C1)	NA	34% (11/32) condition outcomes
MCDC (C1)	NA	13% (2/16) conditions reversed the outcome
Look-up Table	NA	1% (15/1508) interpolation/extrapolation intervals

Subsystem Details. Each subsystem Details section contains a summary of the test coverage results for the subsystem and a list of the subsystems it contains. The overview is followed by sections for blocks, charts, and Embedded MATLAB functions, one for each object that contains a decision point in the subsystem.

The following graphic shows the coverage results for the EGO sensor subsystem in the `fuelsys` model.

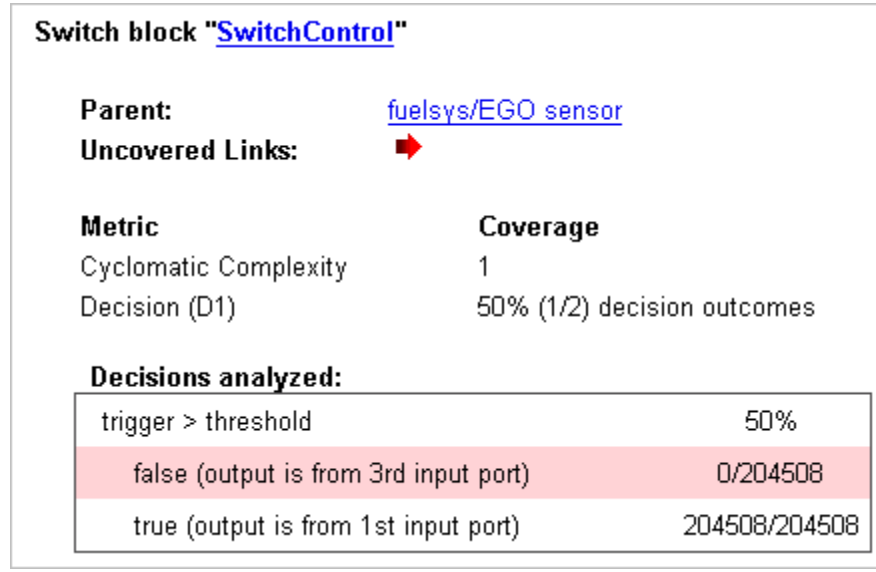


2. Subsystem "EGO sensor"

Parent: /fuelsys

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	1
Decision (D1)	NA	50% (1/2) decision outcomes

Block Details. The following graphic shows the coverage results for the Switch block in the EGO sensor subsystem of the `fuelsys` model.



Switch block "SwitchControl"

Parent: fuelsys/EGO sensor

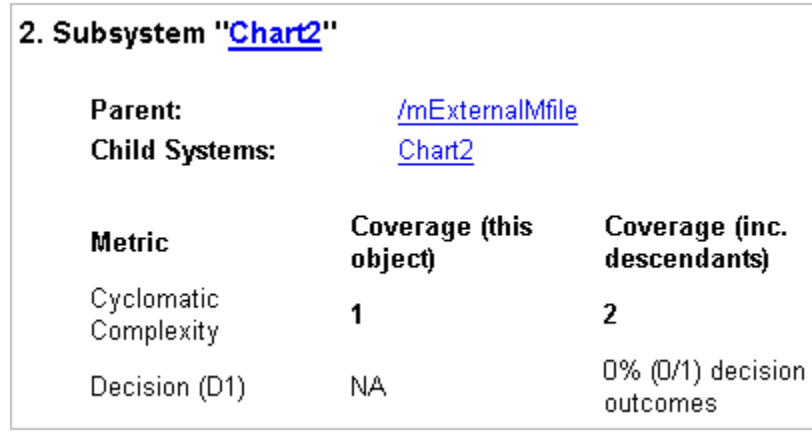
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	50% (1/2) decision outcomes

Decisions analyzed:

trigger > threshold	50%
false (output is from 3rd input port)	0/204508
true (output is from 1st input port)	204508/204508

Chart Details. The following graphic shows the coverage results for the Stateflow chart `Chart2` in the `mExternalMfile` model.



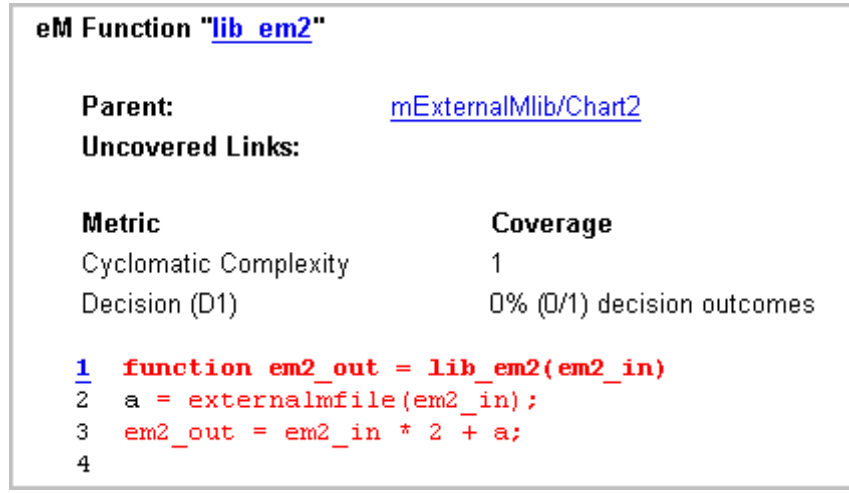
2. Subsystem "Chart2"

Parent:	/mExternalMfile
Child Systems:	Chart2

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	**1**	**2**
Decision (D1)	NA	0% (0/1) decision outcomes

For more information about model coverage reports for Stateflow charts and their objects, see "Understanding Model Coverage for Stateflow Charts" in the Stateflow documentation.

Embedded MATLAB Function Details. The following graphic shows coverage results for the `lib_em2` function call in the `Chart2` Stateflow chart of the `mExternalMfile` model.



eM Function "lib_em2"

Parent:	mExternalMlib/Chart2
Uncovered Links:	

Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	0% (0/1) decision outcomes

```
1  function em2_out = lib_em2(em2_in)
2  a = externalmfile(em2_in);
3  em2_out = em2_in * 2 + a;
4
```

For more information about coverage for Embedded MATLAB functions, see "Model Coverage for Embedded MATLAB Function Blocks" on page 5-74.

Decisions Analyzed

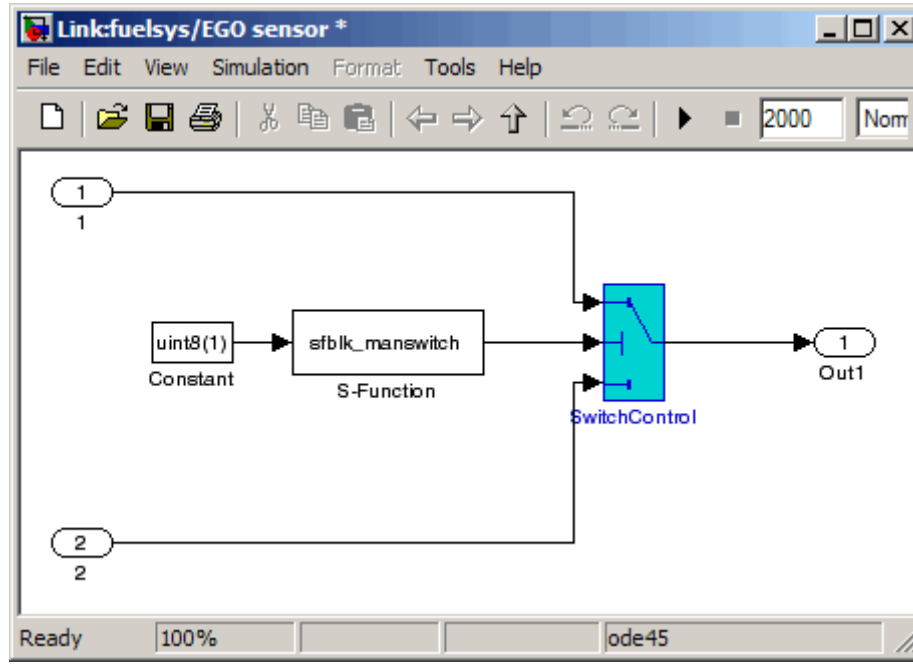
The Decisions analyzed table lists possible outcomes for a decision and the number of times that an outcome occurred in each test simulation. Outcomes that did not occur are in red highlighted table rows.

The following graphic shows the Decisions analyzed table for the Switch Control block in the EGO sensor subsystem of the `fuelsys` model.

Decisions analyzed:

trigger > threshold	50%
false (output is from 3rd input port)	0/204508
true (output is from 1st input port)	204508/204508

To display and highlight the block in question, click the block name associated with the Decisions analyzed table, as in this example from the `fuelsys` model.



The next graphic shows the Decisions analyzed table for the `lib_em2` function call in `Chart2` of the `MexternalMfile` model.

#1: function em2_out = lib_em2(em2_in)

Decisions analyzed:

function em2_out = lib_em2(em2_in)	0%
executed	0/0

Conditions Analyzed

The Conditions analyzed table lists the number of occurrences of true and false conditions on each input port of the corresponding block.

Conditions analyzed:

Description:	True	False
input port 1	481	199520
input port 2	0	200001

MC/DC Analysis

The MC/DC analysis table lists the MC/DC input condition cases represented by the corresponding block and the extent to which the reported test cases cover the condition cases.

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	True Out	False Out
expression for output		
input port 1	FF	TF
input port 2	FF	(FT)

Each row of the MC/DC analysis table represents a condition case for a particular input to the block. A condition case for input `n` of a block is a combination of input values. Changing the value of input `n` alone is sufficient to change the value of the block's output. Input `n` is called the *deciding input* of the condition case.

The table uses a condition case expression to represent a condition case. A condition case expression is a character string where:

- The position of a character in the string corresponds to the input port number.
- The character at the position represents the value of the input (`T` means `true`, `F` means `false`).
- A boldface character corresponds to the value of the deciding input.

For example, **`FTF`** represents a condition case for a three-input block where the second input is the deciding input.

The **Decision/Condition** column specifies the deciding input for an input condition case. The **#1 True Out** column specifies the deciding input value that causes the block to output a `true` value for a condition case. The **#1 True Out** entry uses a condition case expression, for example, `FF`, to express the values of all the inputs to the block, with the value of the deciding variable indicated in bold.

Parentheses around the expression indicate that the specified combination of inputs did not occur during the first (or only) test case included in this report. In other words, the test case did not cover the corresponding condition case. The **#1 False Out** column specifies the deciding input value that causes the block to output a false value and whether the value actually occurred during the first (or only) test case included in the report.

If you select **Treat Simulink Logic blocks as short-circuited** in the Coverage Settings dialog box (see "Model Coverage Reporting Options" on page 5-12), MC/DC coverage analysis does not verify whether short-circuited inputs actually occur. The MC/DC details table uses an `x` in a condition expression (for example, `TFxxx`) to indicate short-circuited inputs that were not analyzed by the tool.

Navigation Arrows. The section for each block contains a backward and a forward arrow. Click the forward arrow to go to the next section in the report that lists an uncovered outcome. Click the back arrow to return to the previous uncovered outcome in the report.

Cumulative Coverage

On the **Results** tab, if you select **Save cumulative results in workspace variable** and on the **Report** tab, **Cumulative runs**, the results of each simulation are saved and recorded in the report.

In a cumulative coverage report, the right-most results in all tables reflect the running total value. The report is organized so that you can easily compare the additional coverage from the most recent run with the coverage from all prior runs in the session.

A cumulative coverage report contains information about:

- Current Run — The coverage results of the simulation just completed

- Delta — Percentage of additional coverage achieved with the simulation just completed
- Cumulative — The total coverage of the model up to but not including the simulation not completed

After running three test cases for the `slvnv_autopilot_test_harness` model, the Summary report shows how much additional coverage the third test case achieved and the cumulative coverage achieved for the first two test cases.

Summary

Model Hierarchy/Complexity:	Current Run						Delta						Cumulative					
	D1		C1		MCDC		D1		C1		MCDC		D1		C1		MCDC	
1. slvnvdemo_autopilot_test_harness	31	38%		41%		17%		8%		6%		0%		51%		41%		17%
2. . . . Logic	25	34%		38%		17%		9%		8%		0%		47%		38%		17%
3. . . . . SF: Logic	24	34%		38%		17%		9%		8%		0%		47%		38%		17%
4. . . . . SF: Altitude	11	64%		67%		33%		21%		17%		0%		93%		67%		33%
5. . . . . SF: Active	4	38%		NA		NA		13%		NA		NA		88%		NA		NA
6. . . . . SF: GS	13	11%		8%		0%		0%		0%		0%		11%		8%		0%
7. . . . . SF: Active	6	0%		NA		NA		0%		NA		NA		0%		NA		NA
8. . . . . SF: Coupled	3	0%		NA		NA		0%		NA		NA		0%		NA		NA
9. . . . Verify Outputs	5	60%		50%		NA		0%		0%		NA		80%		50%		NA
10. . . . . Subsystem1	1	0%		NA		NA		0%		NA		NA		100%		NA		NA
11. . . . . Capture time	1	0%		NA		NA		0%		NA		NA		100%		NA		NA
12. . . . . Subsystem2	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
13. . . . . Capture time	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
14. . . . . Subsystem3	1	0%		NA		NA		0%		NA		NA		0%		NA		NA
15. . . . . Capture time	1	0%		NA		NA		0%		NA		NA		0%		NA		NA
16. . . . . Verification	2	100%		50%		NA		0%		0%		NA		100%		50%		NA

The Decisions analyzed table for cumulative coverage contains three columns of data about decision outcomes that represent the current run, the delta since the last run, and the cumulative data, respectively.

Decisions analyzed:

Transition trigger expression	100%	50%	100%
false	401/402	0/1	3399/3400
true	1/402	1/1	1/3400

The Conditions analyzed table uses column headers **#n T** and **#n F** to indicate results for individual test cases, and **Tot T** and **Tot F** for the cumulative results. You can identify the true and false conditions on each input port of the corresponding block for each test case.

Conditions analyzed:

Description:	#1 T	#1 F	#2 T	#2 F	Tot T	Tot F
Condition 1, "in(GS.Active.Coupled)"	0	402	0	0	0	3400
Condition 2, "alt_ctrl"	401	1	0	1	3399	1
Condition 3, "wow"	0	401	0	0	0	3399

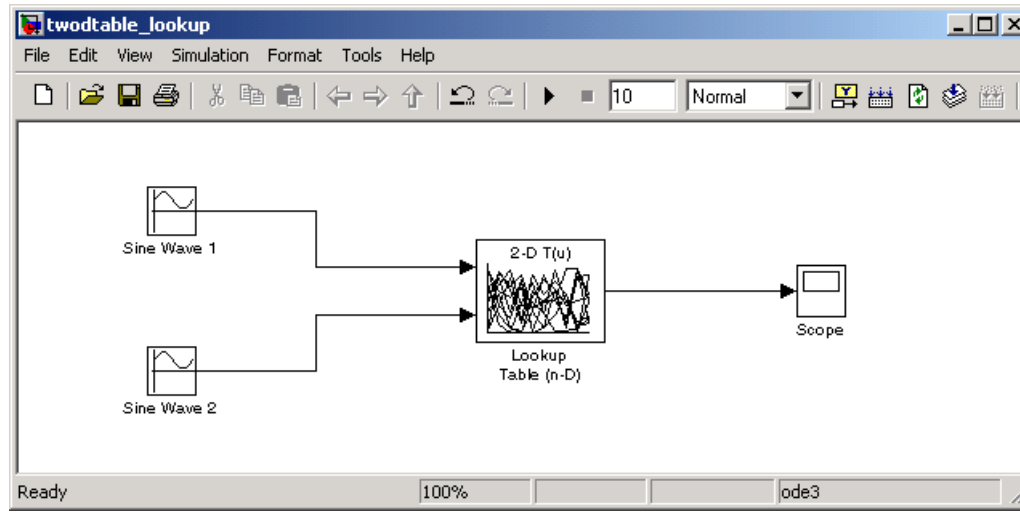
The MC/DC analysis **#n True Out** and **#n False Out** columns show the condition cases for each test case. The **Total Out T** and **Total Out F** column show the cumulative results.

MC/DC analysis (combinations in parentheses did not occur)

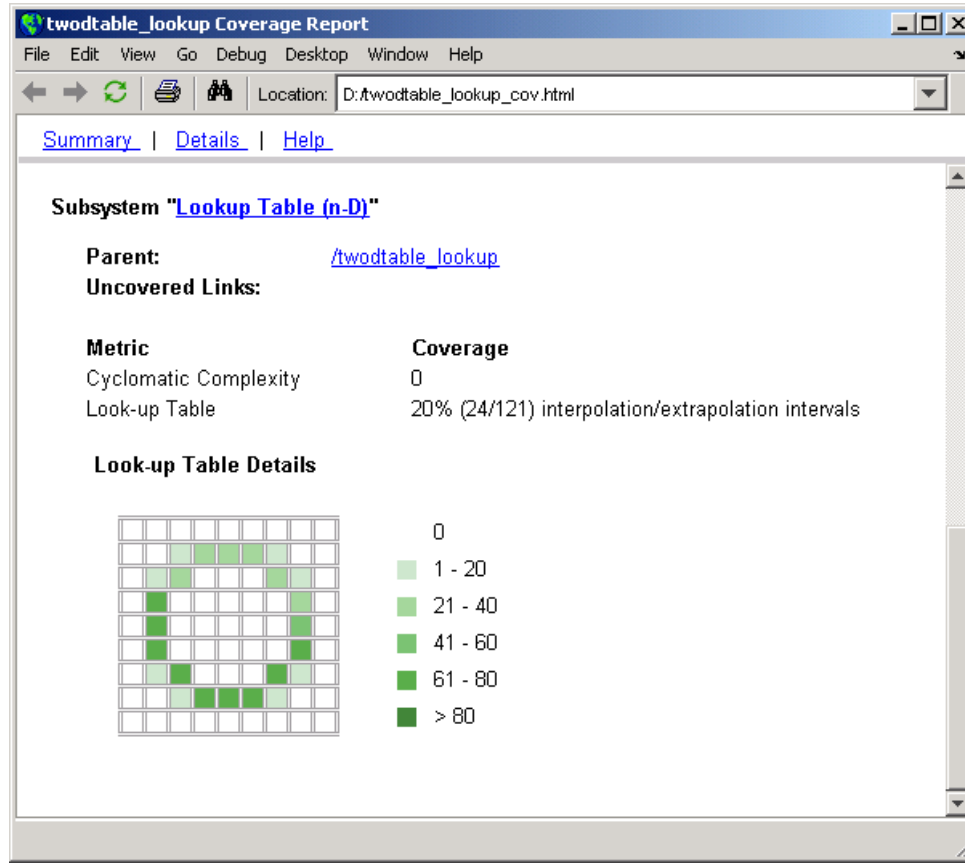
Decision/Condition:	#1 True Out	#1 False Out	#2 True Out	#2 False Out	Total Out T	Total Out F
Transition trigger expression						
Condition 1, "in(GS.Active.Coupled)"	(Txx)	FTF	(Txx)	(FTF)	(Txx)	FTF
Condition 2, "alt_ctrl"	FFx	FTF	FFx	(FTF)	FFx	FTF
Condition 3, "wow"	(FTT)	FTF	(FTT)	(FTF)	(FTT)	FTF

N-Dimensional Lookup Table

This section displays an interactive chart that summarizes the extent to which elements of a lookup table are accessed. In the following example, a Lookup Table (n-D) block of 10-by-10 elements filled with random values is accessed with *x* and *y* indices generated from two Sine Wave blocks.



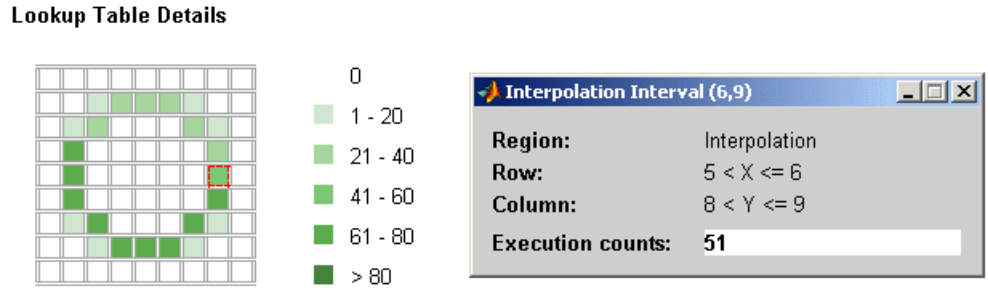
In this example, table indices are 1, 2,..., 10 in each direction. The Sine Wave 2 block is out of phase with the Sine Wave 1 block by `pi`/2 radians. This generates *x* and *y* numbers for the edge of a circle, which you see when you examine the resulting Lookup Table coverage.



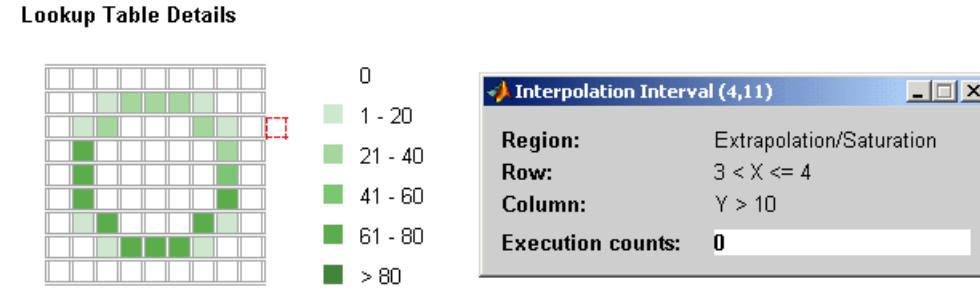
The report contains a two-dimensional table representing the elements of the lookup table. The element indices are represented by the cell border grid lines, which number 10 in each dimension. Areas where the lookup table interpolates between table values are represented by the cell areas. Areas of extrapolation left of element 1 and right of element 10 are represented by cells at the edge of the table, which have no outside border.

The number of values interpolated (or extrapolated) for each cell (*execution counts*) during testing is represented by a shade of green assigned to the cell. Each of six levels of green shading and the range of execution counts represented are displayed on the side of the table.

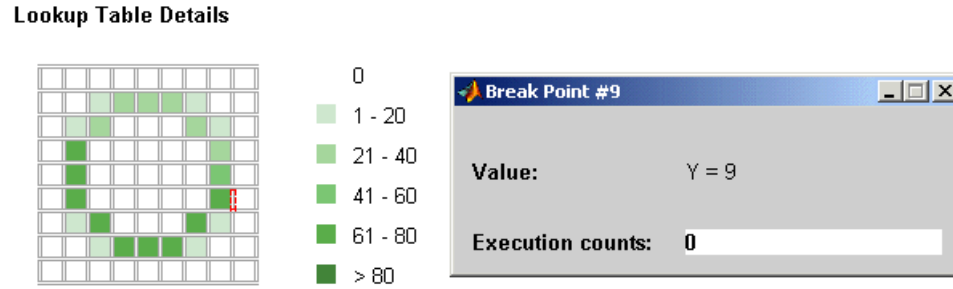
If you click an individual table cell, you see a dialog box that displays the index location of the cell and the exact number of execution counts generated for it during testing. The following example shows the contents of a color-shaded cell on the right edge of the circle:



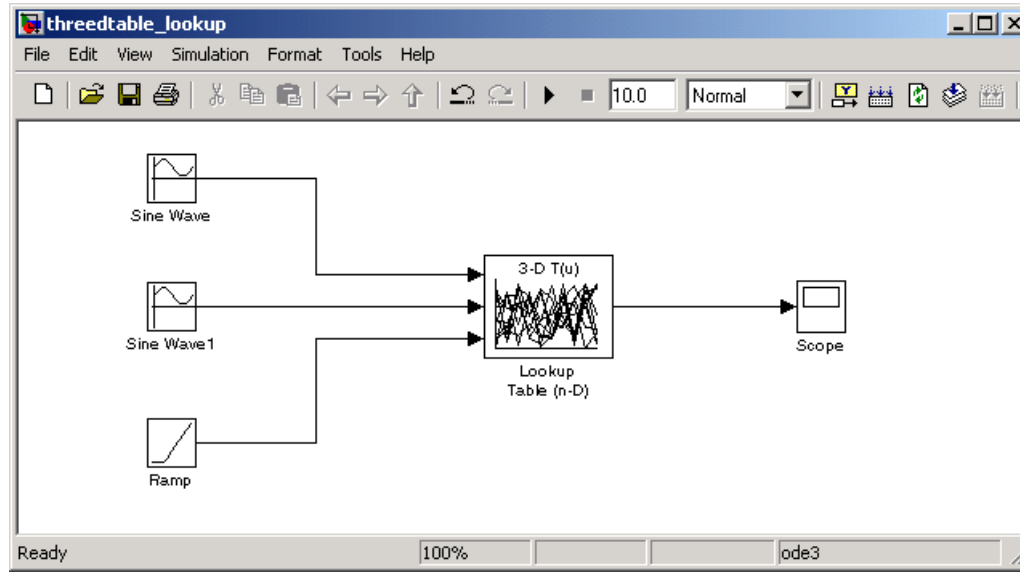
The selected cell is outlined in red. You can also click the extrapolation cells on the edge of the table.



A bold grid line indicates that at least one block input equal to its exact index value occurred during the simulation. Click the border to display the exact number of hits for that index value.

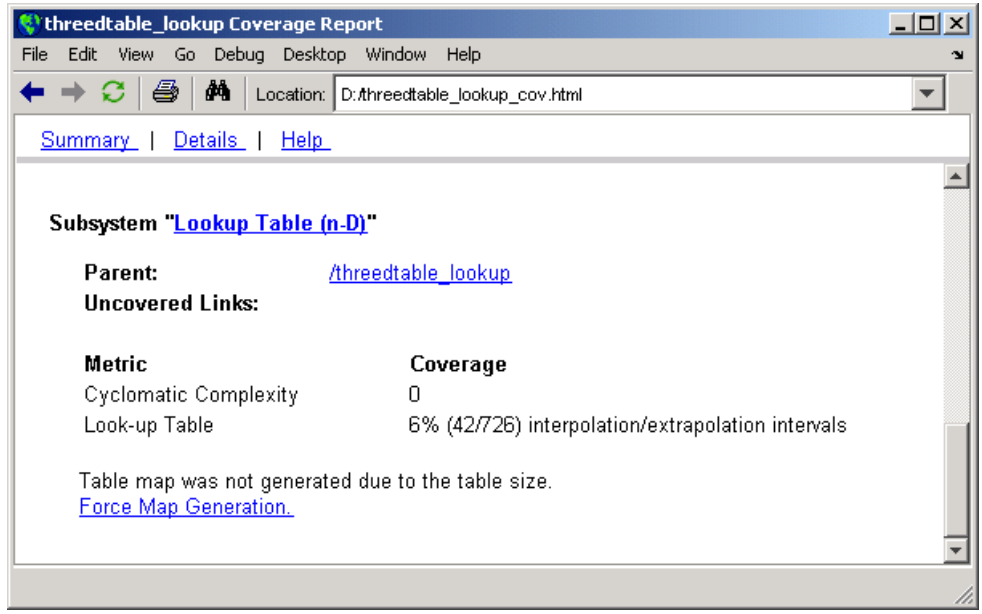


The following example model uses a Lookup Table (n-D) block of 10-by-10-by-5 elements filled with random values.

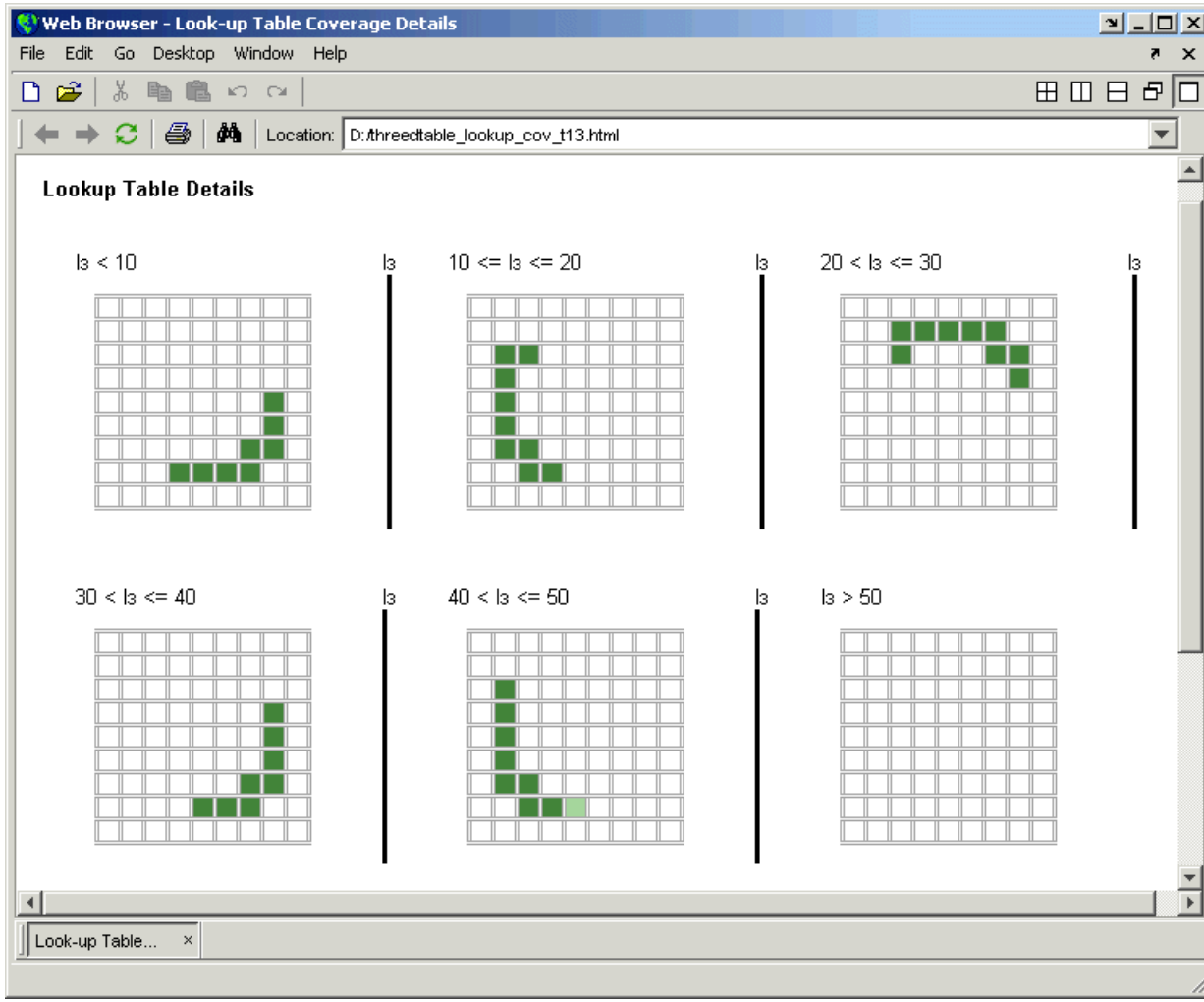


Both the *x* and *y* table axes have the indices 1, 2,..., 10, while the *z* axis has the indices 10, 20,..., 50. Lookup table values are accessed with *x* and *y* indices generated from the two Sine Wave blocks in the preceding example, and a *z* index generated from a Ramp block.

After simulation, the following lookup table report appears.



Instead of a two-dimensional table, the link `Force Map Generation` appears, which displays the following tables:



Lookup table coverage for a three-dimensional lookup table block is reported as a set of two-dimensional tables.

The vertical bars represent the exact *z* index values: 10, 20, 30, 40, 50. If a vertical bar is bold, this indicates that at least one block input was equal to the exact index value it represents during the simulation. Click a bar to get a report of coverage for the exact index value it represents.

You can report lookup table coverage for lookup tables of any dimension. Coverage for four-dimensional tables is reported as sets of three-dimensional sets like those in the preceding example. Five-dimensional tables are reported as sets of sets of three-dimensional sets, and so on.

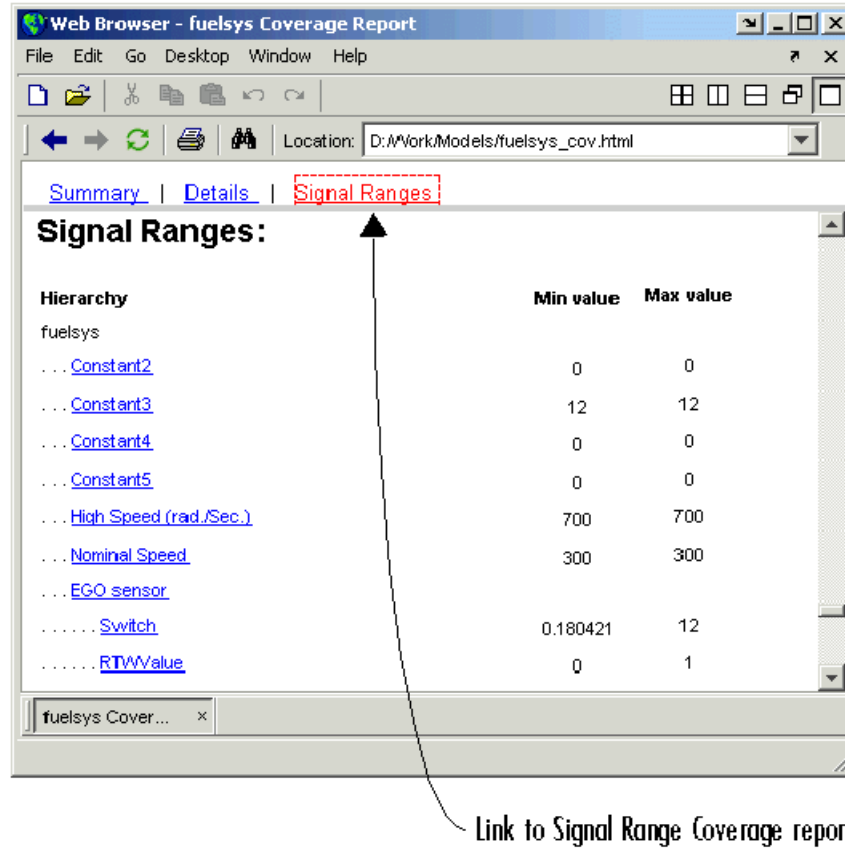
Signal Range Analysis

If you select **Signal Range Coverage** in the Coverage Settings dialog box, the software creates a Signal Range Analysis section at the bottom of the model coverage report. This section shows you the maximum and minimum signal values at each block in the model measured during simulation.

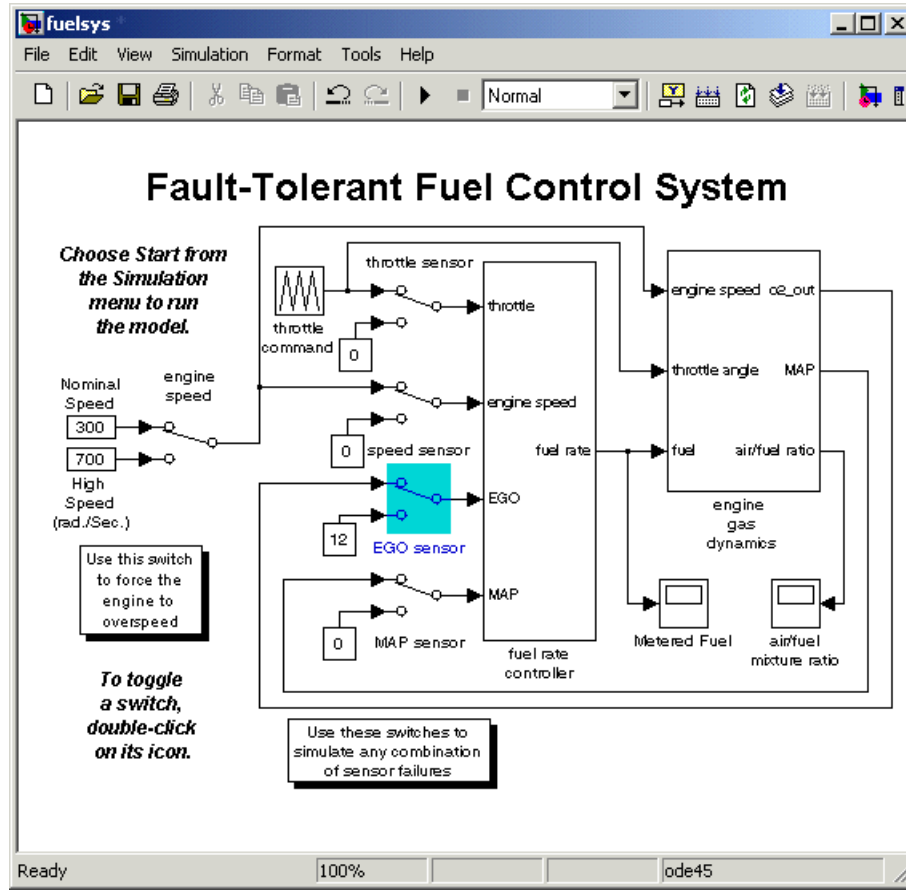
Note When **Inline parameters** is enabled, some signal range information may be missing (for example, if there is a gain with a value of 1). On the **Optimization** tab of the model's active configuration, clear the **Inline parameters** option for a complete signal range report

---

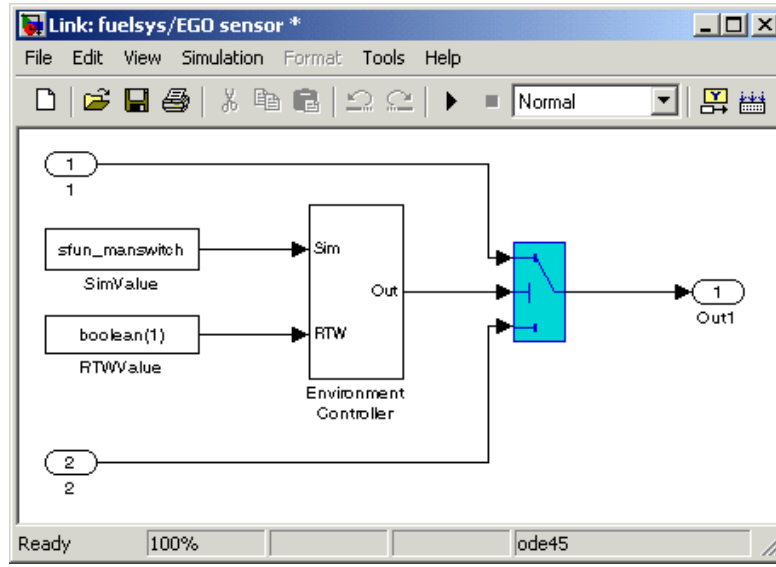
You can access the Signal Range Analysis report quickly with the **Signal Ranges** link in the nonscrolling region at the top of the model coverage report, as shown for the `fuelsys` model.



Each block is reported in hierarchical fashion: child blocks are displayed directly under parent blocks. Each block name in the signal report is a link. For example, selecting the link `EGO sensor` displays this block highlighted in its native diagram, as shown.



Selecting the link `Switch` displays this block in its own subsystem by looking under the mask for `EGO sensor`, as shown.

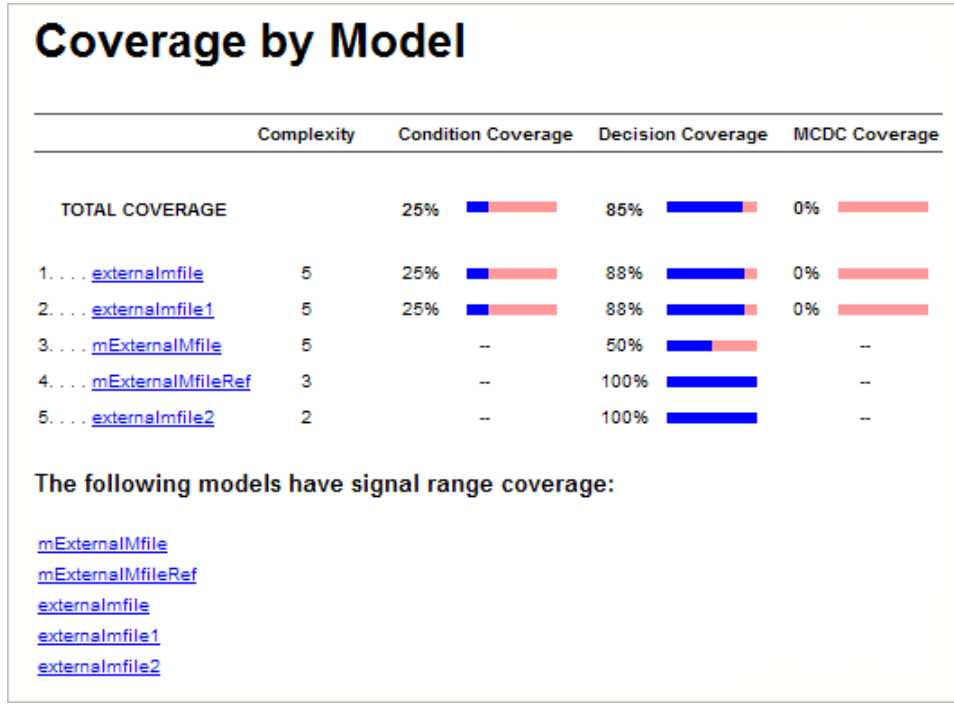


Model Summary Reports

If the top-level model contains Model blocks or calls external M-files, the software creates a model summary coverage report named *model_name*`_summary_cov.html`. The title of this report is **Coverage by Model**.

The summary report lists and provides links to coverage reports for all Model block referenced models and external M-files called by Embedded MATLAB code in the model. For more information, see "Model Reference Coverage Reports" on page 5-50 and "External M-File Coverage Reports" on page 5-50.

The following graphic shows an example of a model summary report. It contains links to the model coverage report (`mExternalMfile`), a report for the Model block (`mExternalMfileRef`), and three external M-files called from the model (`externalmfile`, `externalmfile1`, `externalmfile2`).



Coverage by Model

	Complexity	Condition Coverage	Decision Coverage	MCDC Coverage
TOTAL COVERAGE		25%	85%	0%
1. . . . externalmfile	5	25%	88%	0%
2. . . . externalmfile1	5	25%	88%	0%
3. . . . mExternalMfile	5	--	50%	--
4. . . . mExternalMfileRef	3	--	100%	--
5. . . . externalmfile2	2	--	100%	--

The following models have signal range coverage:

mExternalMfile
mExternalMfileRef
externalmfile
externalmfile1
externalmfile2

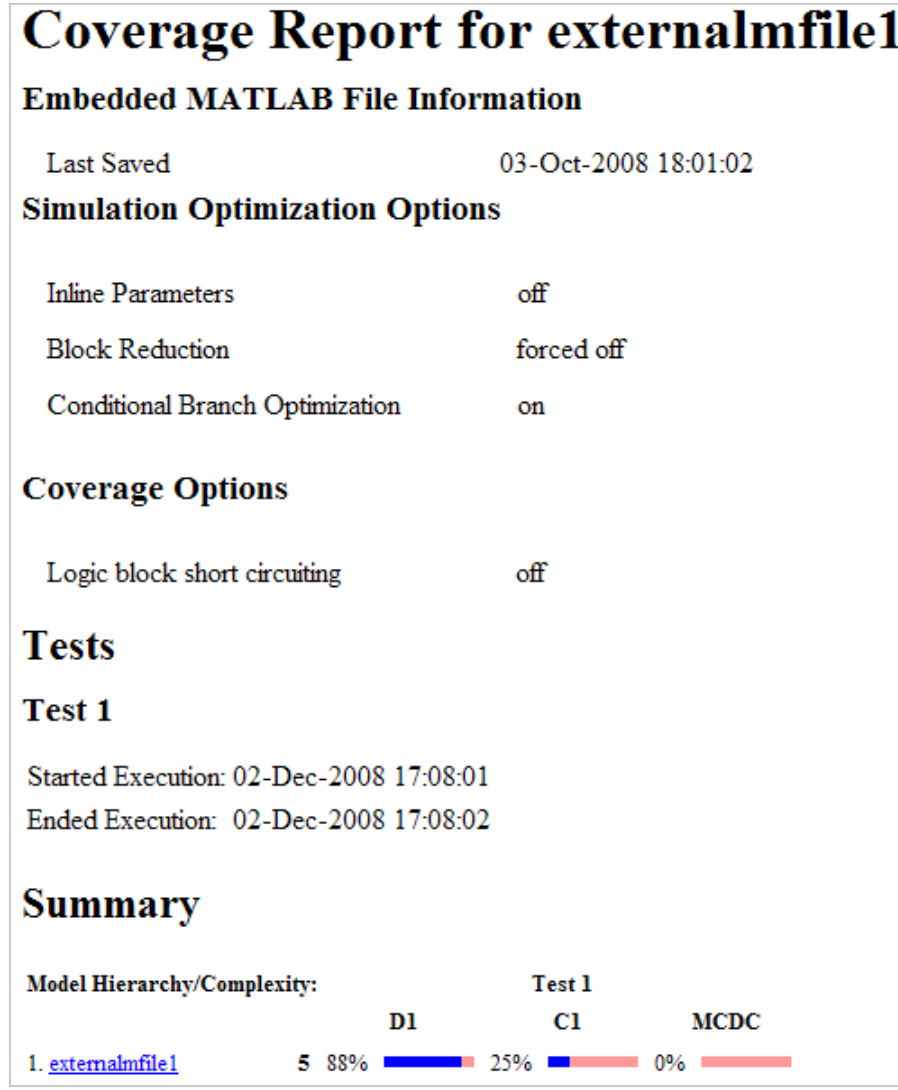
Model Reference Coverage Reports

If your top-level model references a model in a Model block, the software creates a separate report, named *reference_model_name*_cov.html, that includes coverage for the referenced model. This report has the same format as the "Model Coverage Reports" on page 5-27. Coverage results are recorded as if the referenced model was a stand-alone model; the report gives no indication that the model is referenced in a Model block.

External M-File Coverage Reports

If your top-level model calls any external M-files, the software creates a report, named *M-file_name*_cov.html, for each distinct M-file called from the model. If there are several calls to a given M-file from the model, the software creates only one report for that M-file, but it accumulates coverage from all the calls to it. The external M-file coverage report does not include information about what parts of the model call the external M-file.

The first section of the external M-file coverage report contains summary information similar to the model coverage report.



Coverage Report for externalmfile1

Embedded MATLAB File Information

Last Saved	03-Oct-2008 18:01:02

Simulation Optimization Options

Inline Parameters	off
Block Reduction	forced off
Conditional Branch Optimization	on

Coverage Options

Logic block short circuiting	off

Tests

Test 1

Started Execution: 02-Dec-2008 17:08:01
Ended Execution: 02-Dec-2008 17:08:02

Summary

Model Hierarchy/Complexity:		Test 1				
		D1		C1		MCDC
1. externalmfile1	5	88%		25%		0%

The **Details** section first reports coverage for the external M-file and the function in that M-file.

Details:

1. Embedded MATLAB file "externalmfile"

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	5
Decision (D1)	NA	88% (7/8) decision outcomes
Condition (C1)	NA	25% (1/4) condition outcomes
MCDC (C1)	NA	0% (0/6) conditions reversed the outcome

Embedded MATLAB function "externalmfile"

Parent: externalmfile

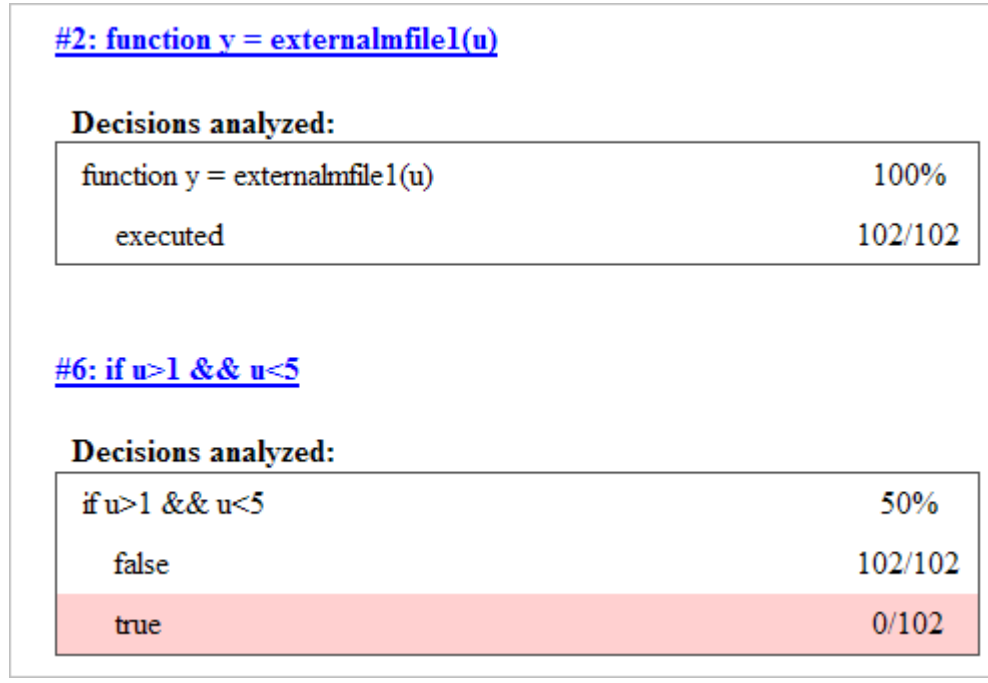
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	4
Decision (D1)	88% (7/8) decision outcomes
Condition (C1)	25% (1/4) condition outcomes
MCDC (C1)	0% (0/6) conditions reversed the outcome

The **Details** section also lists the content of the M-file, highlighting the code lines that have decision points or function definitions.

```
 1  %#eml
 2  function y = externalmfile1(u)
 3
 4  %   Copyright 2008 The MathWorks, Inc.
 5
 6  if u>1 && u<5
 7      a = 2;
 8  else
 9      a = 3;
10  end
11
12  for i=1:5
13      a = a+2;
14  end
15
16  y = a+localtest(a);
17
18  [x,y] = pol2cart(u,u);
19  [y2,y3] = cart2pol(x,y);
20
21  function y = localtest(u)
22
23  y = 0;
24  flg = true;
25  while flg
26      u = u/2;
27      y = y+1;
28      flg = u>2;
29  end
30
```

Coverage results for each of the highlighted code lines follow in the report. The following graphic shows a portion of these coverage results from the preceding code example.



#2: function y = externalmfile1(u)

Decisions analyzed:

function y = externalmfile1(u)	100%
executed	102/102

#6: if u>1 && u<5

Decisions analyzed:

if u>1 && u<5	50%
false	102/102
true	0/102

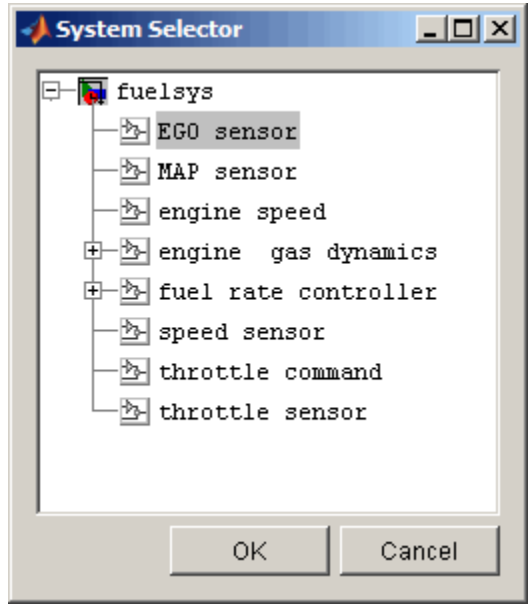
Subsystem Coverage Reports

In the Coverage Settings dialog box, when you select **Coverage for this model**, you can click **Select Subsystem** to request coverage for only the selected subsystem in the model. The software creates a model coverage report for the top-level model, but includes coverage results only for the subsystem.

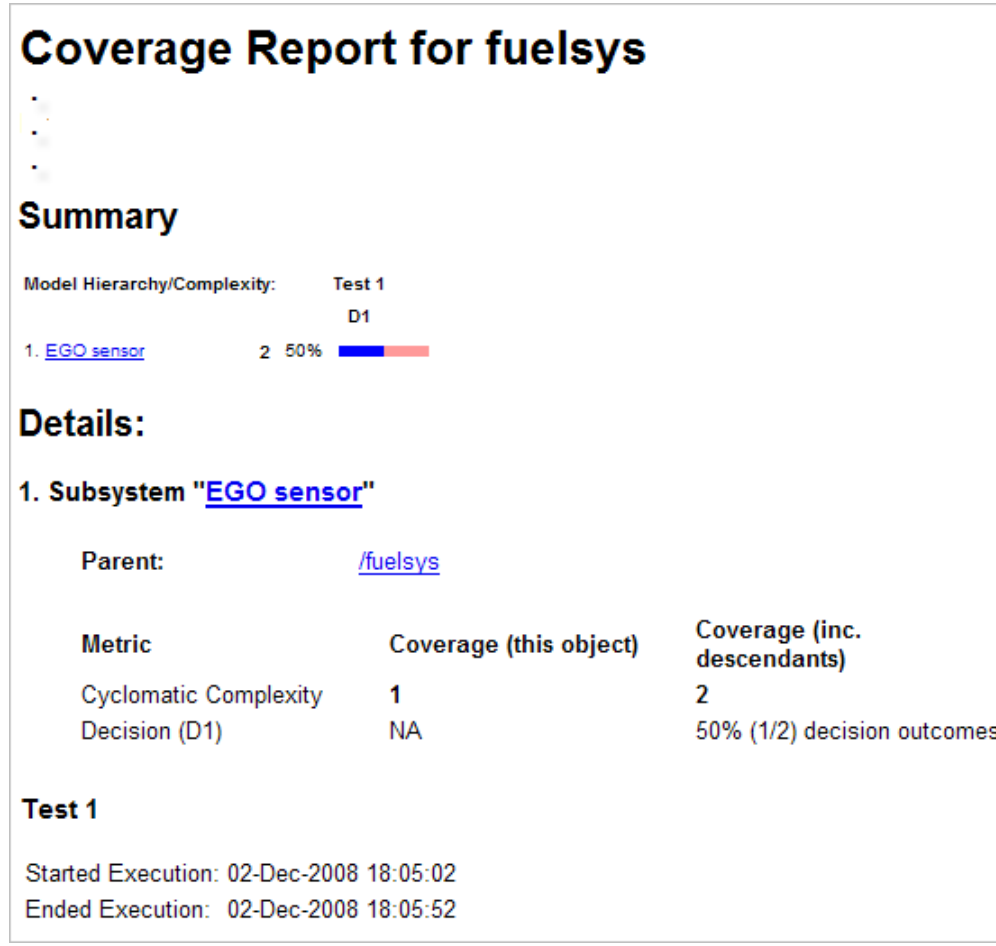
However, if the top-level model calls any external M-files and you selected **Coverage for External Embedded MATLAB files** in the Coverage Settings dialog box, the results include coverage for all external M-files called from:

- The subsystem for which you are recording coverage
- The top-level model that includes the subsystem

For example, in the `fuelsys` model, you click **Select Subsystem**, and select coverage for the EGO sensor subsystem.



The report is similar to the model coverage report, except that it includes only results for the EGO sensor subsystem (shown in the following graphic) and its contents.



Coverage Report for fuelsys

Summary

Model Hierarchy/Complexity:		Test 1 D1
1. EGO sensor	2	50%

Details:

1. Subsystem "EGO sensor"

Parent: /fuelsys

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	2
Decision (D1)	NA	50% (1/2) decision outcomes

Test 1

Started Execution: 02-Dec-2008 18:05:02
Ended Execution: 02-Dec-2008 18:05:52

Colored Simulink Diagram Coverage Display

In this section...

How Model Coverage Highlighting Works

The Simulink Verification and Validation software displays model coverage results for individual blocks directly in Simulink diagrams. If you enable model coverage, the tool does the following:

- Highlights (colors) blocks that have received model coverage during simulation
- Provides a context-sensitive display of summary model coverage information for each block

Coloring is used to highlight structural coverage in Simulink models. When you enable coloring for model coverage results (see "Enabling the Colored Diagram Display" on page 5-57), the tool highlights blocks that received the following types of model coverage:

- "Decision Coverage (DC)" on page 5-3
- "Condition Coverage (CC)" on page 5-3
- "Modified Condition/Decision Coverage (MC/DC)" on page 5-4

Enabling the Colored Diagram Display

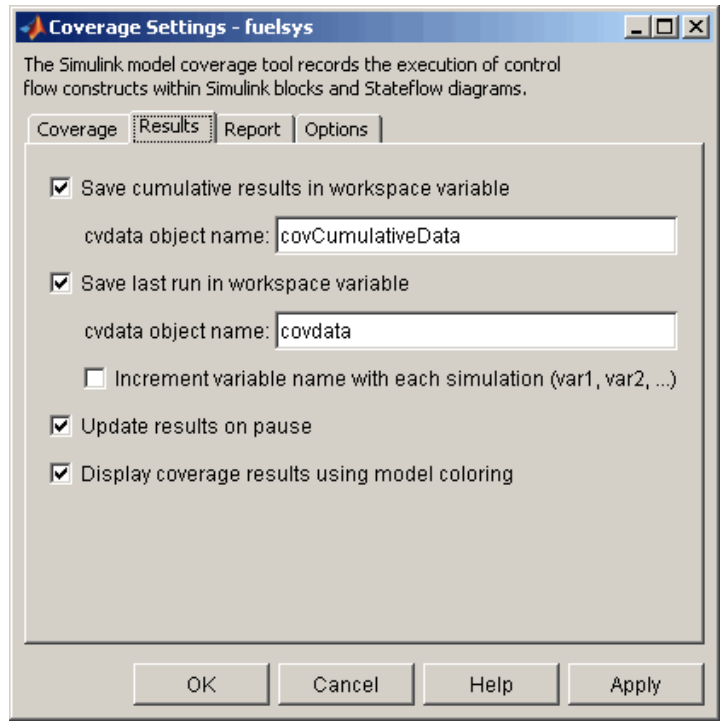
You enable the model coverage colored diagram display as follows:

1 In the Simulink window, from the **Tools** menu, select **Coverage Settings**.

The Coverage Settings dialog box appears.

2 In the **Coverage** tab of the Coverage Settings dialog box, select **Coverage for this model**.

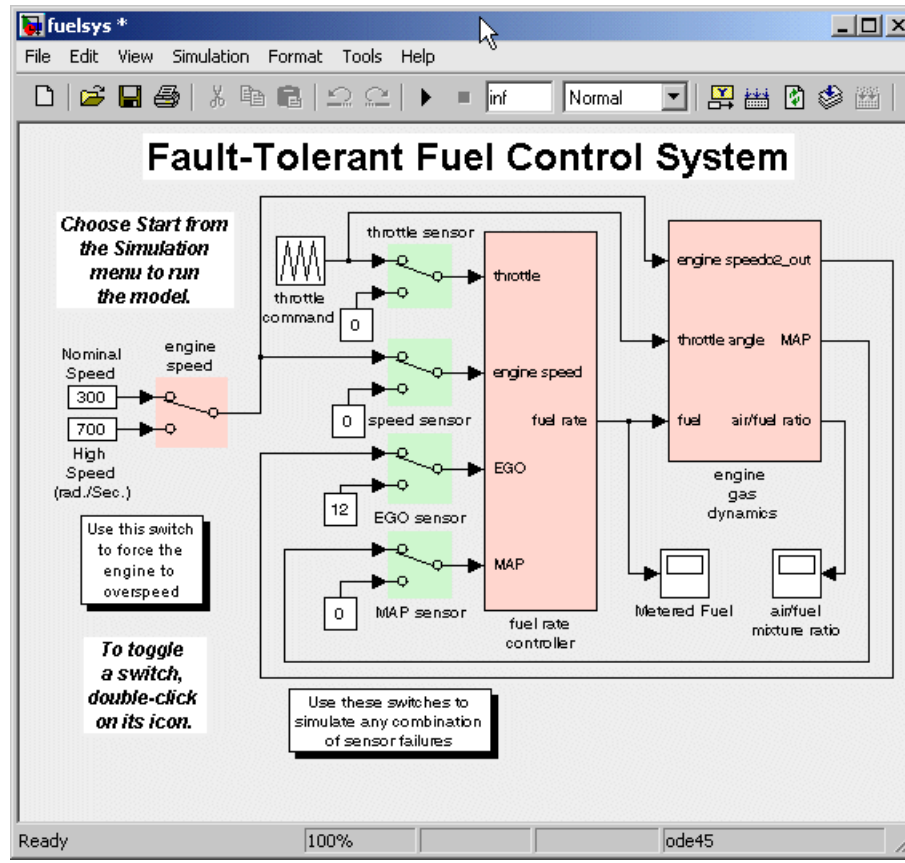
3 Select the **Results** tab, as shown.



The **Display coverage results using model coloring** option is selected by default for all models. This check box becomes visible only after **Coverage for this model** is enabled in the **Coverage** tab. You can disable this option for the current session by clearing this check box.

Displaying Model Coverage with Model Coloring

You enable display coverage as described in "Enabling the Colored Diagram Display" on page 5-57, After you enable this display, any time that the model generates a model coverage report, individual blocks receiving coverage are highlighted with light green or light red.

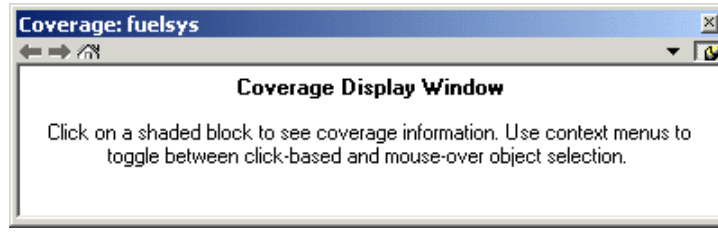


The light green Manual Switch blocks received full coverage during testing. The light red blocks (the `engine speed` Manual Switch block, and the `fuel rate controller` and `engine gas dynamics` subsystems) received incomplete coverage during testing. Blocks with no color highlighting (the Constant blocks, Scope blocks, and the `throttle command` Repeating Sequence block) received no coverage at all.

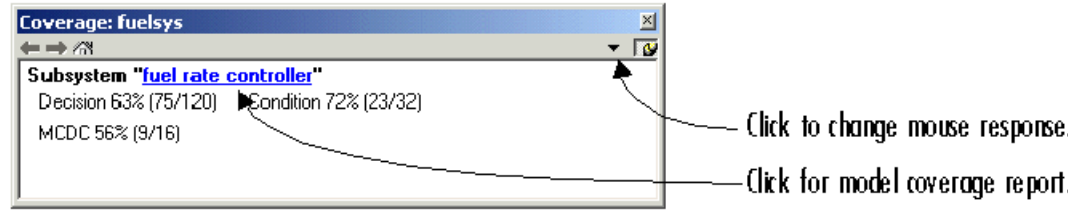
Note To restore the Simulink diagram to its original colors, right-click a colored block and select **Coverage** from the resulting context menu followed by **Remove information** from the resulting submenu. Alternatively, you can select **Remove Highlighting** from the Simulink **View** menu or the diagram's context menu to remove model coloring.

Accessing Coverage Information for Colored Blocks

"Displaying Model Coverage with Model Coloring" on page 5-58 describes the highlighted Simulink diagram that appears after simulation when you enable display coverage with model coloring in the coverage settings for the model. Along with the highlighted Simulink diagram, a Coverage Display window appears, as shown.



If you click a colored block in the Simulink model, its summarized coverage appears in the Coverage Display window. In the preceding example, the following summary report appears when you click the `fuel rate controller` subsystem:



Summary coverage information appears in the Coverage Display window for the block, whose hyperlinked name appears at the top of the window. Click the hyperlink to access the appropriate section of the coverage report for this

block. You can also see this section of the report by right-clicking the block and selecting **Coverage > Report**.

You can set the Coverage Display window to display coverage for a block in response to a hovering mouse cursor instead of a mouse click in one of two ways:

- Select the down arrow on the right side of the Coverage Display window, and, from the resulting menu, select **Focus**.
- Right-click a colored block and select **Coverage** from the resulting context menu followed by **Display details on mouse-over** from the resulting submenu.

Tip You can adjust the font size that the Coverage Display window uses. To increase the font size, press the **Ctrl+** keys; to decrease the font size, press the **Ctrl-** keys.

Using Model Coverage Commands

In this section...

About Model Coverage Commands

Using model coverage commands lets you automate the entire model coverage process with MATLAB scripts. You can use model coverage commands to set up model coverage tests, execute them in simulation, store the results, and report them. For a list of the model coverage commands that the Simulink Verification and Validation software provides, see Chapter 7, "Function Reference".

The sections that follow describe a workflow for using model coverage commands to create, run, store, and report model coverage tests.

Creating Tests with cvtest

The `cvtest` command creates a test specification object. Once you create the object, you simulate it with the `cvsim` command.

The call to `cvtest` has the following default syntax:

```
cvto = cvtest(root)
```

`root` is the name of, or a handle to, a Simulink model or a subsystem of a model. `cvto` is a handle to the resulting test specification object. Only the specified model or subsystem and its descendants are subject to model coverage testing.

The following command creates a test object with a specified label used for reporting results:

```
cvto = cvtest(root, label)
```

The following command creates a test with a setup command:

```
cvto = cvtest(root, label, setupcmd)
```

The setup command is executed in the base MATLAB workspace just prior to running the instrumented simulation. This command is useful for loading data prior to a test.

The returned `cvtest` object, `cvto`, has the following structure.

Field	Description
`id`	Read-only internal data-dictionary ID
`modelcov`	Read-only internal data-dictionary ID
`rootPath`	Name of the system or subsystem instrumented for analysis
`label`	String used when reporting results
`setupCmd`	Command executed in the base workspace just prior to simulation.
`settings.condition`	Set to 1 if condition coverage is desired
`settings.decision`	Set to 1 if decision coverage is desired
`settings.mcdc`	Set to 1 if MC/DC coverage is desired
`settings.sigrange`	Set to 1 if signal range coverage is desired
`settings.tableExec`	Set to 1 if lookup table coverage is desired

Field	Description
`modelRefSettings.enable`	String specifying one of the following values: • `Off` — Disables coverage for all referenced models • `all` — Enables coverage for all referenced models • `filtered` — Enables coverage only for referenced models not listed in the `excludedModels` subfield
`modelRefSettings.excludeTopModel`	Set to 1 if excluding coverage for the top model is desired
`modelRefSettings.excludedModels`	String specifying a comma-separated list of referenced models for which coverage is disabled when `modelRefSettings.enable` specifies `filtered`
`emlSettings.enableExternal`	Set to 1 to enable coverage for external M-files called by Embedded MATLAB functions in your model.

Running Tests with cvsim

Once you create a test specification object, you simulate it with the `cvsim` command.

Note You do not have to enable model coverage reporting for the model (see "Creating and Running Test Cases" on page 5-8) to use the `cvsim` command.

The call to `cvsim` has the following default syntax:

```
cvdo = cvsim(cvto)
```

This command executes the `cvtest` object `cvto` by starting a simulation run for the corresponding model. The results are returned in the `cvdata` object `cvdo`. But when recording coverage for multiple models in a hierarchy, `cvsim` returns its results in a `cv.cvdatagroup` object.

You can also control the simulation in a `cvsim` command by using parameters for the Simulink `sim` command, as shown in the following examples:

- The following command returns the simulation time vector `t`, matrix of state values `x`, and matrix of output values `y`.

  ```
  [cvdo,t,x,y] = cvsim(cvto)
  ```

- The following command overrides default simulation values with new values.

  ```
  [cvdo,t,x,y] = cvsim(cvto, timespan, options)
  ```

See documentation for the Simulink sim command for descriptions of the parameters `t`, `x`, `y`, `timespan`, and `options` in the previous examples.

You can execute multiple test objects with the `cvsim` command. The following command executes a set of coverage test objects, `cvto1, cvto2, ...` and returns the results in a set of `cvdata` objects, `cvdo1, cvdo2, ...`.

```
[cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)
```

You can also use the `cvsim` command to create and execute a `cvtest` object in one command as shown in the following example:

```
[cvdo,t,x,y] = cvsim(cvto, label, setupcmd)
```

Producing HTML Reports with cvhtml

Once you run a test in simulation with `cvsim`, you produce results that are saved to `cv.cvdatagroup` or `cvdata` objects in the base MATLAB workspace. Use the `cvhtml` command to produce an HTML report of these objects.

The following command creates an HTML report of the coverage results in the `cvdata` object `cvdo`, which is written to the file `file` in the current MATLAB directory:

```
cvhtml(file, cvdo)
```

The following example creates a combined report of several `cvdata` objects:

```
cvhtml(file, cvdo1, cvdo2, ...)
```

The results from each object are displayed in a separate column of the HTML report. Each `cvdata` object must correspond to the same root model or subsystem, or the function produces errors.

You can specify the detail level of the report with the value of `detail`, an integer between 0 and 3, as shown in the following example:

```
cvhtml(file, cvdo1, cvdo2,..., detail)
```

Greater numbers for `detail` indicate greater detail. The default value is 2.

Saving Test Runs to a File with cvsave

Once you run a test with `cvsim`, save its coverage tests and results to a file with the function `cvsave`:

```
cvsave(filename, model)
```

Save all the tests and results related to `model` in the text file `filename.cvt`:

```
cvsave(filename, cvto1, cvto2, ...)
```

Save the specified tests in the text file `filename.cvt`. Information about the referenced models is also saved.

You can also save specified `cvdata` objects to file. The following example saves the tests, test results, and referenced models' structure in `cvdata` objects to the text file `filename.cvt`:

```
cvsave(filename, cvdo1, cvdo2, ...)
```

Loading Stored Coverage Test Results with cvload

The `cvload` command loads into memory the coverage tests and results stored in a file by the `cvsave` command. The following example loads the tests and data stored in the text file `filename.cvt`:

```
[cvtos, cvdos] = cvload(filename)
```

The `cvtest` objects that are successfully loaded are returned in `cvtos`, a cell array of `cvtest` objects. The `cvdata` objects that are successfully loaded are returned in `cvdos`, a cell array of `cvdata` objects. `cvdos` has the same size as `cvtos`, but can contain empty elements if a particular test has no results.

In the following example, if `restoretotal` is 1, the cumulative results from prior runs are restored:

```
[cvtos, cvdos] = cvload(filename, restoretotal)
```

If `restoretotal` is unspecified or 0, the model's cumulative results are cleared.

cvload Special Considerations

The following are some special considerations for using the `cvload` command:

- If a model with the same name exists in the coverage database, only the compatible results are loaded from the file and they reference the existing model to prevent duplication.
- If the Simulink models referenced in the file are open but do not exist in the coverage database, the coverage tool resolves the links to the models that are already open.
- When you are loading several files that reference the same model, only the results that are consistent with the earlier files are loaded.

Coverage Script Example

The following example is a portion of `simcovdemo2.m`, located in the coverage root folder. This example demonstrates common model coverage commands.

```
mdl = 'slvnvdemo_ratelim_harness';

testObj1 = cvtest([mdl, '/Adjustable Rate Limiter']);
testObj1.label = 'Gain within slew limits';
testObj1.setupCmd = 'load(''within_lim.mat'');';
testObj1.settings.mcdc = 1;
```

```
testObj2 = cvtest([mdl, '/Adjustable Rate Limiter']);
testObj2.label='Rising gain that temporarily exceeds slew limit';
testObj2.setupCmd = 'load(''rising_gain.mat'');';
testObj2.settings.mcdc = 1;

[dataObj1,T,X,Y] = cvsim(testObj1,[0 2]);
[dataObj2,T,X,Y] = cvsim(testObj2,[0 2]);

cvhtml('ratelim_report',dataObj1,dataObj2);
cumulative = dataObj1+dataObj2;
cvsave('ratelim_testdata',cumulative);
```

In this example, you create two `cvtest` objects, `testObj1` and `testObj2`, and simulate them according to their specifications. Each `cvtest` object uses the `setupCmd` property to load a data file before simulation. Decision coverage is enabled by default, and MC/DC coverage is enabled as well. After simulation, you use `cvhtml` to display the coverage results for two tests and the cumulative coverage. Lastly, you compute cumulative coverage with the `+` operator and save the results. For another detailed example of how to use the model coverage commands, enter `simcovdemo` at the MATLAB command prompt.

Using Model Coverage Commands for Referenced Models

In this section...

Introduction

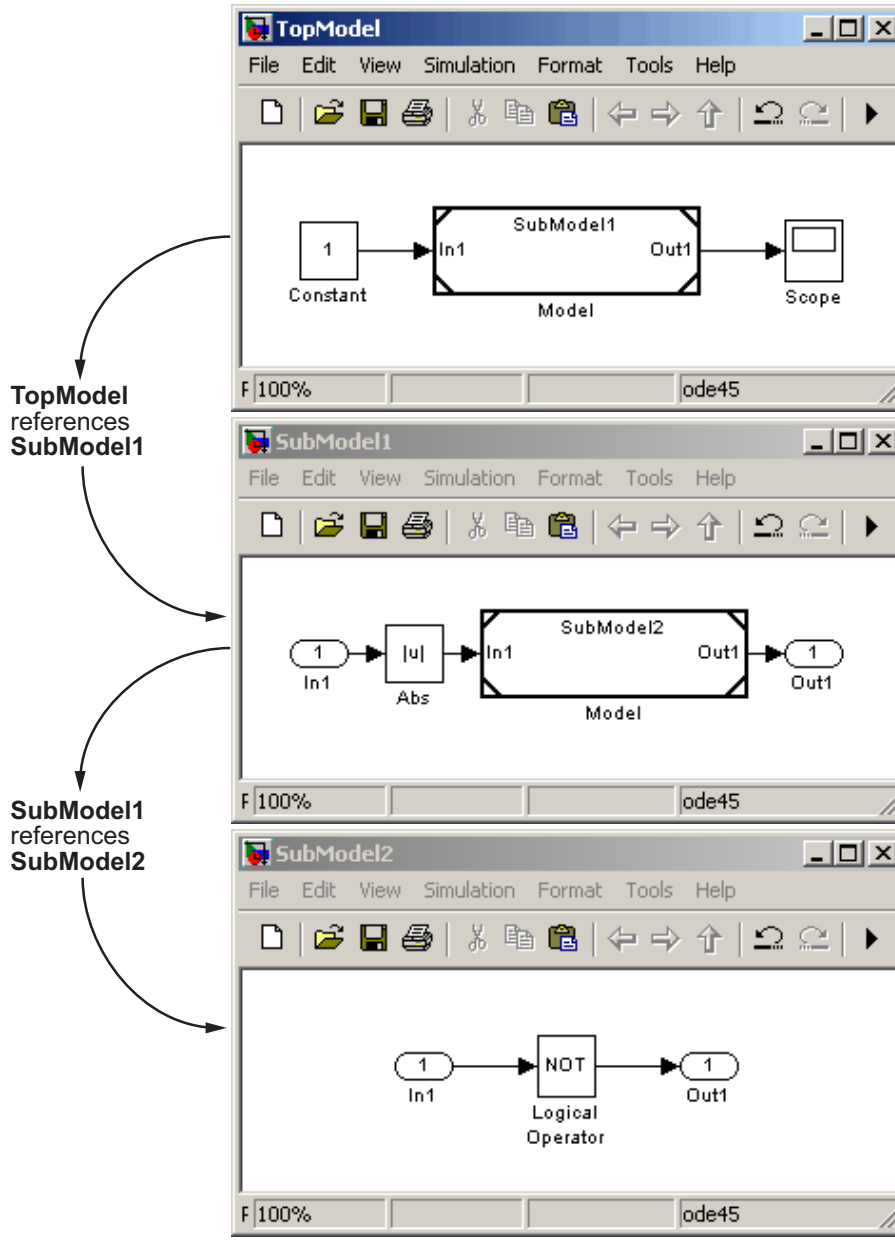
The Simulink software allows you to include one model in another by using Model blocks. Each Model block represents a reference to another model, called a *referenced model* or *submodel*. A referenced model itself can contain Model blocks that reference other models. You can construct a hierarchy of referenced models, in which the topmost model is called the *top model*. See "Referencing a Model" in *Simulink User's Guide* for more information.

Model coverage supports referenced models that operate in Normal mode. That is, you can record coverage only for those Model blocks whose **Simulation mode** parameter specifies `Normal`. You can use model coverage commands to record coverage for referenced models (see "Using Model Coverage Commands" on page 5-62). However, if you want to record different types of coverage for models in a hierarchy, you must use the `cvsimref` function. The following steps describe a basic workflow for using this function to obtain model coverage results for Model blocks:

Step	Description	See...
1	Use `cv.cvtestgroup` to group together test specification objects that correspond to each model in a hierarchy.	"Creating a Test Group with cv.cvtestgroup" on page 5-72

Step	Description	See...
2	Use `cvsimref` to simulate the top model in a hierarchy and record coverage results for its referenced models.	"Running Tests with cvsimref" on page 5-72
3	Use `cv.cvdatagroup` to extract the coverage data objects that correspond to each model in a hierarchy.	"Extracting Results from cv.cvdatagroup" on page 5-73

The next sections illustrate how to complete each of these steps using the following model hierarchy:


TopModel
File Edit View Simulation Format Tools Help
1
Constant
SubModel1
In1
Out1
Model
Scope
100%
ode45
TopModel
references
SubModel1
SubModel1
File Edit View Simulation Format Tools Help
1
In1
|u|
Abs
SubModel2
In1
Out1
Model
1
Out1
100%
ode45
SubModel1
references
SubModel2
SubModel2
File Edit View Simulation Format Tools Help
1
In1
NOT
Logical
Operator
1
Out1
100%
ode45

Creating a Test Group with cv.cvtestgroup

The `cvtest` command creates a test specification object for a Simulink model (see "Creating Tests with cvtest" on page 5-62). But if your model references other models, you might use a different test specification object for each model in the hierarchy. In this case, the `cv.cvtestgroup` object allows you to group together multiple test specification objects. After you create a group of test specification objects, you simulate it using the `cvsimref` function.

For example, suppose that you create a different test specification object for each of the models in your hierarchy:

```
cvto1 = cvtest('TopModel')
cvto2 = cvtest('SubModel1')
cvto3 = cvtest('SubModel2')
```

The following command creates a test group object named `cvtg`, which contains all the `cvtest` objects associated with your model hierarchy:

```
cvtg = cv.cvtestgroup(cvto1, cvto2, cvto3)
```

A `cv.cvtestgroup` object provides methods, such as `add` and `get`, which allow you to customize its contents to meet your needs. For more information, see the documentation for the `cv.cvtestgroup` function.

Running Tests with cvsimref

Once you create a test group object, you simulate it with the `cvsimref` function.

Note You must use the `cvsimref` function to record coverage for referenced models in a hierarchy.

The call to `cvsimref` has the following default syntax:

```
cvdg = cvsimref(topModelName, cvtg)
```

This command executes the test group object `cvtg` by simulating the top model in the corresponding model hierarchy, `topModelName`. It returns the coverage results in a `cv.cvdatagroup` object named `cvdg`.

Like the `cvsim` function, you can use parameters from the Simulink `sim` function in a `cvsimref` command to control the simulation, as shown in the following examples:

- The following command returns the simulation time vector `t`, matrix of state values `x`, and matrix of output values `y`:

  ```
  [cvdg,t,x,y] = cvsimref(topModelName, cvtg)
  ```

- The following command overrides default simulation values with new values:

  ```
  [cvdg,t,x,y] = cvsimref(topModelName, cvtg, timespan, options)
  ```

For descriptions of the parameters `t`, `x`, `y`, `timespan`, and `options`, see the documentation for the `sim` function in the *Simulink Reference*.

Extracting Results from cv.cvdatagroup

Once you simulate a test group with `cvsimref`, the function returns results that reside in a `cv.cvdatagroup` object. The data group object contains multiple `cvdata` objects, each of which corresponds to coverage results for a particular model in the hierarchy.

A `cv.cvdatagroup` object provides methods, such as `allNames` and `get`, which allow you to extract individual `cvdata` objects. For example, enter the following command to obtain a cell array that lists all model names associated with the data group `cvdg`:

```
modelNames = cvdg.allNames
```

To extract the `cvdata` objects that correspond to the particular models, enter

```
cvdo1 = cvdg.get('TopModel')
cvdo2 = cvdg.get('SubModel1')
cvdo3 = cvdg.get('SubModel2')
```

After you extract the individual `cvdata` objects, you can use other model coverage commands to operate on the coverage data of a particular model. For example, you can use the `cvhtml` function to create and display an HTML report of the coverage results (see “Producing HTML Reports with cvhtml” on page 5-65).

Model Coverage for Embedded MATLAB Function Blocks

In this section...

Types of Model Coverage in Embedded MATLAB Function Blocks

This section describes the model coverage that an Embedded MATLAB Function block receives.

Note Model coverage is available to you only if you have a Simulink Verification and Validation software license.

During simulation, the following Embedded MATLAB Function block function statements are tested for decision coverage:

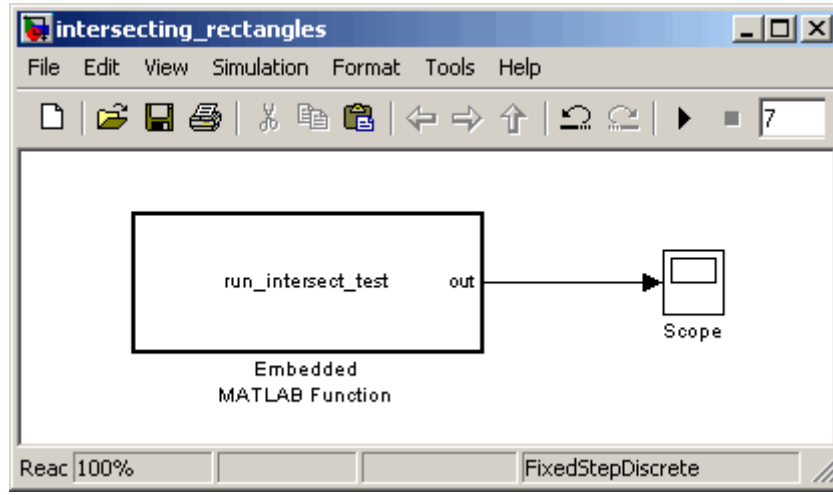
- Function header — Decision coverage is 100% if the function or subfunction is executed.
- `if` — Decision coverage is 100% if the `if` expression evaluates to true at least once, and false at least once.
- `switch` — Decision coverage is 100% if every `switch` case is taken, including the fall-through case.
- `for` — Decision coverage is 100% if the equivalent loop condition evaluates to true at least once, and false at least once.
- `while` — Decision coverage is 100% if the equivalent loop condition evaluates to true at least once, and false at least once.

During simulation, the following logical conditions are tested for condition coverage and MCDC coverage in the Embedded MATLAB Function block function:

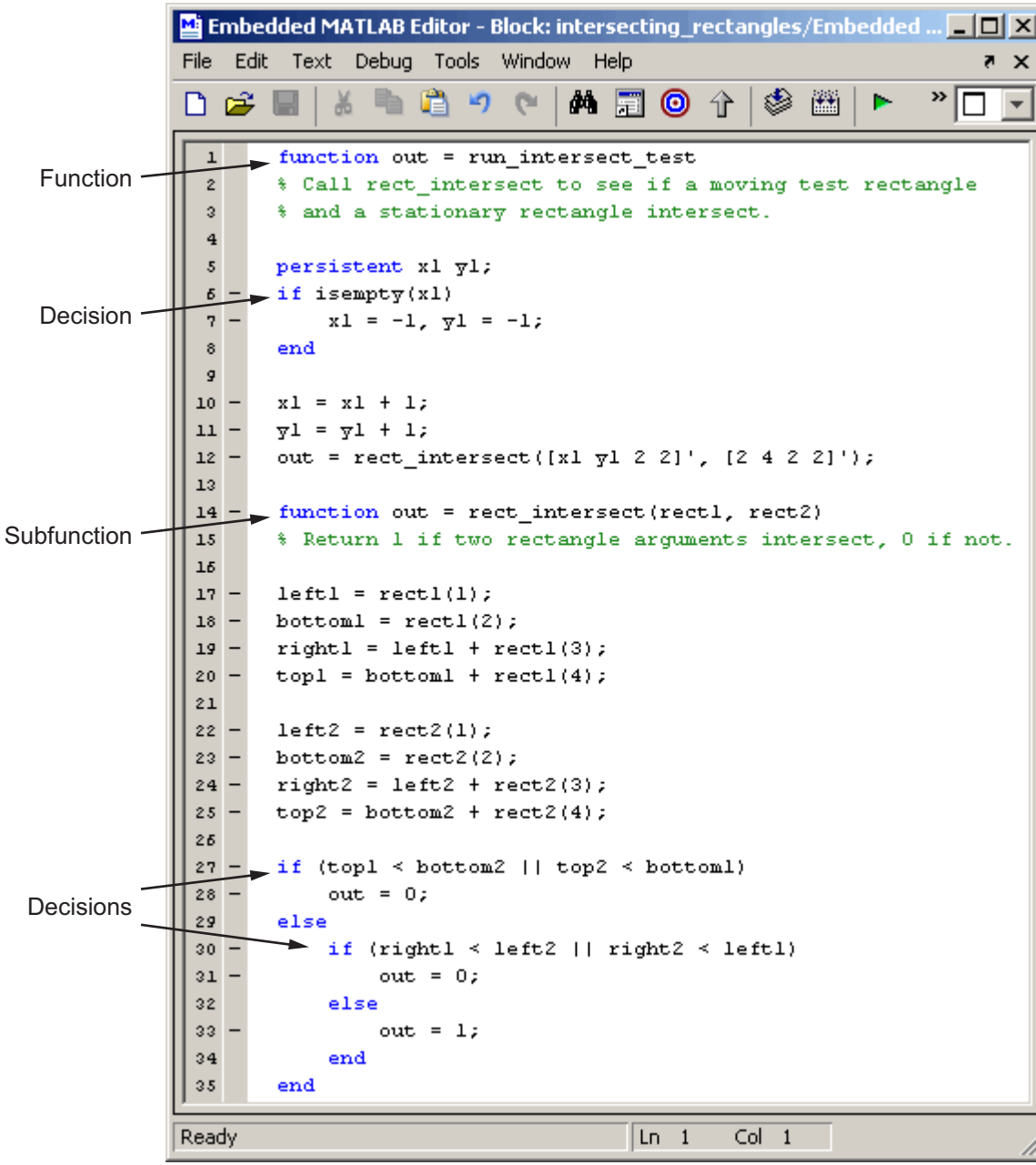
- `if` statement conditions
- `while` statement conditions, if present

Creating a Model with Embedded MATLAB Function Block Decisions

In this topic you use an example model to examine model coverage of an Embedded MATLAB Function block. The following model contains a single Embedded MATLAB Function block with output data sent to a Scope block.

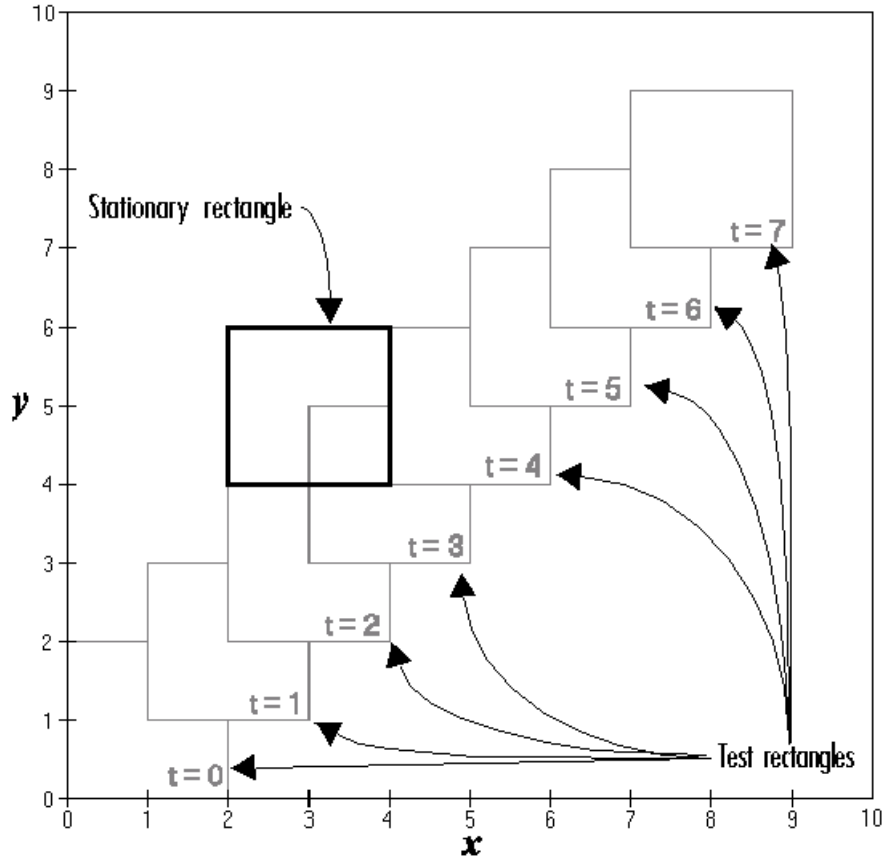


Double-click the Embedded MATLAB Function block to specify its program content as shown.


Function
Decision
Subfunction
Decisions
Embedded MATLAB Editor - Block: intersecting_rectangles/Embedded ...
File Edit Text Debug Tools Window Help
1 function out = run_intersect_test
2 % Call rect_intersect to see if a moving test rectangle
3 % and a stationary rectangle intersect.
4
5 persistent xl yl;
6 if isempty(xl)
7 xl = -1, yl = -1;
8 end
9
10 xl = xl + 1;
11 yl = yl + 1;
12 out = rect_intersect([xl yl 2 2]', [2 4 2 2]');
13
14 function out = rect_intersect(rect1, rect2)
15 % Return 1 if two rectangle arguments intersect, 0 if not.
16
17 left1 = rect1(1);
18 bottom1 = rect1(2);
19 right1 = left1 + rect1(3);
20 top1 = bottom1 + rect1(4);
21
22 left2 = rect2(1);
23 bottom2 = rect2(2);
24 right2 = left2 + rect2(3);
25 top2 = bottom2 + rect2(4);
26
27 if (top1 < bottom2 || top2 < bottom1)
28 out = 0;
29 else
30 if (right1 < left2 || right2 < left1)
31 out = 0;
32 else
33 out = 1;
34 end
35 end
Ready
Ln 1 Col 1

The `run_intersect_test` Embedded MATLAB Function block contains two functions. The top-level function, `run_intersect_test`, sends the coordinates for two rectangles, one fixed and the other moving, as arguments to the subfunction `rect_intersect`, which tests for intersection between the two. The origin of the moving rectangle increases by 1 in the `x` and `y` directions with each time step.

The coordinates for the origin of the moving test rectangle are represented by persistent data `x1` and `y1`, which are both initialized to -1. For the first sample, `x1` and `y1` are both incremented to 0. From then on, the progression of rectangle arguments during simulation is as follows:

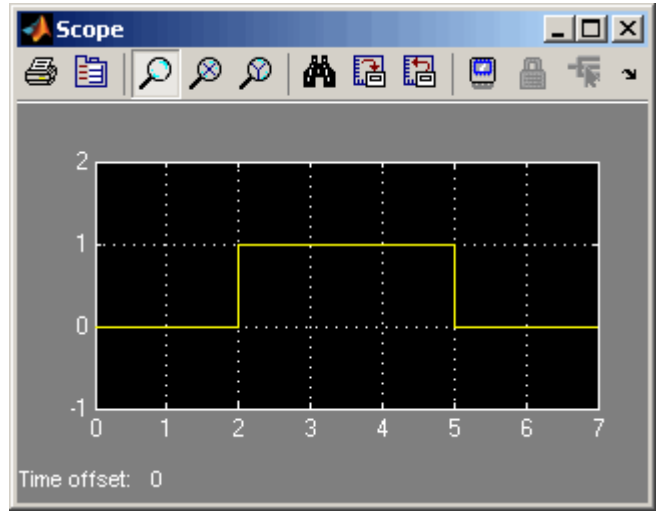


The fixed rectangle is shown in bold with a lower left origin of `(2,4)` and a width and height of 2. At time `t = 0`, the first test rectangle has an origin of `(0,0)` and a width and height of 2. For each succeeding sample, the origin of the test rectangle is incremented by `(1,1)`. The rectangles at sample times `t = 2`, `3`, and `4` intersect with the test rectangle.

The subfunction `rect_intersect` checks to see if its two rectangle arguments intersect. Each argument consists of coordinates for the lower left corner of the rectangle (origin), and its width and height. `x` values for the left and right sides and `y` values for the top and bottom are calculated for each rectangle and

compared in nested `if-else` decisions. The function returns a logical value of 1 if the rectangles intersect and 0 if they do not.

Scope output during simulation, which plots the return value against the sample time, confirms the intersecting rectangles for sample 2, 3, and 4 as shown.



Understanding Embedded MATLAB Function Block Model Coverage

Model coverage reports are generated automatically after a simulation if you specify them. See “Creating and Running Test Cases” on page 5-8 for instructions on how to specify a model coverage report.

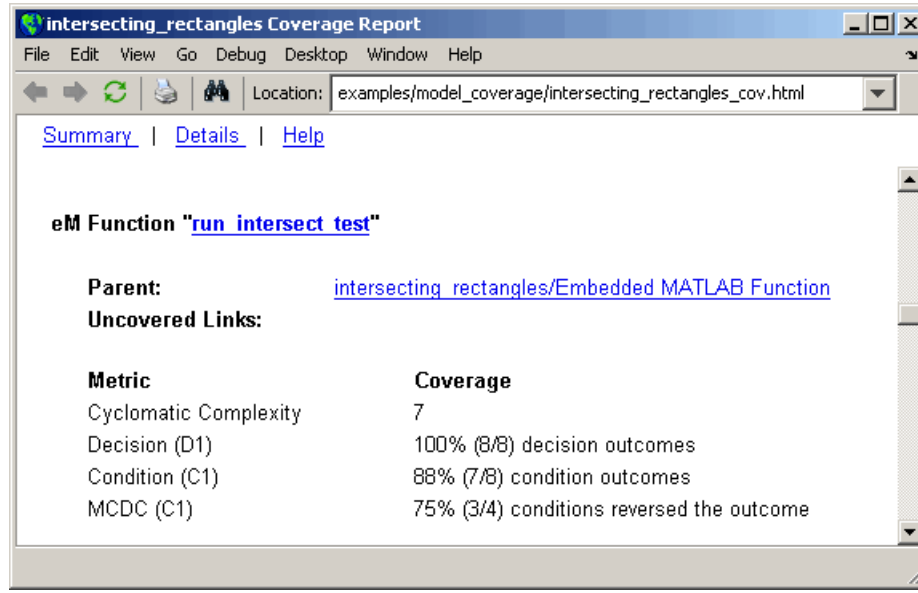
When simulation is finished, the model coverage report appears in a browser window. After the summary for the model, the Details section of the model coverage report reports on each of the parts of the model. Model coverage for the parts of the example model in “Creating a Model with Embedded MATLAB Function Block Decisions” on page 5-75 appears in the following model-block-function order.

Model:	`intersecting_rectangles`
Block:	`Embedded MATLAB Function`
Function:	`run_intersect_test`
Decision Lines:	`1:  function out = rect_intersect_test`
	`6:  if isempty(x1)`
	`14:  function out = rect_intersect(rect1, rect2)`
	`27:  if (top1 < bottom2 || top2 < bottom1)`
	`30:  if (right1 < left2 || right2 < left1)`

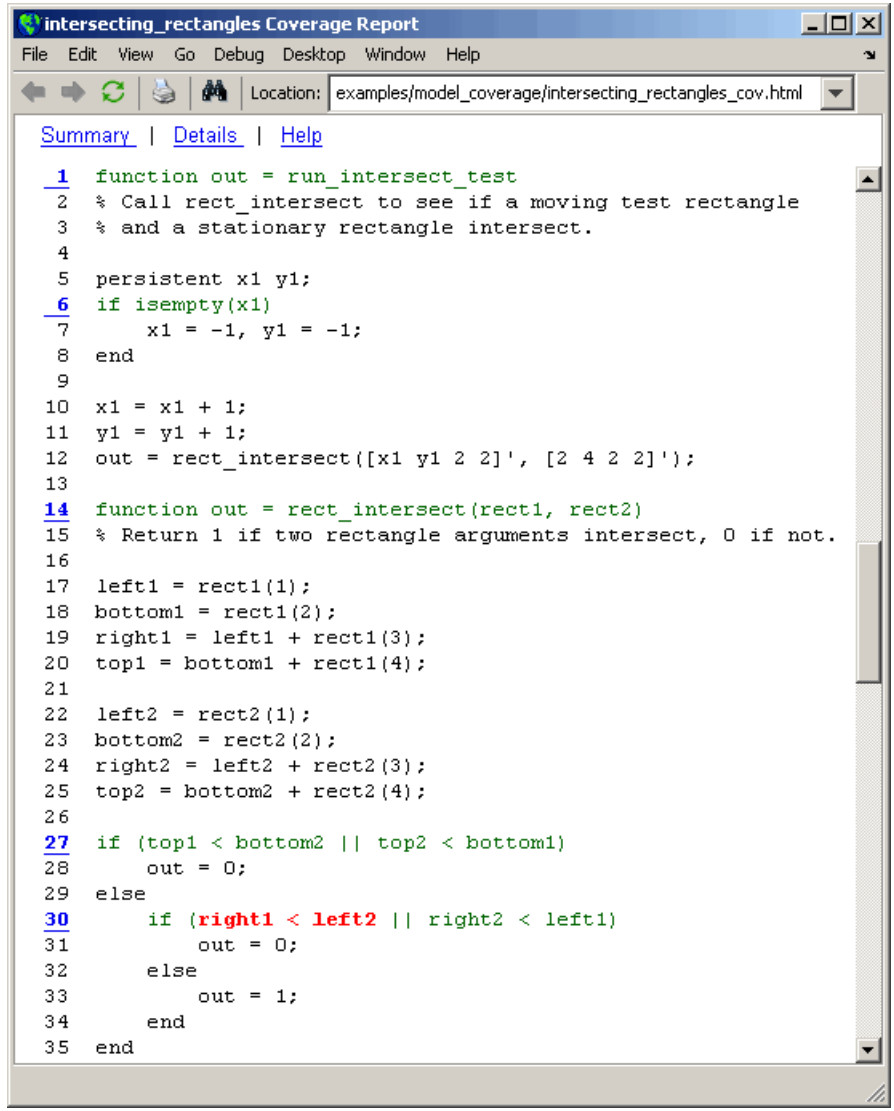
The following subtopics examine the model coverage report for the example model in reverse function-block-model order. Reversing the order helps you make sense of the summary information that appears at the top of each section.

Model Coverage for the Embedded MATLAB Function Block Function run_intersect_test

Model coverage for the Embedded MATLAB Function block function `run_intersect_test` is reported under the linked name of the function. Clicking this link opens the function in the Embedded MATLAB Editor. Following the linked function name is a link to the model coverage report for the parent Embedded MATLAB Function block of `run_intersect_test`.



The top half of the report for the function summarizes its model coverage results as shown. The coverage metrics for `run_intersect_test` include decision, condition, and MCDC coverage. These metrics are best understood by examining the code listing for `run_intersect_test` that follows.



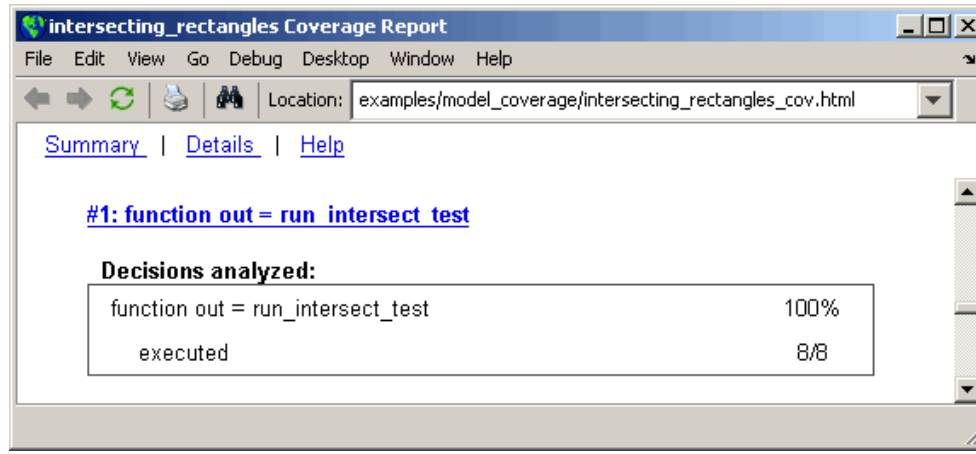
Lines with coverage elements are marked by a highlighted line number in the listing. Line 1 receives decision coverage on whether the top-level function `run_intersect_test` is executed. Line 6 receives decision coverage for its `if` statement. Line 14 receives decision coverage on whether the subfunction

`rect_intersect` is executed. Lines 27 and 30 receive decision, condition, and MCDC coverage for their `if` statements and conditions. Each of these lines is the subject of a report that follows the listing.

Notice that the condition `right1 < left2` in line 30 is highlighted in red. This means that this condition was not tested for all of its possible outcomes during simulation. Exactly which of the outcomes was not tested is answered by the report for the decision in line 30.

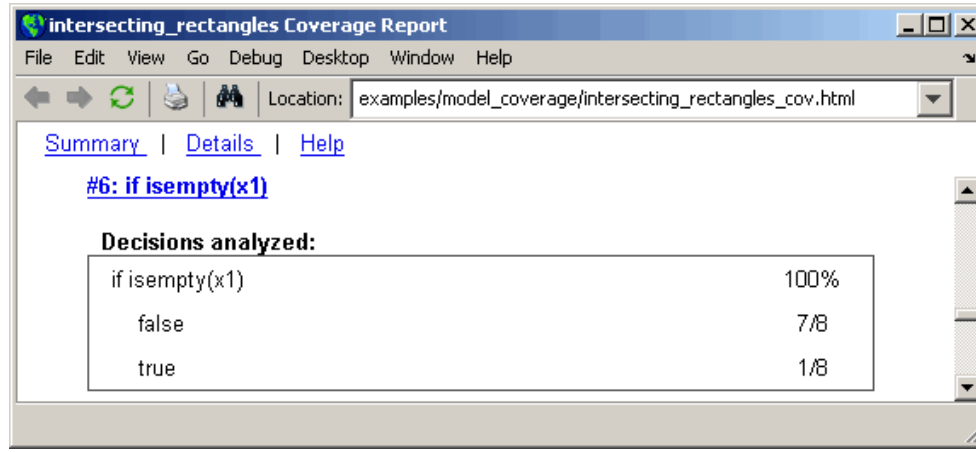
The following subtopics display the coverage for each decision line of `run_intersect_test`. The coverage for each line is titled with the line itself, which is linked to display the function with the line highlighted.

Coverage for Line 1. The coverage metrics for line 1 appear below the listing for the function `run_intersect_test`.



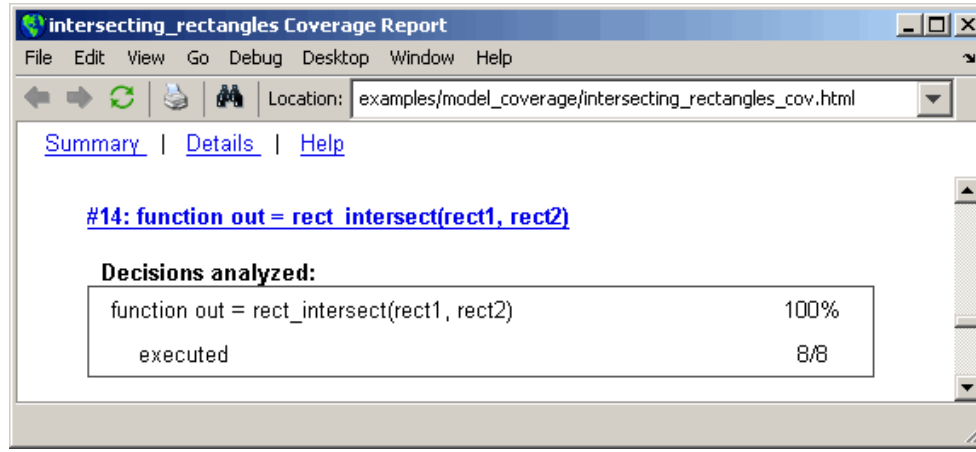
The first line of every function receives coverage analysis indicative of the decision to run the function in response to a call. Coverage for `run_intersect_test` indicates that it executed during testing.

Coverage for Line 6. The coverage metrics for line 6 appear below the coverage metrics for line 1.



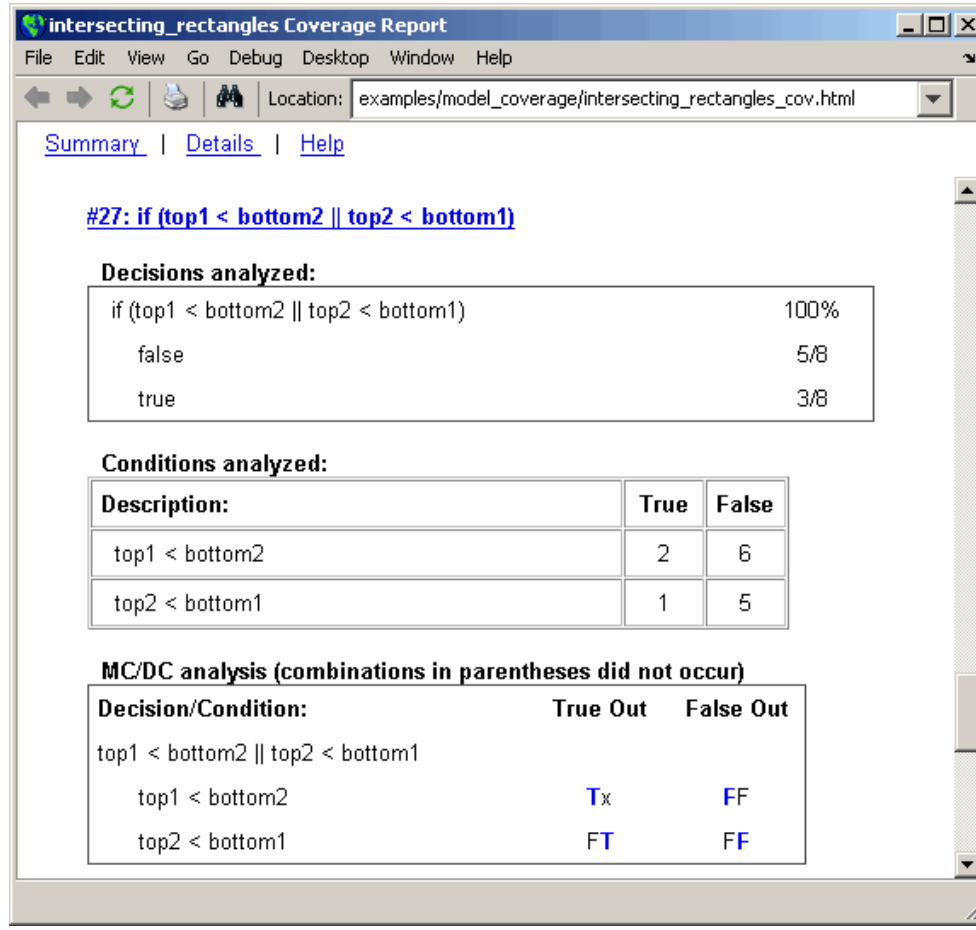
The **Decisions analyzed** table indicates that the decision in line 6, `if isempty(x1)`, executed a total of eight times. The first time it executed, the decision evaluated to true, enabling `run_intersect_test` to initialize the values of its persistent data. The remaining seven times the decision executed, it evaluated to false. Because both possible outcomes occurred, decision coverage is 100%.

Coverage for Line 14. The coverage metrics for line 14 appear below the coverage metrics for line 6.



This table indicates that the subfunction `rect_intersect` executed during testing.

Coverage for Line 27. Coverage metrics for line 27 appear below the coverage metrics for line 14.



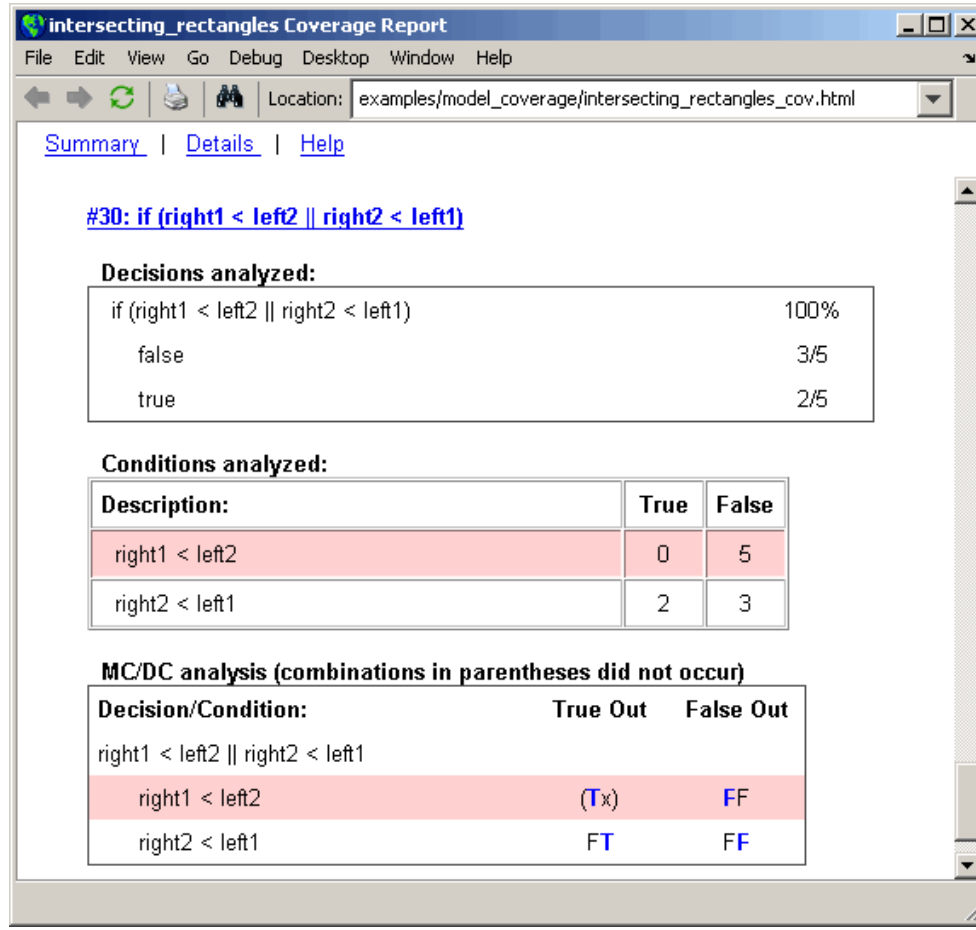
The **Decisions analyzed** table indicates that there are two possible outcomes for the decision in line 27: true and false. Five of the eight times it was executed, the decision evaluated to false, and the remaining three times, it

evaluated to true. Because both possible outcomes occurred, decision coverage is 100%.

The **Conditions analyzed** table sheds some additional light on the decision in line 27. Because this decision consists of two conditions linked by a logical OR (||) operation, only one condition must evaluate true for the decision to be true. If the first condition evaluates to true, there is no need to evaluate the second condition. The first condition, `top1 < bottom2`, was evaluated eight times, and was true twice. This means that it was necessary to evaluate the second condition only six times. In only one case was it true, which brings the total true occurrences for the decision to three, as reported in the **Decisions analyzed** table.

MCDC coverage looks for decision reversals that occur because one condition outcome changes from `T` to `F` or from `F` to `T`. The **MC/DC analysis** table identifies all possible combinations of outcomes for the conditions that lead to a reversal in the decision. The character `x` is used to indicate a condition outcome that is irrelevant to the decision reversal. Decision-reversing condition outcomes that are not achieved during simulation are marked with a set of parentheses. There are no parentheses, therefore all decision-reversing outcomes occurred and MCDC coverage is complete for the decision in line 27.

Coverage for Line 30. Coverage metrics for line 30 appear below the coverage metrics for line 27.



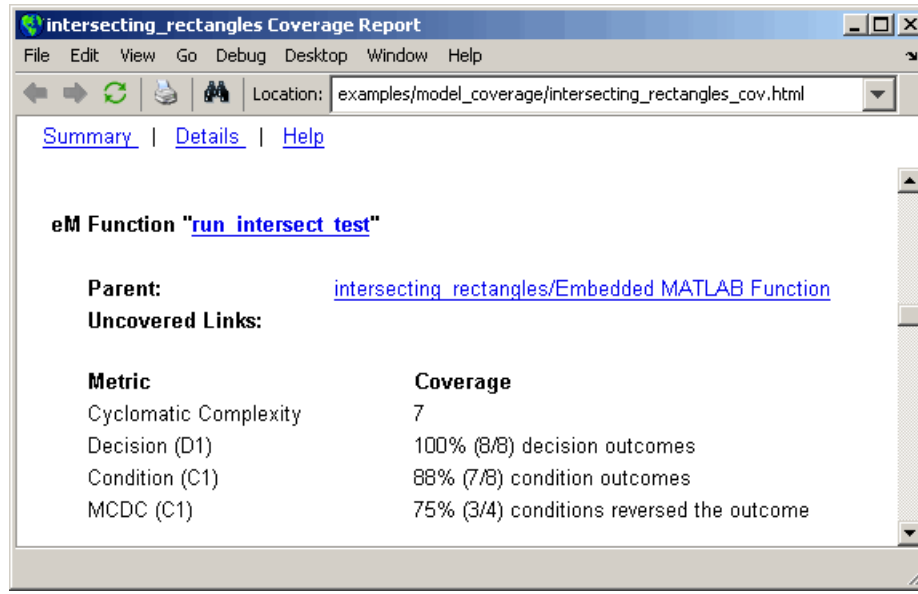
The line 30 decision, `if (right1 < left2 || right2 < left1)`, is nested in the `if` statement of the line 27 decision and is evaluated only if the line 27 decision is false. Because the line 27 decision evaluated false five times, line 30 is evaluated five times, three of which were false. Because both the true and false outcomes were achieved, decision coverage for line 30 is 100%.

Because line 30, like line 27, has two conditions related by a logical OR operator (||), condition 2 is tested only if condition 1 is false. Because condition 1 tests false five times, condition 2 is tested five times. Of these,

condition 2 tests true two times and false three times, which accounts for the two occurrences of the true outcome for this decision.

Because the first condition of the line 30 decision does not test true, both outcomes did not occur for that condition and the condition coverage for the first condition is highlighted with a rose color. MCDC coverage is also highlighted in the same way for a decision reversal based on the true outcome for that condition.

Coverage for run_intersect_test. The metrics that summarize coverage for the entire `run_intersect_test` function are reported prior to its listing and are repeated here as shown.



The results summarized in the coverage metrics summary can be expressed in the following conclusions:

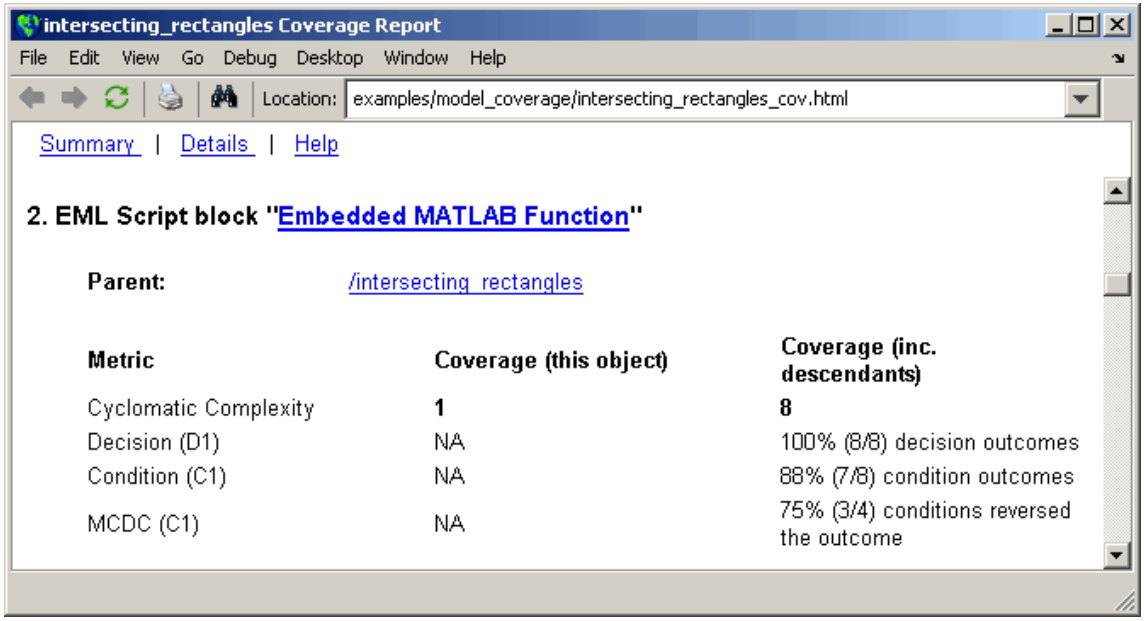
- There are eight decision outcomes reported for `run_intersect_test` in the line reports: one for line 1 (executed), two for line 6 (true and false), one for line 14 (executed), two for line 27 (true and false), and two for line 30 (true and false). The decision coverage for each line shows 100% decision

coverage. This means that decision coverage for `run_intersect_test` is eight of eight possible outcomes, or 100%.

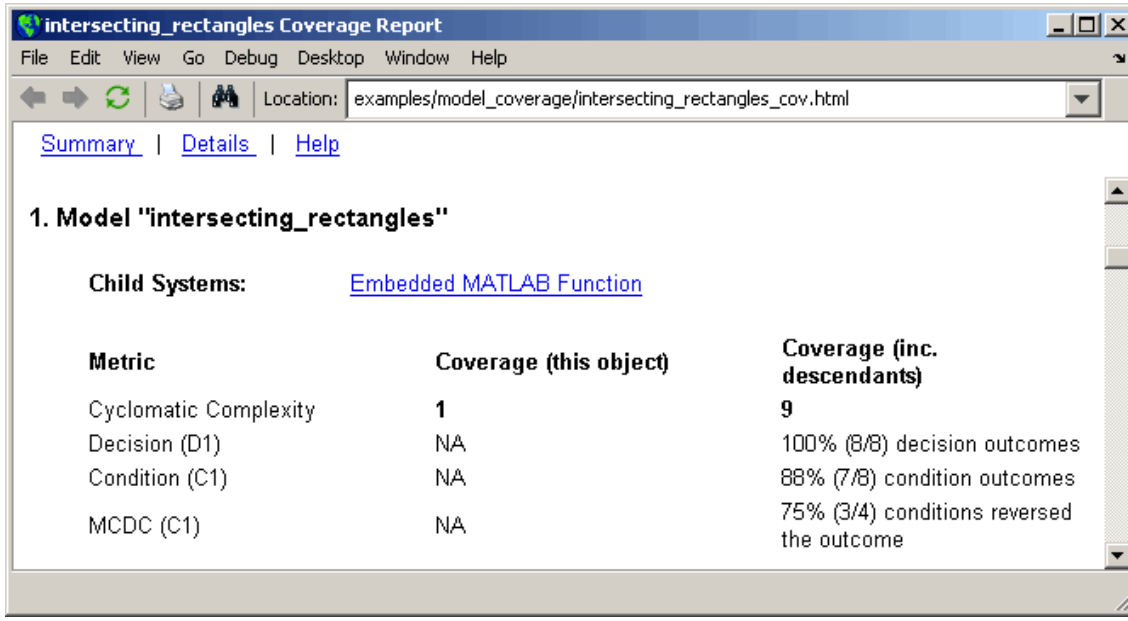
- There are four conditions reported for `run_intersect_test` in the line reports. Lines 27 and 30 each have two conditions, and each condition has two condition outcomes (true and false), for a total of eight condition outcomes in `run_intersect_test`. All conditions tested positive for both the true and false outcome except for the first condition of line 30 (`right1 < left2`). This means that condition coverage for `run_intersect_test` is seven of eight, or 88%.
- The MCDC coverage tables for decision lines 27 and 30 each list two cases of decision reversal for each condition, for a total of four possible reversals. Only the decision reversal for a change in the evaluation of the condition `right1 < left2` of line 30 from true to false did not occur during simulation. This means that three of four, or 75% of the possible reversal cases were tested for during simulation, for a coverage of 75%.

Model Coverage for the Embedded MATLAB Function Block and the Model

The model coverage report for the block `Embedded MATLAB Function` shows that it has no decisions of its own apart from its function. However, it does repeat the summary information for its function `run_intersect_test` as coverage for its descendent objects, as shown.



Because there are no additional coverage objects in the model apart from the Embedded MATLAB Function block, the remaining report for the model `intersecting_rectangles` also repeats the preceding coverage for descendent objects, as shown.


intersecting_rectangles Coverage Report
File Edit View Go Debug Desktop Window Help
Location: examples/model_coverage/intersecting_rectangles_cov.html
Summary | Details | Help
1. Model "intersecting_rectangles"
Child Systems: Embedded MATLAB Function
Metric
Coverage (this object)
Coverage (inc. descendants)
Cyclomatic Complexity
1
9
Decision (D1)
NA
100% (8/8) decision outcomes
Condition (C1)
NA
88% (7/8) condition outcomes
MCDC (C1)
NA
75% (3/4) conditions reversed the outcome

6

Customizing the Model Advisor

Why Customize the Model Advisor?

The Model Advisor is a tool that runs a set of checks on a Simulink model or subsystem to identify conditions and configuration settings that cause inaccurate or inefficient simulation or code generation. For more information, see "Consulting the Model Advisor" in the Simulink documentation.

The Simulink Verification and Validation software provides an API that allows you to customize the behavior of the Model Advisor by defining your own custom checks and folders, and writing your own callback functions.

Before Customizing the Model Advisor

Before customizing the Model Advisor:

- Know how to create an M-file.
- Identify which MathWorks checks you want to include in your custom Model Advisor view.
- Have a plan for organizing the Model Advisor tree.
- Understand how to access model constructs that you want to check. For example, know how to find block and model parameters. For more information on using utilities for creating check callbacks, see "Common Utilities for Authoring Checks" on page 6-29.

When you are ready to customize the Model Advisor, follow the "Customization Workflow" on page 6-3 to create your custom checks and folders. The code examples in each section provide you with detailed examples of the code required to create that piece of the `sl_customization.m` file.

Customization Workflow

To customize the Model Advisor, perform the following high-level tasks:

1. Create an M-file called `sl_customization.m` on your MATLAB path. In this file, create a `sl_customization()` function to register checks, tasks, folders, and optional process callbacks with the Model Advisor. For detailed information, see "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7.

2. Define checks, tasks, and folders, and where they appear in the Model Advisor tree. For detailed information, see "Defining Custom Checks" on page 6-13 and "Defining Custom Tasks and Folders" on page 6-23.

3. Specify what actions you want the Model Advisor to take for the checks by creating a check callback function for each check. For detailed information, see "Creating Callback Functions and Results" on page 6-28.

4. Optionally, specify what automatic fix operations the Model Advisor performs by creating an action callback function. For detailed information, see "Action Callback Function" on page 6-43.

5. Optionally, specify startup and post-execution actions by creating a process callback function. For detailed information, see "Defining Startup and Post-Execution Actions" on page 6-9.

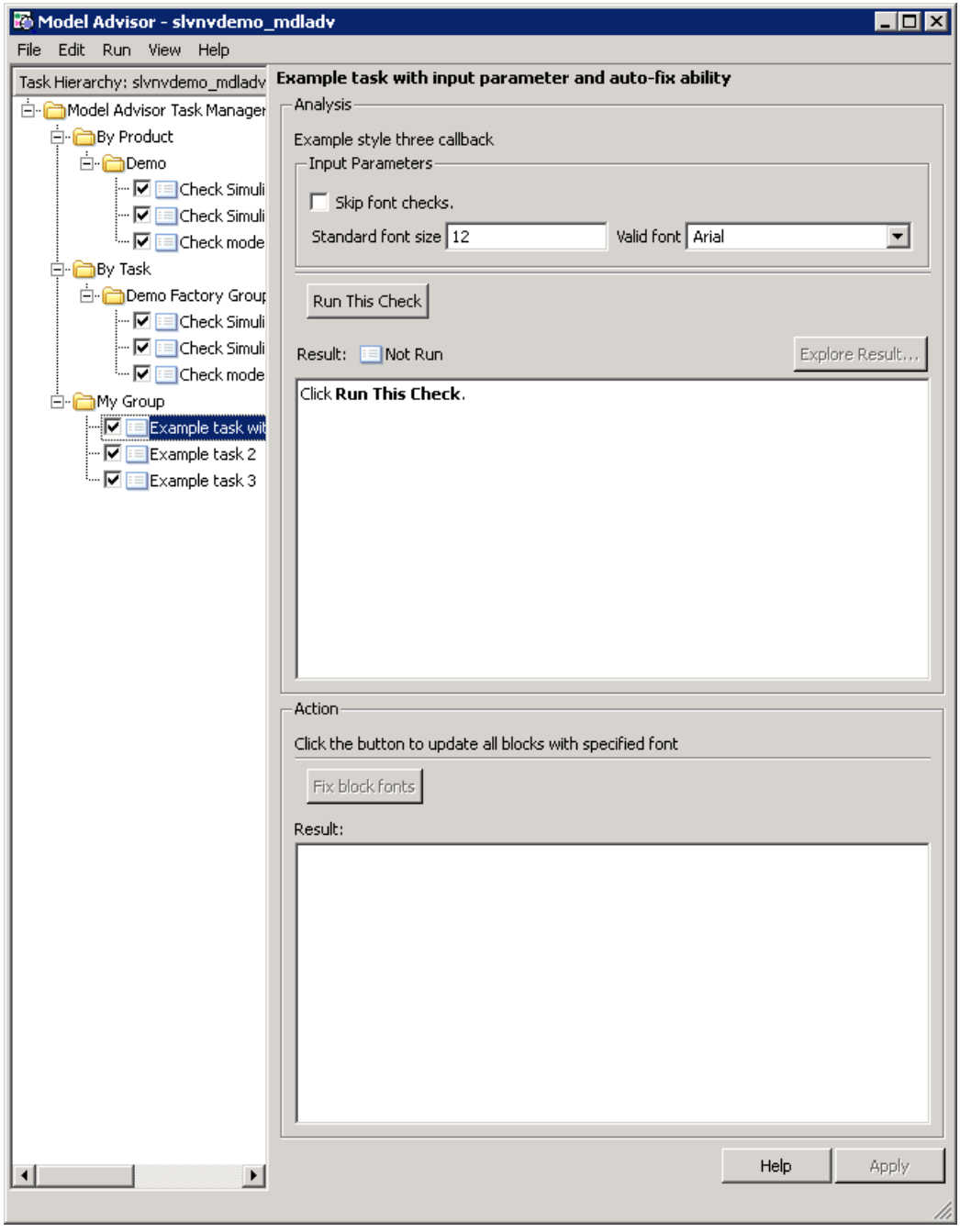
The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

Function	Description	When Required
`sl_customization()`	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at startup (see "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7).	Required for all customizations to the Model Advisor.

Function	Description	When Required
One or more check definitions	Defines all custom checks (see "Defining Custom Checks" on page 6-13).	Required for custom checks and to add custom checks to the **By Product** folder.
One or more task definitions	Defines all custom tasks (see "Defining Custom Tasks and Folders" on page 6-23).	Required to add custom checks to the Model Advisor tree, except when adding the checks to the **By Product** folder. You must write one task for each check that you add to the tree.
One or more groups	Defines all custom groups (see "Defining Custom Tasks and Folders" on page 6-23).	Required to add custom tasks to new folders in the Model Advisor tree, except when adding a new subfolder to the **By Product** folder. You must write one group definition for each new folder.
Check callback functions	Defines the actions of the custom checks (see "Creating Callback Functions and Results" on page 6-28).	Required for custom checks. You must write one callback function for each custom check.
One process callback function	Specifies actions to be performed at startup and post-execution time (see "Defining Startup and Post-Execution Actions" on page 6-9).	Optional.
One or more calls to check input parameters	Specifies input parameters to custom checks (see "Defining Check Input Parameters" on page 6-17).	Optional.

Function	Description	When Required
One or more calls to check list views	Specifies calls to the Model Advisor Result Explorer for custom checks (see "Defining Model Advisor Result Explorer Views" on page 6-19).	Optional.
One or more calls to check actions	Specifies actions the software performs for custom checks (see "Defining Check Actions" on page 6-21 and "Action Callback Function" on page 6-43).	Optional.

The following is an example of the Model Advisor that has custom checks defined in a custom groups. The selected check includes input parameters, list view parameters, and actions.


Model Advisor - slvnvdemo_mdladv
File Edit Run View Help
Task Hierarchy: slvnvdemo_mdladv
Model Advisor Task Manager
By Product
Demo
Check Simuli
Check Simuli
Check mode
By Task
Demo Factory Group
Check Simuli
Check Simuli
Check mode
My Group
Example task wit
Example task 2
Example task 3
Example task with input parameter and auto-fix ability
Analysis
Example style three callback
Input Parameters
Skip font checks.
Standard font size 12
Valid font Arial
Run This Check
Result: Not Run
Explore Result...
Click Run This Check.
Action
Click the button to update all blocks with specified font
Fix block fonts
Result:
Help
Apply

Register Checks, Tasks, Folders, and Process Callbacks

In this section...

Create sl_customization Function

To add checks and folders to the Model Advisor tree, you must create the `sl_customization()` function in the `sl_customization.m` file on your MATLAB path.

Note

- You can have more than one `sl_customization.m` file on your MATLAB path.
- Do not place an `sl_customization.m` file that customizes the Model Advisor in your root MATLAB directory or any of its subdirectories, with the exception of the *`matlabroot`*`/work` directory. Otherwise, the Model Advisor ignores the customizations that the file specifies.

The `sl_customization` function accepts one argument, a handle to an object called `Simulink.CustomizationManager`, as in this example:

```
function sl_customization(cm)
```

The customization manager object includes methods for registering custom checks, tasks, folders, and process callbacks. Use these methods to register customizations specific to your application, as described in the sections that follow.

Registering Checks, Tasks, Folders, and Process Callbacks

The `Simulink.CustomizationManager` class includes the following methods for registering custom checks, tasks, folders, and process callbacks:

- `addModelAdvisorCheckFcn (@checkDefinitionFcn)`

  Registers the checks you define in *`checkDefinitionFcn`* to the **By Product** folder of the Model Advisor.

  The *`checkDefinitionFcn`* argument is a handle to the function that defines all custom checks to be added to the Model Advisor as instances of the `ModelAdvisor.Check` class (see "Defining Custom Checks" on page 6-13).

- `addModelAdvisorTaskFcn (@factorygroupDefinitionFcn)`

  Registers the checks you define in *`factorygroupDefinitionFcn`* to the **By Task** folder of the Model Advisor.

  The *`factorygroupDefinitionFcn`* argument is a handle to the function that defines the checks to add to the Model Advisor as instances of the `ModelAdvisor.FactoryGroup` class (see "Defining Custom Tasks and Folders" on page 6-23).

- `addModelAdvisorTaskAdvisorFcn (@taskDefinitionFcn)`

  Registers the tasks and folders defined in *`taskDefinitionFcn`* to the folder in the Model Advisor tree that you specify using the `ModelAdvisor.Root.publish` method or the `ModelAdvisor.Group` class.

  The *`taskDefinitionFcn`* argument is a handle to the function that defines all custom tasks and folders to be added to the Model Advisor as instances of the `ModelAdvisor.Task` or `ModelAdvisor.Group` classes (see "Defining Custom Tasks and Folders" on page 6-23).

- `addModelAdvisorProcessFcn (@modelAdvisorProcessFcn)`

  Registers the process callback function for the Model Advisor checks (see "Defining Startup and Post-Execution Actions" on page 6-9).

Note The @ sign defines a function handle that MATLAB calls. For more information, see "At — @" in the MATLAB documentation.

Code Example: Registering Custom Checks, Tasks, Folders, and Process Callbacks

The following code example registers custom checks, tasks, folders, and a process callback function:

```
function sl_customization(cm)

% Register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

% Register custom tasks and folder
cm.addModelAdvisorTaskAdvisorFcn(@defineTaskAdvisor);

% Register custom process callback functions
cm.addModelAdvisorProcessFcn(@ModelAdvisorProcessFunction);
```

Defining Startup and Post-Execution Actions

Define Actions Using Process Callback Functions

The *process callback function* is an optional function that you can use to configure the Model Advisor tree and process check results at run time. The process callback function specifies actions that the software performs at different stages of Model Advisor execution:

- `configure` stage: The Model Advisor executes `configure` actions at startup, after all checks and tasks have been initialized. At this stage, you can customize how the Model Advisor constructs lists of checks and tasks by modifying `Visible`, `Enable`, and `Value` properties. For example, you can remove, rename, and selectively display checks and tasks.
- `process_results` stage: The Model Advisor executes `process_results` actions after checks complete execution. You can specify actions to examine and report on the results returned by check callback functions.

If you create a process callback function, you must register it, as described in "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7. The sections that follow provide mode information about defining your own process callback functions.

Process Callback Function Arguments

The process callback function takes the following arguments.

Argument	I/O Type	Data Type	Description
`stage`	Input	Enumeration	Specifies the stages at which process callback actions are executed. Use this argument in a switch statement to specify actions for the stages `configure` and `process_results`.
`system`	Input	Path	Model or subsystem to be analyzed by the Model Advisor.
`checkCellArray`	Input/Output	Cell array	As input, the array of checks constructed in the check definition function. As output, the array of checks modified by actions in the configure stage.
`taskCellArray`	Input/Output	Cell array	As input, the array of tasks constructed in the task definition function. As output, the array of tasks modified by actions in the configure stage.

Code Example: Process Callback Function

Here is an example of a process callback function that specifies actions in the `configure` stage to enable only custom checks. In the `process_results`

stage, this function displays a information at the MATLAB command line for checks that do not pass.

```
% Process Callback Function
% Defines actions to execute at startup and post-execution
function [checkCellArray taskCellArray] = ...
    ModelAdvisorProcessFunction(stage, system, checkCellArray, taskCellArray)
switch stage
    % Specify the appearance of the Model Advisor tree at startup
    case 'configure'
        for i=1:length(checkCellArray)
            % Hide all checks that do not belong to custom folder
            if isempty(strfind(checkCellArray{i}.ID, 'mathworks.example'))
                checkCellArray{i}.Visible = false;
                checkCellArray{i}.Value = false;
            end
        end
    % Specify actions to perform after the Model Advisor completes execution
    case 'process_results'
        for i=1:length(checkCellArray)
            % Print message if check does not pass
            if checkCellArray{i}.Selected && (strcmp(checkCellArray{i}.Title, ...
                    'Check Simulink window screen color'))
                mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
                % Verify whether the check was run and if it failed
                if mdladvObj.verifyCheckRan(checkCellArray{i}.ID)
                    if ~mdladvObj.getCheckResultStatus(checkCellArray{i}.ID)
                        % Display text in MATLAB Command Window
                        disp(['Example message from Model Advisor Process'...
                            ' callback.']);
                    end
                end
            end
        end
end
```

Updating the sl_customization File

When Simulink starts, it reads `sl_customization.m` files. If you subsequently change the contents of your customization file, update your environment by performing these tasks:

1. If you previously opened the Model Advisor:
   a. Close the model from which you opened the Model Advisor
   b. Reinitialize the Model Advisor by removing the `slprj` folder from your working directory.
   .
2. At the MATLAB command line, enter:

   ```
   sl_refresh_customizations
   ```

3. Open your model.
4. Start the Model Advisor.

Defining Custom Checks

In this section...

About Custom Checks

You can create a new check to use in the Model Advisor. You define custom checks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Check` class. You must define one instance of this class for each custom check that you want to add to the Model Advisor, and register the custom check as described in "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7.

Tip You can add a check to multiple folders by creating a *task*. For more information, see "Adding a Check to Custom or Multiple Folders Using Tasks" on page 6-23.

The sections that follow describe how to define custom checks.

Contents of Check Definitions

When you define a Model Advisor check, it contains:

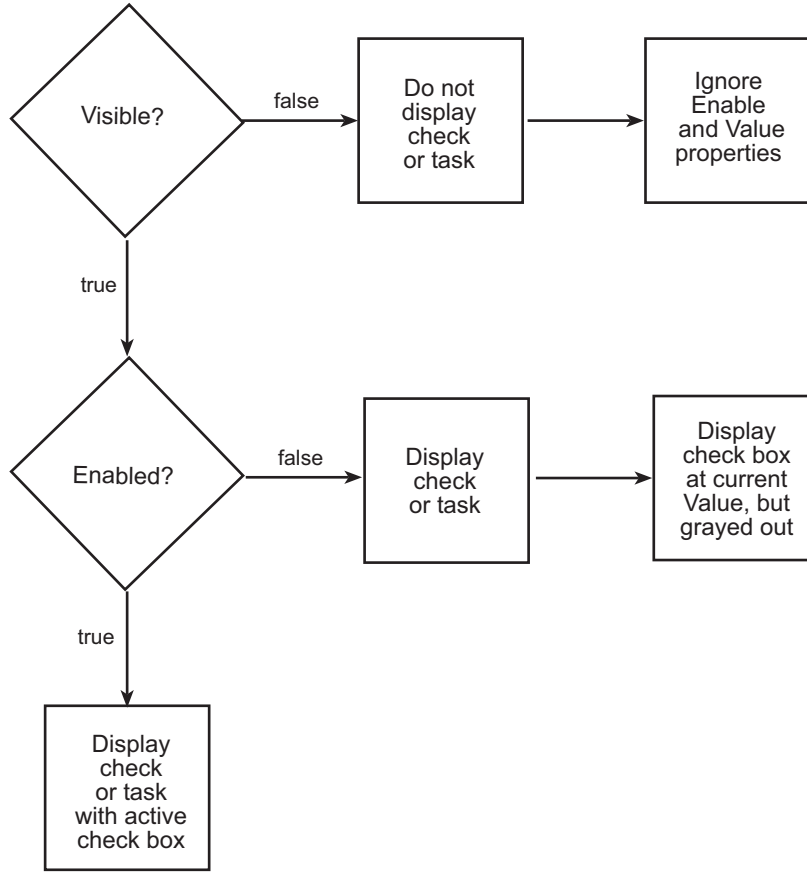
Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this id to access the check.
Handle to check callback function (required)	Function that specifies the contents of a check.
Check name (recommended)	Creates a name for the check that the Model Advisor displays.
Check properties (optional)	Creates user interface with the check. When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`. For more information on properties, see `ModelAdvisor.Check` and `ModelAdvisor.Task`.
Input Parameters (optional)	Adds input parameters used to request input from the user. The Model Advisor uses the input to perform the check.
Action (optional)	Adds automatic fixing action.
Explore Result button (optional)	Adds the **Explore Result** button that the user clicks to open the Model Advisor Result Explorer.

Displaying and Enabling Checks

You can create a check and specify how it appears in the Model Advisor tree. You can define when to display a check, or whether a user is able to select or clear a check using the `Visible`, `Enable`, and `Value` properties of the `ModelAdvisor.Check` class.

Note When adding checks to the Model Advisor as tasks, specify these properties in the `ModelAdvisor.Task` class. If you specify the properties in both locations, the `ModelAdvisor.Task` properties take precedence, except for the `Visible` property. For more information, see `ModelAdvisor.Task`.

You modify the behavior of the `Visible`, `Enable`, and `Value` properties in a process callback function (see "Defining Startup and Post-Execution Actions" on page 6-9). The following chart illustrates how these properties interact.



Defining Where Custom Checks Appear

You can specify where the Model Advisor places custom checks within the Model Advisor tree using the following guidelines:

- To place a check in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class. See "Defining Custom Tasks and Folders" on page 6-23.
- To place a check in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See "Defining Custom Tasks and Folders" on page 6-23.
- To place a check in the **By Product** folder, use the `ModelAdviso.Root.publish` method.

Code Example: Check Definition Function

The following is an example of a function that defines the custom checks associated with the callback functions described in "Creating Callback Functions and Results" on page 6-28. The check definition function returns a cell array of custom checks to be added to the Model Advisor.

The check definitions in the example are used in the tasks described in "Defining Custom Tasks and Folders" on page 6-23.

Tip When you add checks to the Model Advisor as tasks, you have to specify only the required properties of a check because the task definition includes the additional properties. For example, you define the description of the check in the task definition using the `ModelAdvisor.Task.Description` property instead of the `ModelAdvisor.Check.TitleTips` property. For more details, see the `ModelAdvisor.Task` class documentation.

```
% Defines custom Model Advisor checks
function defineModelAdvisorChecks

% Sample check 1: Informational check
rec = ModelAdvisor.Check('mathworks.example.configManagement');
rec.Title = 'Informational check for model configuration management';
setCallbackFcn(rec, @modelVersionChecksumCallbackUsingFT,'None','StyleOne');
```

```
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample check 2: Basic Check with Pass/Fail Status
rec = ModelAdvisor.Check('mathworks.example.unconnectedObjects');
rec.Title = 'Check for unconnected objects';
setCallbackFcn(rec, @unconnectedObjectsCallbackUsingFT,'None','StyleOne');
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample Check 3: Check with Subchecks and Actions
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
rec.Title = 'Check safety-related optimization settings';
setCallbackFcn(rec, @OptmizationSettingCallback,'None','StyleOne');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
setCallbackFcn(modifyAction, @modifyOptmizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
                            ' settings that can impact safety.'];
modifyAction.Enable = true;
setAction(rec, modifyAction);
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);
```

Defining Check Input Parameters

About Input Parameters

Input parameters allow the check author to request input from the user for a Model Advisor check. You define input parameters using the `ModelAdvisor.InputParameter` class inside a custom check function (see "Defining Custom Checks" on page 6-13). You must define one instance of this class for each input parameter that you want to add to a Model Advisor check.

Note You do not have to create Input parameters for every custom check.

Specifying Input Parameter Layout

You can specify the layout of input parameters in the right pane of the Model Advisor window in an input parameter definition. To place input parameters, use the following methods.

Method	Description
`ModelAdvisor.Check setInputParametersLayoutGrid`	Specifies the size of the input parameter grid.
`ModelAdvisor.InputParameter setRowSpan`	Specifies the number of rows the parameter occupies in the Input Parameter layout grid.
`ModelAdvisor.InputParameter setColSpan`	Specifies the number of columns the parameter occupies in the Input Parameter layout grid.

For information on using these methods, see the `ModelAdvisor.Check` and `ModelAdvisor.InputParameter` class pages.

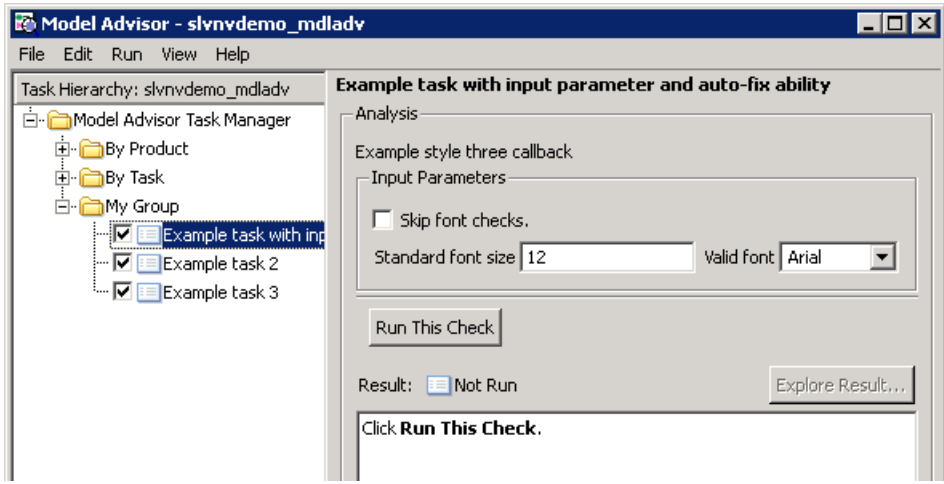
Code Example: Input Parameter Definition

The following is an example of defining input parameters to add to a custom check. You must include input parameter definitions inside a custom check definition (see "Code Example: Check Definition Function" on page 6-16). The following code, when included in a custom check definition, creates three input parameters.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.setInputParametersLayoutGrid([3 2]);
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
```

```
inputParam1.setColSpan([1 1]);
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
```

The Model Advisor displays these input parameters in the right pane, in an **Input Parameters** box.



Defining Model Advisor Result Explorer Views

- "About Model Advisor Result Explorer Views" on page 6-20

About Model Advisor Result Explorer Views

A *list view* allows you to provide the information to populate the Model Advisor Result Explorer window, and adds the **Explore Result** button to a custom check in the Model Advisor window. You define list views using the `ModelAdvisor.ListViewParameter` class inside a custom check function (see "Defining Custom Checks" on page 6-13). You must define one instance of this class for each list view that you want to add to a Model Advisor Result Explorer window. For information on using the Model Advisor Results Explorer, see "Batch-Fixing Warnings or Failures" in the Simulink documentation.

Note You do not have to create list views for every custom check.

Code Example: List View Definition

The following is an example of defining list views. You must make the **Explore Result** button visible using the `ModelAdvisor.Check.ListViewVisible` property inside a custom check function, and include list view definitions inside a check callback function (see "Detailed Check Callback Function" on page 6-37).

The following code, when included in a custom check function, adds the **Explore Result** button to the check in the Model Advisor window.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
% add 'Explore Result' button
rec.ListViewVisible = true;
```

The following code, when included in a check callback function, provides the information to populate the Model Advisor Result Explorer window.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
```

```
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object')';
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

Defining Check Actions

About Actions

Actions allow the check author to specify an action that the Model Advisor performs to fix a Model Advisor check. When you define an action, the Model Advisor window includes an Action box below the Analysis box. You define actions using the `ModelAdvisor.Action` class inside a custom check function (see "Defining Custom Checks" on page 6-13). You must define:

- One instance of this class for each action that you want to take.
- One action callback function for each action (see "Action Callback Function" on page 6-43).

Note

- Each check can contain only one action.
- You do not have to create actions for every custom check.

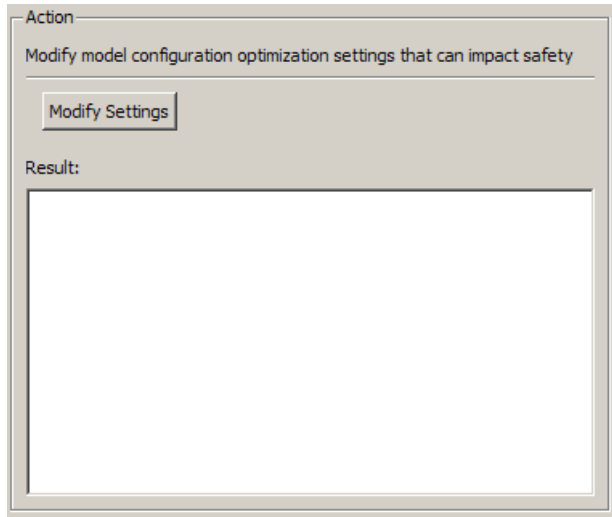
Code Example: Action Definition

The following is an example of defining actions within a custom check. You must include action definitions inside a check definition function (see "Code Example: Check Definition Function" on page 6-16). The following code, when included in a check definition function, provides the information to populate the Action box in the Model Advisor window.

```
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
```

```
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
modifyAction.setCallbackFcn(@modifyOptmizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
                            ' settings that can impact safety'];
modifyAction.Enable = true;
rec.setAction(modifyAction);
```

The Model Advisor, in the right pane, displays an Action box.



Defining Custom Tasks and Folders

In this section...

Adding a Check to Custom or Multiple Folders Using Tasks

Custom tasks provide you with a method for adding checks to the Model Advisor tree, either in multiple folders or in a single, custom folder. You define custom tasks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Task` class. You must define one instance of this class for each custom task that you want to add to the Model Advisor, and register the custom task, as described in "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7. The sections that follow describe how to define custom tasks.

To add a check to multiple folders or a single, custom folder:

1. Create a check using the `ModelAdvisor.Check` class, as described in "Defining Custom Checks" on page 6-13.
2. Register a task wrapper for the check, as described in "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7.
3. If you want to add the check to folders that are not already present, register and create the folders using the `ModelAdvisor.Group` class.
4. Add a check to the task using the `ModelAdvisor.Task.setCheck` method.

5 Add the task to each folder using the `ModelAdvisor.Group.addTask` method and the task ID.

Creating Custom Tasks Using MathWorks Checks

You can add MathWorks checks to your custom folders by defining the checks as custom tasks, as described above.

Find MathWorks check IDs as follows:

1 In the Model Advisor, select **View > Source Tab**.

2 Navigate to the folder that contains the MathWorks check.

3 In the right pane, click **Source**. The Model Advisor displays the **Title**, **TitleID**, and **Source** information for each check in the folder.

4 Select and copy the **TitleID** of the check that you want to add as a task.

Displaying and Enabling Tasks

The `Visible`, `Enable`, and `Value` properties interact the same way for tasks as for checks (see "Displaying and Enabling Checks" on page 6-14).

Defining Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor tree using the following guidelines:

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class. See "Defining Custom Tasks and Folders" on page 6-23.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See "Defining Custom Tasks and Folders" on page 6-23.

Code Example: Task Definition Function

The following is an example of a task definition function. This function defines three tasks. For an example of placing these tasks into a custom group, see "Code Example: Group Definition" on page 6-27.

```
% Defines Model Advisor tasks and a custom folder
% Add checks to a custom folder using task definitions
function defineTaskAdvisor
mdladvRoot = ModelAdvisor.Root;

% Define task that uses Sample Check 1: Informational check
MAT1 = ModelAdvisor.Task('mathworks.example.task.configManagement');
MAT1.DisplayName = 'Informational check for model configuration management';
MAT1.Description = 'Display model configuration and checksum information.';
setCheck(MAT1, 'mathworks.example.configManagement');
mdladvRoot.register(MAT1);

% Define task that uses Sample Check 2: Basic Check with Pass/Fail Status
MAT2 = ModelAdvisor.Task('mathworks.example.task.unconnectedObjects');
MAT2.DisplayName = 'Check for unconnected objects';
setCheck(MAT2, 'mathworks.example.unconnectedObjects');
MAT2.Description = ['Identify unconnected lines, input ports, and output ' ...
                                        'ports in the model or subsystem.'];
mdladvRoot.register(MAT2);

% Define task that uses Sample Check 3: Check with Subresults and Actions
MAT3 = ModelAdvisor.Task('mathworks.example.task.optimizationSettings');
MAT3.DisplayName = 'Check safety-related optimization settings';
MAT3.Description = ['Check model configuration for optimization ' ...
                    'settings that can impact safety.'];
MAT3.setCheck('mathworks.example.optimizationSettings');
mdladvRoot.register(MAT3);

% Custom folder definition
MAG = ModelAdvisor.Group('mathworks.example.ExampleGroup');
MAG.DisplayName = 'My Group';
% Add tasks to My Group folder
addTask(MAG, MAT1);
addTask(MAG, MAT2);
addTask(MAG, MAT3);
% Add My Group folder to tree under Model Advisor Task Manager (root)
mdladvRoot.publish(MAG);
```

About Custom Folders

Folders are used to group checks in the Model Advisor by functionality or usage. You define custom folders in:

- A factory group definition function that specifies the properties of each instance of the `ModelAdvisor.FactoryGroup` class.
- A task definition function that specifies the properties of each instance of the `ModelAdvisor.Group` class. For more information about task definition functions, see "Adding a Check to Custom or Multiple Folders Using Tasks" on page 6-23.

You must define one instance of the group classes for each folder that you want to add to the Model Advisor, and register the custom folder, as described in "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7. The sections that follow describe how to define custom groups.

Adding Custom Folders

To add a custom folder:

1. Create the folder using the `ModelAdvisor.Group` or `ModelAdvisor.FactoryGroup` classes.
2. Add the folder to the Model Advisor tree, as described in "Register Checks, Tasks, Folders, and Process Callbacks" on page 6-7.

Defining Where Custom Folders Appear

You can specify where custom folders are placed within the Model Advisor tree using the following guidelines:

- To define a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To define a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Note To define a new folder in the **By Product** folder, you must use the `ModelAdvisor.Root.publish` method within a custom check. For more information, see "Defining Where Custom Checks Appear" on page 6-16.

Code Example: Group Definition

The following is an example of a group definition that places the tasks defined in "Code Example: Task Definition Function" on page 6-24 inside a folder called **My Group** under the **Model Advisor Task Manager** folder. This group definition is included in the task definition function.

```
% Custom folder definition
MAG = ModelAdvisor.Group('mathworks.example.ExampleGroup');
MAG.DisplayName='My Group';
% Add tasks to My Group folder
MAG.addTask(MAT1);
MAG.addTask(MAT2);
MAG.addTask(MAT3);
% Add My Group folder to tree under Model Advisor Task Manager (root)
mdladvRoot.publish(MAG);
```

The following is an example of a factory group definition function that places the checks defined in "Code Example: Check Definition Function" on page 6-16 into a folder called **Demo Factory Group** inside of the **By Task** folder.

```
function defineModelAdvisorTasks
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='Demo Factory Group';
rec.Description='Demo Factory Group';
rec.addCheck('mathworks.example.configManagement');
rec.addCheck('mathworks.example.unconnectedObjects');
rec.addCheck('mathworks.example.optimizationSettings');
mdladvRoot.publish(rec); % publish inside By Task
```

Creating Callback Functions and Results

In this section...

About Callback Functions

A *callback function* specifies the actions the Model Advisor performs on a model or subsystem, based on the check or action that the user runs. You must create a callback function for each custom check and action so that the Model Advisor can execute the function when the check is run by a user. There are several types of callback functions:

- "Simple Check Callback Function" on page 6-29
- "Detailed Check Callback Function" on page 6-37
- "Check Callback Function with Hyperlinked Results" on page 6-39
- "Action Callback Function" on page 6-43

All types of callback functions provide one or more return arguments for displaying the results after executing the check or action. In some cases, return arguments are strings or cell arrays of strings that support embedded HTML tags for text formatting. The MathWorks™ recommends that you use the Model Advisor Result Template API to format check results, as described in "Formatting Model Advisor Results" on page 6-44. Limit HTML tags to be compatible with alternate output formats.

Common Utilities for Authoring Checks

When you create a check, there are common Simulink utilities that you can use to make the check perform different actions. Following is a list of the utilities and when to use them. Click the link in the Utility column for more information about the utility.

Utility	Used for...
`find_system`	Getting handle or path to: • Blocks • Lines • Annotations When getting the object, you can choose to: • Specify a search depth • Search under masks and libraries
`get_param` / `set_param`	Getting and setting system and block parameter values.
`inspect`	Getting object properties. First you must get a handle to the object.
`simget` / `simset`	Getting and setting model simulation parameters.
`evalin`	Working in the base workspace.
Stateflow API	Programmatic access to Stateflow objects.

Simple Check Callback Function

Use a simple check callback function with results formatted using the Result Template API to indicate whether the model passed or failed the check, or to provide a recommendation for correcting an issue. The keyword for this callback function is `StyleOne`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-13).

The check callback function takes the following arguments.

Argument	I/O Type	Description
`system`	Input	Path to the model or subsystem analyzed by the Model Advisor.
`result`	Output	MATLAB string that supports Model Advisor Formatting API calls or embedded HTML tags for text formatting.

Code Example: Informational Check Callback Function

Here is an example of a callback function for a custom informational check that finds and displays the model configuration and checksum information. The informational check uses the Result Template API to format the check result.

An informational check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.

An informational check does not include the following items in the results:

- The check status. The Model Advisor displays the overall check status, but the status is not in the result.
- A description of the status.
- The recommended action to take when the check does not pass.
- Subcheck results.
- A line below the results.

```
% Sample Check 1 Callback Function: Informational Check
% Find and display model configuration and checksum information
% Informational checks do not have a passed or warning status in the results

function resultDescription = modelVersionChecksumCallbackUsingFT(system)
resultDescription = [];
```

```
system = getfullname(system);
model = bdroot(system);

% Format results in a list using Model Advisor Result Template API
ft = ModelAdvisor.FormatTemplate('ListTemplate');
% Add See Also section for references to standards
docLinkSfunction{1}     = {['IEC 61508-3, Table A.8 (5)' ...
                            ' ''Software configuration management'' ']};
setRefLink(ft,docLinkSfunction);

% Description of check in results
desc = 'Display model configuration and checksum information.';
% If running the Model Advisor on a subsystem, add note to description
if strcmp(system, model) == false
    desc = strcat(desc, ['<br/>NOTE: The Model Advisor is reviewing a' ...
        ' sub-system, but these results are based on root-level settings.']);
end
setCheckText(ft, desc);

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% If err, use these values
mdlver = 'Error - could not retrieve Version';
mdlauthor = 'Error - could not retrieve Author';
mdldate = 'Error - could not retrieve Date';
mdlsum = 'Error - could not retrieve CheckSum';

% Get model configuration and checksum information
try
    mdlver = get_param(model,'ModelVersion');
    mdlauthor = get_param(model,'LastModifiedBy');
    mdldate = get_param(model,'LastModifiedDate');
    mdlsum = Simulink.BlockDiagram.getChecksum(model);
    mdlsum = [num2str(mdlsum(1)) ' ' num2str(mdlsum(2)) ' ' ...
                    num2str(mdlsum(3)) ' ' num2str(mdlsum(4))];
    mdladvObj.setCheckResultStatus(true); % init to true
catch err
    mdladvObj.setCheckResultStatus(false);
    setSubResultStatusText(ft,err.message);
    resultDescription{end+1} = ft;
    return
```

```
end

% Display the results
lbStr ='<br/>';
resultStr = ['Model Version: ' mdlver lbStr 'Author: ' mdlauthor lbStr ...
    'Date: ' mdldate lbStr 'Model Checksum: ' mdlsum];
setSubResultStatusText(ft,resultStr);

% Informational checks do not have subresults, supress line
setSubBar(ft,false);
resultDescription{end+1} = ft;
```

Code Example: Basic Check with Pass/Fail Status

Here is an example of a callback function for a custom basic check that finds and reports unconnected lines, input ports, and output ports.

A basic check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.
- The status of the check.
- A description of the status.
- Results for the check.
- The recommended action to take when the check does not pass.

A basic check does not include the following items in the results:

- Subcheck results.
- A line below the results.

```
% Sample Check 2 Callback Function: Basic Check with Pass/Fail Status
% Find and report unconnected lines, input ports, and output ports
function ResultDescription = unconnectedObjectsCallbackUsingFT(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% Initialize variables
mdladvObj.setCheckResultStatus(false);
```

```
ResultDescription ={};
ResultStatus = false; % Default check status is 'Warning'
system = getfullname(system);
isSubsystem = ~strcmp(bdroot(system), system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object
ft = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
if isSubsystem
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                                        'output ports in the subsystem.'];
else
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                                        'output ports in the model.'];
end
setCheckText(ft,checkDescStr);

% Add See Also section with references to applicable standards
checkStdRef = 'IEC 61508-3, Table A.3 (3) ''Language subset'' ';
docLinkSfunction{1}     = {checkStdRef};
setRefLink(ft,docLinkSfunction);

% Basic checks do not have subresults, supress line
setSubBar(ft,false);

% Check for unconnected lines, inputs, and outputs
sysHandle = get_param(system, 'Handle');
uLines = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'line', ...
    'Connected', 'off');
uPorts = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'port', ...
    'Line', -1);
```

```
% Use parents of port objects for the correct highlight behavior
if ~isempty(uPorts)
    for i=1:length(uPorts)
        uPorts(i) = get_param(get_param(uPorts(i), 'Parent'), 'Handle');
    end
end

% Create cell array of unconnected object handles
modelObj = {};
searchResult = union(uLines, uPorts);
for i = 1:length(searchResult)
    modelObj{i} = searchResult(i);
end

% No unconnected objects in model
% Set result status to 'Pass' and display text describing the status
if isempty(modelObj)
    setSubResultStatus(ft,'Pass');
    if isSubsystem
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this subsystem.']);
    else
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this model.']);
    end
    ResultStatus            = true;
% Unconnected objects in model
% Set result status to 'Warning' and display text describing the status
else
    setSubResultStatus(ft,'Warn');
    if ~isSubsystem
        setSubResultStatusText(ft,['The following lines, input ports, ' ...
         'or output ports are not properly connected in the system: ' system]);
    else
        setSubResultStatusText(ft,['The following lines, input ports, or ' ...
         'output ports are not properly connected in the subsystem: ' system]);
    end
    % Specify recommended action to fix the warning
    setRecAction(ft,'Connect the specified blocks.');
    % Create a list of handles to problem objects
```

```
    setListObj(ft,modelObj);
    ResultStatus = false;
end
% Pass the list template object to the Model Advisor
ResultDescription{end+1} = ft;
% Set overall check status
mdladvObj.setCheckResultStatus(ResultStatus);
```

Code Example: Check With Subchecks and Actions

Here is an example of a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting, and the second reviews the **Conditional input branch execution** optimization setting.

A check with subchecks includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.
- Recommended actions to take when the subcheck does not pass.
- A line between the subcheck results.

```
% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
function ResultDescription = OptmizationSettingCallback(system)
% Initialize variables
system =getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};
system = bdroot(system);
```

```
% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,['Check model configuration for optimization settings that'...
    'can impact safety.']);

% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction setting');
setInformation(ft1,'Check whether the ''Block reduction'' check box is cleared.');
% Add See Also section with references to applicable standards
docLinks{1}     = {['Reference DO-178B Section 6.3.4e - Source code ' ...
    'is traceable to low-level requirements']};
% Review 'Block reduction' optimization
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is cleared.');
    ResultStatus = true;
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is selected.');
    setRecAction(ft1,['Clear the ''Optimization > Block reduction''' ...
        ' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check whether the ''Conditional input branch execution'''...
    ' check box is cleared.'])
```

```
% Add See Also section and references to applicable standards
docLinks{1} = {['Reference DO-178B Section 6.4.4.2 - Test coverage ' ...
    'of software structure is achieved']};
setRefLink(ft2,docLinks);

% Last subcheck, supress line
setSubBar(ft2,false);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution''' ...
        'check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution'''...
        ' check box is selected.']);
    setRecAction(ft2,['Clear the ''Optimization > Conditional input branch ' ...
        'execution'' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);
```

Detailed Check Callback Function

Use the detailed check callback function to return and organize results as strings in a layered, hierarchical fashion. The function provides two output arguments that allow you to associate text descriptions with one or more paragraphs of detailed information. The keyword for the detailed callback function is `StyleTwo`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-13).

The detailed callback function takes the following arguments.

Argument	I/O Type	Description
`system`	Input	Path to the model or system analyzed by the Model Advisor.
`ResultDescription`	Output	Cell array of MATLAB strings that supports Model Advisor Formatting API calls or embedded HTML tags for text formatting. The Model Advisor concatenates the `ResultDescription` string with the corresponding array of `ResultDetails` strings.
`ResultDetails`	Output	Cell array of cell arrays, each of which contains one or more strings.

Note The `ResultDetails` cell array must be the same length as the `ResultDescription` cell array.

Here is an example of a detailed check callback function that checks optimization settings for simulation and code generation.

```
function [ResultDescription, ResultDetails] = SampleStyleTwoCallback(system)
ResultDescription ={};
ResultDetails ={};

model = bdroot(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object
mdladvObj.setCheckResultStatus(true);  % init result status to pass

% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '...
    'optimization settings:']);
if strcmp(get_param(model,'BlockReduction'),'off');
    ResultDetails{end+1}     = {ModelAdvisor.Text(['It is recommended to '...
        'turn on Block reduction optimization option.',{'italic'}])};
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
```

```
else
    ResultDetails{end+1}     = {ModelAdvisor.Text('Passed',{'pass'})};
end

% Check code generation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check code generation '...
    'optimization settings:']);
ResultDetails{end+1}  = {};
if strcmp(get_param(model,'LocalBlockOutputs'),'off');
    ResultDetails{end}{end+1}     = ModelAdvisor.Text(['It is recommended to'...
        ' turn on Enable local block outputs option.',{'italic'}]);
    ResultDetails{end}{end+1}     = ModelAdvisor.LineBreak;
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if strcmp(get_param(model,'BufferReuse'),'off');
    ResultDetails{end}{end+1}     = ModelAdvisor.Text(['It is recommended to'...
        ' turn on Reuse block outputs option.',{'italic'}]);
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if isempty(ResultDetails{end})
    ResultDetails{end}{end+1}     = ModelAdvisor.Text('Passed',{'pass'});
end
```

Check Callback Function with Hyperlinked Results

This callback function automatically displays hyperlinks for every object returned by the check so that you can easily locate problem areas in your model or subsystem. The keyword for this type of callback function is `StyleThree`. This keyword is required for the check definition (see "Defining Custom Checks" on page 6-13).

This callback function takes the following arguments.

Argument	I/O Type	Description
`system`	Input	Path to the model or system analyzed by the Model Advisor.

Argument	I/O Type	Description
`ResultDescription`	Output	Cell array of MATLAB strings that supports the Model Advisor Formatting API calls or embedded HTML tags for text formatting.
`ResultDetails`	Output	Cell array of cell arrays, each of which contains one or more Simulink objects such as blocks, ports, lines, and Stateflow charts. The objects must be in the form of a handle or Simulink path.

Note The `ResultDetails` cell array must be the same length as the `ResultDescription` cell array.

The Model Advisor automatically concatenates each string from `ResultDescription` with the corresponding array of objects from `ResultDetails`. The Model Advisor displays the contents of `ResultDetails` as a set of hyperlinks, one for each object returned in the cell arrays. When you click a hyperlink, the Model Advisor displays the target object highlighted in your Simulink model.

The following is an example of a check callback function with hyperlinked results. This example checks a model for consistent use of font type and font size in its blocks. It also contains input parameters, actions, and a call to the Model Advisor Result Explorer, which are discussed in later sections.

```
function [ResultDescription, ResultDetails] = SampleStyleThreeCallback(system)
ResultDescription ={};
ResultDetails ={};

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);
needEnableAction = false;
% get input parameters
inputParams = mdladvObj.getInputParameters;
skipFontCheck = inputParams{1}.Value;
```

```
regularFontSize = inputParams{2}.Value;
regularFontName = inputParams{3}.Value;
if skipFontCheck
    ResultDescription{end+1} = ModelAdvisor.Paragraph('Skipped.');
    ResultDetails{end+1}     = {};
    return
end
regularFontSize = str2double(regularFontSize);
if regularFontSize<1 || regularFontSize>=99
    mdladvObj.setCheckResultStatus(false);
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['Invalid font size. '...
        'Please enter a value between 1 and 99']);
    ResultDetails{end+1}     = {};
end

% find all blocks inside current system
allBlks = find_system(system);

% block diagram doesn't have font property
% get blocks inside current system that have font property
allBlks = setdiff(allBlks, {system});

% find regular font name blocks
regularBlks = find_system(allBlks,'FontName',regularFontName);

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font other than ' regularFontName ': ']);
    ResultDetails{end+1}     = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult,'object')';
    myLVParam.Attributes = {'FontName'}; % name is default property
    mdladvObj.setListViewParameters({myLVParam});
    needEnableAction = true;
else
```

```
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font names '...
        'are identical.']);
    ResultDetails{end+1}     = {};
end

% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font size for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font size other than ' ...
        num2str(regularFontSize) ': ']);
    ResultDetails{end+1}     = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font size blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult,'object')';
    myLVParam.Attributes = {'FontSize'}; % name is default property
    mdladvObj.setListViewParameters...
        ({mdladvObj.getListViewParameters{:}, myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font sizes '...
        'are identical.']);
    ResultDetails{end+1}     = {};
end

mdladvObj.setActionEnable(needEnableAction);
mdladvObj.setCheckErrorSeverity(1);
```

If you run **Example task with input parameter and auto-fix ability** for the `slvnvdemo_mdladv` model in the Model Advisor, you can view the hyperlinked results. Clicking the first hyperlink, `slvnvdemo_mdladv/Input`, displays the Simulink model with the Input block highlighted.

Action Callback Function

An action callback function specifies the actions that the Model Advisor performs on a model or subsystem when the user clicks the action button. You must create one callback function for the action that you want to take.

The action callback function takes the following arguments.

Argument	I/O Type	Description
`taskobj`	Input	The `ModelAdvisor.Task` object for the check that includes an action definition.
`result`	Output	MATLAB string that supports Model Advisor Formatting API calls or embedded HTML tags for text formatting.

Code Example: Action Callback Function

The following is an example of an action callback function that updates all of the blocks in the model with the font specified in the Input Parameter defined in “Code Example: Input Parameter Definition” on page 6-18.

```
% Sample Check 3 Action Callback Function: Check with Subresults and Actions
% Fix optimization settings
function result = modifyOptmizationSetting(taskobj)
% Initialize variables
result =  ModelAdvisor.Paragraph();
mdladvObj = taskobj.MAObj;
system = bdroot(mdladvObj.System);

% 'Block reduction' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'BlockReduction'),'off')
    set_param(system,'BlockReduction','off');
    result.addItem(ModelAdvisor.Text( ...
        'Cleared the ''Block reduction'' check box.',{'Pass'}));
    result.addItem(ModelAdvisor.LineBreak);
end
% 'Conditional input branch execution' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
```

```
    set_param(system,'ConditionallyExecuteInputs','off');
    result.addItem(ModelAdvisor.Text( ...
       'Cleared the ''Conditional input branch execution'' check box.', ...
   {'Pass'}));
end
```

Formatting Model Advisor Results

Things to Consider When Displaying Results

You can make all of the analysis results of your custom checks appear similar to each other with minimal scripting using the Model Advisor `ModelAdvisor.FormatTemplate` class, as described in `ModelAdvisor.FormatTemplate`. For examples of callback functions using the `ModelAdvisor.FormatTemplate` class, see"Simple Check Callback Function" on page 6-29.

If this format template does not meet your needs, or if you want to format action results, use the Model Advisor Formatting API, as described in the following sections.

Formatting Model Advisor Results

You can use the Model Advisor Formatting API to produce formatted outputs in the Model Advisor. The following constructors of the `ModelAdvisor` class provide the ability to format the output. For more information on each constructor and associated methods, click the link in the Constructor column.

Constructor	Description
`ModelAdvisor.Text`	Formats element text.
`ModelAdvisor.Paragraph`	Combines elements into paragraphs.
`ModelAdvisor.List`	Creates a list of elements.
`ModelAdvisor.LineBreak`	Adds a line break between elements.
`ModelAdvisor.Table`	Creates a table.
`ModelAdvisor.Image`	Adds an image to the output.

Formatting Text

Text is the simplest form of output. You can format text in many different ways. The default text formatting is:

- Empty
- Default color (black)
- Unformatted (not bold, italicized, underlined, linked, subscripted, or superscripted)

To change text formatting, use the `ModelAdvisor.Text` constructor. When you want one type of formatting for all text, use this syntax:

```
ModelAdvisor.Text(content, {attributes})
```

When you want multiple types of formatting, you must build the text, as shown:

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' to ensure uniform appearance of model.');

result = [t1, t2, t3, t4, t5];
```

You can add ASCII and Extended ASCII characters using the MATLAB `char` command. For more information, see the `ModelAdvisor.Text` class page.

Formatting Lists

You can create two types of lists: numbered and bulleted. The default list formatting is bulleted. Use the `ModelAdvisor.List` constructor to create and format lists (see `ModelAdvisor.List`). You can create lists with indented subsections, formatted as either numbered or bulleted, as shown:

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));

topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1',{'keyword','bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2',{'keyword','bold'}), subList]);
```

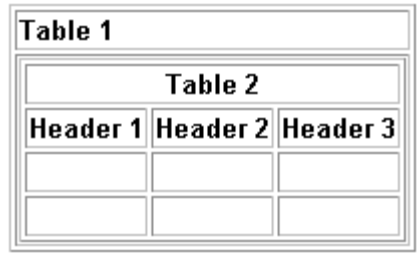
Formatting Tables

The default table formatting is:

- Default color (black)
- Left justified
- Bold title, row, and column headings

You can change table formatting using the `ModelAdvisor.Table` constructor. The following example code creates a subtable within a table, as shown:

```
table1 = ModelAdvisor.Table(1,1);
table2 = ModelAdvisor.Table(2,3);
table2.setHeading('Table 2');
table2.setHeadingAlign('center');
table2.setColHeading(1, 'Header 1');
table2.setColHeading(2, 'Header 2');
table2.setColHeading(3, 'Header 3');
table1.setHeading('Table 1');
table1.setEntry(1,1,table2);
```



Formatting Paragraphs

You must handle paragraphs explicitly because most markup languages do not support line breaks. The default paragraph formatting is:

- Empty
- Default color (black)
- Unformatted, (not bold, italicized, underlined, linked, subscripted, or superscripted)
- Aligned left

If you want to change paragraph formatting, use the `ModelAdvisor.Paragraph` class.

Code Example: Model Advisor Formatted Output

The following is the example from "Simple Check Callback Function" on page 6-29, reformatted using the Model Advisor Formatting API.

```
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
if strcmp(get_param(bdroot(system), 'ScreenColor'),'white')
   result = ModelAdvisor.Text('Passed',{'pass'});
   mdladvObj.setCheckResultStatus(true);
else
   msg1 = ModelAdvisor.Text(...
       ['It is recommended to select a Simulink window screen color'...
       ' of white to ensure a readable and printable model. Click ']);
   msg2 = ModelAdvisor.Text('here');
```

```
    msg2.setHyperlink('matlab: set_param(bdroot,''ScreenColor'',''white'')');
    msg3 = ModelAdvisor.Text(' to change screen color to white.');
    result = [msg1, msg2, msg3];
    mdladvObj.setCheckResultStatus(false);
end
```

Demo and Code Example

The Simulink Verification and Validation software provides a demo that shows how to customize the Model Advisor by adding:

- Custom checks
- Check input parameters
- Check actions
- Check list views to call the Model Advisor Result Explorer
- Custom tasks to include the custom checks in the Model Advisor tree
- Custom folders for grouping the checks
- A process callback function

The demo also provides the source code of the `sl_customization.m` file that executes the customizations.

To run the demo:

1. At the MATLAB command line, type `slvnvdemo_mdladv`.
2. Follow the instructions in the model.

7

Function Reference

Requirements Management Interface

rmi	Interact with Requirements Management Interface
rminav	Start Requirements Management Interface

Model Coverage

add (cv.cvtestgroup)	Add `cvtest` objects
allNames (cv.cvdatagroup)	Get names of all models associated with `cvdata` objects in `cv.cvdatagroup`
allNames (cv.cvtestgroup)	Get names of all models associated with `cvtest` objects in `cvtestgroup`
conditioninfo	Display condition coverage information for model object
cv.cvdatagroup	Create collection of `cvdata` objects for model reference hierarchy
cv.cvtestgroup	Create collection of `cvtest` objects for model reference hierarchy
cvexit	Exit model coverage environment
cvhtml	Produce HTML report from model coverage objects in memory
cvload	Load coverage tests and stored results into memory
cvmodelview	Display model coverage results with model coloring
cvsave	Save coverage tests and results to file
cvsim	Simulate and return model coverage results for test objects
cvsimref	Simulate and return model coverage results for referenced models
cvtest	Create model coverage test specification object
decisioninfo	Display decision coverage information for model object
get (cv.cvdatagroup)	Get `cvdata` object

get (cv.cvtestgroup)	Get `cvtest` objects
getAll (cv.cvdatagroup)	Get all `cvdata` objects
mcdcinfo	Display modified condition/decision coverage information for model object
sigrangeinfo	Display signal range coverage information for model object
tableinfo	Display lookup table coverage information for model object

Model Advisor Customization API

addCheck (ModelAdvisor.FactoryGroup)	Add check to folder
addGroup (ModelAdvisor.Group)	Add subfolder to folder
addTask (ModelAdvisor.Group)	Add task to folder
getID (ModelAdvisor.Check)	Return check identifier
ModelAdvisor.Action	Add actions to custom checks
ModelAdvisor.Check	Create custom checks
ModelAdvisor.FactoryGroup	Define subfolder in **By Task** folder
ModelAdvisor.Group	Define custom folder
ModelAdvisor.InputParameter	Add input parameters to custom checks
ModelAdvisor.ListViewParameter	Add list view parameters to custom checks
ModelAdvisor.Root	Identify root node
ModelAdvisor.Task	Define custom tasks
publish (ModelAdvisor.Root)	Publish object in Model Advisor root
register (ModelAdvisor.Root)	Register object in Model Advisor root
setAction (ModelAdvisor.Check)	Specify action for check
setCallbackFcn (ModelAdvisor.Action)	Specify action callback function
setCallbackFcn (ModelAdvisor.Check)	Specify callback function for check
setCheck (ModelAdvisor.Task)	Specify check used in task
setColSpan (ModelAdvisor.InputParameter)	Specify number of columns for input parameter
setInputParameters (ModelAdvisor.Check)	Specify input parameters for check

setInputParametersLayoutGrid (ModelAdvisor.Check)	Specify layout grid for input parameters
setRowSpan (ModelAdvisor.InputParameter)	Specify rows for input parameter

Model Advisor Result Template API

addRow (ModelAdvisor.FormatTemplate)	Add row to table
ModelAdvisor.FormatTemplate	Construct template object for formatting Model Advisor analysis results
setCheckText (ModelAdvisor.FormatTemplate)	Add description of check to result
setColTitles (ModelAdvisor.FormatTemplate)	Add column titles to table
setInformation (ModelAdvisor.FormatTemplate)	Add description of subcheck to result
setListObj (ModelAdvisor.FormatTemplate)	Add list of hyperlinks to model objects
setRecAction (ModelAdvisor.FormatTemplate)	Add Recommended Action section and text
setRefLink (ModelAdvisor.FormatTemplate)	Add See Also section and links
setSubBar (ModelAdvisor.FormatTemplate)	Add line between subcheck results
setSubResultStatus (ModelAdvisor.FormatTemplate)	Add status to check or subcheck result
setSubResultStatusText (ModelAdvisor.FormatTemplate)	Add text below status in result
setSubTitle (ModelAdvisor.FormatTemplate)	Add title for subcheck in result
setTableInfo (ModelAdvisor.FormatTemplate)	Add data to table
setTableTitle (ModelAdvisor.FormatTemplate)	Add title to table

Model Advisor Formatting API

addItem (ModelAdvisor.List)	Add item to list
addItem (ModelAdvisor.Paragraph)	Add item to paragraph
getEntry (ModelAdvisor.Table)	Get table cell contents
ModelAdvisor.Image	Create custom checks
ModelAdvisor.LineBreak	Insert line break
ModelAdvisor.List	Create list class
ModelAdvisor.Paragraph	Create and format paragraph
ModelAdvisor.Table	Create table
ModelAdvisor.Text	Create Model Advisor text output
setAlign (ModelAdvisor.Paragraph)	Specify paragraph alignment
setBold (ModelAdvisor.Text)	Specify bold text
setColHeading (ModelAdvisor.Table)	Specify table column title
setColHeadingAlign (ModelAdvisor.Table)	Specify column title alignment
setColor (ModelAdvisor.Text)	Specify text color
setColWidth (ModelAdvisor.Table)	Specify column widths
setEntry (ModelAdvisor.Table)	Add cell to table
setEntryAlign (ModelAdvisor.Table)	Specify table cell alignment
setHeading (ModelAdvisor.Table)	Specify table title
setHeadingAlign (ModelAdvisor.Table)	Specify table title alignment
setHyperlink (ModelAdvisor.Image)	Specify hyperlink location
setHyperlink (ModelAdvisor.Text)	Specify hyperlinked text
setImageSource (ModelAdvisor.Image)	Specify image location
setItalic (ModelAdvisor.Text)	Italicize text

setRetainSpaceReturn (ModelAdvisor.Text)	Retain spacing and returns in text
setRowHeading (ModelAdvisor.Table)	Specify table row title
setRowHeadingAlign (ModelAdvisor.Table)	Specify table row title alignment
setSubscript (ModelAdvisor.Text)	Specify subscripted text
setSuperscript (ModelAdvisor.Text)	Specify superscripted text
setType (ModelAdvisor.List)	Specify list type
setUnderlined (ModelAdvisor.Text)	Underline text

8

Class Reference

Requirements Management Interface

Model Coverage

cv.cvdatagroup	Collection of `cvdata` objects
cv.cvtestgroup	Collection of `cvtest` objects

Model Advisor Customization API

ModelAdvisor.Action	Add actions to custom checks
ModelAdvisor.Check	Create custom checks
ModelAdvisor.FactoryGroup	Define subfolder in **By Task** folder
ModelAdvisor.Group	Define custom folder
ModelAdvisor.InputParameter	Add input parameters to custom checks
ModelAdvisor.ListViewParameter	Add list view parameters to custom checks
ModelAdvisor.Root	Identify root node
ModelAdvisor.Task	Define custom tasks

Model Advisor Result Template API

ModelAdvisor.FormatTemplate	Template for formatting Model Advisor analysis results

Model Advisor Formatting API

ModelAdvisor.Image	Include image in Model Advisor output
ModelAdvisor.LineBreak	Insert line break
ModelAdvisor.List	Create list class
ModelAdvisor.Paragraph	Create and format paragraph
ModelAdvisor.Table	Create table
ModelAdvisor.Text	Create Model Advisor text output

9

Alphabetical List

cv.cvtestgroup.add

Purpose

Add `cvtest` objects

Syntax

```
add(cvtg, cvto1, cvto2, ...)
```

Description

`add(cvtg, cvto1, cvto2, ...)` adds the `cvtest` objects specified by the strings `cvto1`, `cvto2`, etc. to `cvtg`, which is an instantiation of the `cv.cvtestgroup` class.

Example

Create two `cvtest` objects and add them to a newly created `cv.cvtestgroup` object:

```
cvto1 = cvtest;
cvto2 = cvtest;
cvtg = cv.cvtestgroup;
add(cvtg, cvto1, cvto2);
```

Purpose

Add check to folder

Syntax

```
addCheck(fg_obj, check_ID)
```

Description

`addCheck(fg_obj, check_ID)` adds checks, identified by `check_ID`, to the folder specified by `fg_obj`, which is an instantiation of the `ModelAdvisor.FactoryGroup` class.

Examples

Add three checks to `rec`:

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
.
.
.
addCheck(rec, 'com.mathworks.sample.Check1');
addCheck(rec, 'com.mathworks.sample.Check2');
addCheck(rec, 'com.mathworks.sample.Check3');
```

ModelAdvisor.Group.addGroup

Purpose

Add subfolder to folder

Syntax

```
addGroup(group_obj, child_obj)
```

Description

`addGroup(group_obj, child_obj)` adds a new subfolder, identified by `child_obj`, to the folder specified by `group_obj`, which is an instantiation of the `ModelAdvisor.Group` class.

Examples

Add three checks to `rec`:

```
group_obj = ModelAdvisor.Group('com.mathworks.sample.group');
.
.
.
addGroup(group_obj, 'com.mathworks.sample.subgroup1');
addGroup(group_obj, 'com.mathworks.sample.subgroup2');
addGroup(group_obj, 'com.mathworks.sample.subgroup3');
```

Purpose

Add item to list

Syntax

`addItem(element)`

Description

`addItem(element)` adds items to the list created by the `ModelAdvisor.List` constructor.

Inputs

element	Specifies an element to be added to a list in one of the following: • Element • Cell array of elements. When you add a cell array to a list, they form different rows in the list. • String

Example

```
subList = ModelAdvisor.List();
setType(subList, 'numbered')
addItem(subList, ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
addItem(subList, ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

ModelAdvisor.Paragraph.addItem

Purpose

Add item to paragraph

Syntax

```
addItem(text, element)
```

Description

`addItem(text, element)` adds an element to `text`. `element` is one of the following:

- String
- Element
- Cell array of elements

Example

Add two lines of text:

```
result = ModelAdvisor.Paragraph;
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

Purpose

Add row to table

Syntax

`addRow(ft_obj, {item1, item2, ..., itemn})`

Description

`addRow(ft_obj, {item1, item2, ..., itemn})` is an optional method that adds a row to the end of a table in the result. *`ft_obj`* is a handle to the template object previously created. `{`*`item1`*`, `*`item2`*`, ..., `*`itemn`*`}` is a cell array of strings and objects to add to the table. The order of the items in the array determines which column the item is in. If you do not add data to the table, the Model Advisor does not display the table in the result.

Note Before adding rows to a table, you must specify column titles using the `setColTitle` method.

Examples

Find all of the blocks in the model and create a table of the blocks:

```
% Create FormatTemplate object, specify table format
ft = ModelAdvisor.FormatTemplate('TableTemplate');

% Add information to the table
setTableTitle(ft, {'Blocks in Model'});
setColTitles(ft, {'Index', 'Block Name'});
% Find all the blocks in the system and add them to a table.
allBlocks = find_system(system);
for inx = 2 : length(allBlocks)
    % Add information to the table
    addRow(ft, {inx-1,allBlocks(inx)});
end
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.Group.addTask

Purpose

Add task to folder

Syntax

```
addTask(group_obj, task_obj)
```

Description

`addTask(group_obj, task_obj)` adds a task, specified by `task_obj`, to the folder `group_obj`.`group_obj` is an instantiation of the `ModelAdvisor.Group` class.

Example

Add three tasks to `MAG`.

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
addTask(MAG, MAT1);
addTask(MAG, MAT2);
addTask(MAG, MAT3);
```

Purpose

Get names of all models associated with `cvdata` objects in `cv.cvdatagroup`

Syntax

```
models = allNames(cvdg)
```

Description

`models = allNames(cvdg)` returns a cell array of strings identifying all model names associated with the `cvdata` objects in `cvdg`, an instantiation of the `cv.cvdatagroup` class.

Examples

Add three `cvdata` objects to `cvdg` and return a cell array of model names:

```
a = cvdata;
b = cvdata;
c = cvdata;
cvdg = cv.cvdatagroup;
add (cvdg, a, b, c);
model_names = allNames(cvdg)
```

cv.cvtestgroup.allNames

Purpose

Get names of all models associated with `cvtest` objects in `cvtestgroup`

Syntax

```
models = allNames(cvtg)
```

Description

`models = allNames(cvtg)` returns a cell array of strings identifying all model names associated with the `cvtest` objects in `cvtg`, an instantiation of the `cv.cvtestgroup` class.

Examples

Add three `cvtest` objects to `cvtg` and return a cell array of model names:

```
d = cvtest;
e = cvtest;
f = cvtes;
cvtg = cv.cvtestgroup;
add (cvtg, d, e, f);
model_names = allNames(cvtg)
```

Purpose

Display condition coverage information for model object

Syntax

```
coverage = conditioninfo(cvdo, object)
coverage = conditioninfo(cvdo, object, ignore_descendants)
[coverage, description] = conditioninfo(cvdo, object)
```

Description

`coverage = conditioninfo(cvdo, object)` returns condition coverage results from the `cvdata` object `cvdo` for the model component specified by `object`.

`coverage = conditioninfo(cvdo, object, ignore_descendants)` returns condition coverage results for `object`, depending on the value of `ignore_descendants`.

`[coverage, description] = conditioninfo(cvdo, object)` returns condition coverage results and textual descriptions of each condition in `object`.

Inputs

`cvdo`	`cvdata` object
`object`	The `object` argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for `object` include: • `BlockPath` — Full path to a Simulink model or block • `BlockHandle` — Handle to a Simulink model or block • `slObj` — Handle to a Simulink API object • `sfID` — Stateflow ID • `sfObj` — Handle to a Stateflow API object

- `{BlockPath, sfID}` — Cell array with the path to a Stateflow block and the ID of an object contained in that chart
- `{BlockPath, sfObj}` — Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
- `[BlockHandle, sfID]` — Array with a Stateflow block handle and the ID of an object contained in that chart

`ignore_descendants` Specifies to ignore the coverage of descendant objects if `ignore_descendants` is set to `1`.

Outputs

`coverage` The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are:

- `covered_outcomes` — Number of condition outcomes satisfied for `object`
- `total_outcomes` — Total number of condition outcomes for `object`

Note `coverage` is empty if `cvdo` does not contain condition coverage results for `object`.

`description` `description` is a structure array containing the following fields:

- `text` — String describing a condition or the block port to which it applies
- `trueCnts` — Number of times the condition was true in a simulation

- `falseCnts` — Number of times the condition was false in a simulation

Examples

The following commands open the `slvnvdemo_cv_small_controller` demo model, create the test specification object `testObj`, enable condition coverage for `testObj`, and execute `testObj`:

```
mdl = 'slvnvdemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.condition = 1;
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the condition coverage results for the Logic block (in the Gain subsystem) and determine its percentage of condition outcomes covered:

```
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');
cov = conditioninfo(data, blk_handle)
percent_cov = 100 * cov(1) / cov(2)
```

Alternatives

In the Coverage Settings dialog box, on the **Coverage** tab, under **Coverage Metrics**, select **Condition Coverage**.

See Also

- "Condition Coverage (CC)" on page 5-3
- `decisioninfo`
- `mcdcinfo`

cv.cvdatagroup class

Purpose

Collection of `cvdata` objects

Description

Instances of this class contain a collection of `cvdata` objects. For more information, see “Extracting Results from cv.cvdatagroup” on page 5-73.

Construction

cv.cvdatagroup	Create collection of `cvdata` objects for model reference hierarchy

Methods

allNames	Get names of all models associated with `cvdata` objects in `cv.cvdatagroup`
get	Get `cvdata` object
getAll	Get all `cvdata` objects

Properties

name	`cv.cvdatagroup` object name

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Purpose

Create collection of `cvdata` objects for model reference hierarchy

Syntax

```
cvdg = cv.cvdatagroup(cvdo1, cvdo2,...)
```

Description

`cvdg = cv.cvdatagroup(cvdo1, cvdo2,...)` creates an instantiation of the `cv.cvdatagroup` class (`cvdg`) that contains the `cvdata` objects `cvdo1`, `cvdo2`, etc. A `cvdata` object contains results of the simulation runs.

Examples

Create an instantiation of the `cv.cvdatagroup` class and add two `cvdata` objects to it:

```
a = cvdata;
b = cvdata;
cvdg = cv.cvdatagroup(a, b);
```

cv.cvtestgroup class

Purpose

Collection of `cvtest` objects

Description

Instances of this class contain a collection of `cvtest` objects. For more information, see “Creating a Test Group with cv.cvtestgroup” on page 5-72.

Construction

cv.cvtestgroup	Create collection of `cvtest` objects for model reference hierarchy

Methods

add	Add `cvtest` objects
allNames	Get names of all models associated with `cvtest` objects in `cvtestgroup`
get	Get `cvtest` objects

Properties

name	`cv.cvtestgroup` object name

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Purpose

Create collection of `cvtest` objects for model reference hierarchy

Syntax

```
cvtg = cv.cvtestgroup(cvto1, cvto2,...)
```

Description

`cvtg = cv.cvtestgroup(cvto1, cvto2,...)` creates an instantiation of the `cv.cvtestgroup` class (`cvtg`) that contains the `cvtest` objects `cvto1`, `cvto2`, etc. A `cvtest` object is a test specification object for a Simulink model.

Examples

Create an instantiation of the `cv.cvtestgroup` class and add two `cvtest` objects to it:

```
a = cvtest;
b = cvtest;
cvtg = cv.cvtestgroup(a, b);
```

See Also

`cvtest`

cvexit

Purpose

Exit model coverage environment

Syntax

```
cvexit
```

Description

`cvexit` exits the model coverage environment. Issuing this command closes the Coverage Display Window and removes coloring from a block diagram that displays its model coverage results.

Purpose

Produce HTML report from model coverage objects in memory

Syntax

```
cvhtml(file, cvdo)
cvhtml(file, cvdo1, cvdo2, ...)
cvhtml(file, cvdo1, cvdo2, ..., options)
cvhtml(file, cvdo1, cvdo2, ..., options, detail)
```

Description

Use the `cvhtml` command to produce an HTML report from `cv.cvdatagroup` or `cvdata` objects that you produce when you run a model coverage test in simulation.

Note The model must be open when using the `cvhtml` command to generate its coverage report.

`cvhtml(file, cvdo)` creates an HTML report of the coverage results in the `cvdata` or `cv.cvdatagroup` object `cvdo`, which is written to `file` .

`cvhtml(file, cvdo1, cvdo2, ...)` creates a combined report of several `cvdata` objects. The results from each object are displayed in a separate column of the HTML report. Each `cvdata` object must correspond to the same root model or subsystem, or the function produces errors.

`cvhtml(file, cvdo1, cvdo2, ..., options)` creates a combined report of several `cvdata` objects using the report options specified by the `options` string.

`cvhtml(file, cvdo1, cvdo2, ..., options, detail)` creates a combined report of several `cvdata` objects and specifies the detail level of the report with the value of `detail`.

Inputs

`file`	`file` is the name of the HTML file in the MATLAB current directory where `cvhtml` stores the results.
`cvdo`	`cvdo` is a `cv.cvdatagroup` object.

`options`

The following table summarizes the report options that you specify in the `options` string. To enable an option, set it equal to 1 (e.g., `'-hTR=1'`); to disable an option, set it equal to 0 (e.g., `'-bRG=0'`). To specify multiple report options, list individual options in a single `options` string separated by commas or spaces (e.g., `'-hTR=1 -bRG=0 -scm=0'`).

Option	Description	Default Setting
`-aTS`	Include each test in the model summary	`on`
`-bRG`	Produce bar graphs in the model summary	`on`
`-bTC`	Use two color bar graphs (red, blue)	`off`
`-hTR`	Display hit/count ratio in the model summary	`off`
`-nFC`	Do not report fully covered model objects	`off`
`-scm`	Include cyclomatic complexity numbers in summary	`on`
`-bcm`	Include cyclomatic complexity numbers in block details	`on`

`detail`

`detail` specifies the level of detail in the report. Set `detail` to an integer between 0 and 3. Greater numbers for `detail` indicate greater detail. The default value is 2.

Examples

Create a cumulative coverage report:

```
covdata1 = cvsim(test1);
covdata2 = cvsim(test2);
cvhtml('cumulative_report', covdata + covdata2);
```

Alternatives

You can create an HTML model coverage report if you enable the **Generate HTML report** option on the **Report** tab of the Coverage Settings dialog box.

See Also

- Chapter 5, "Using Model Coverage"

cvload

Purpose

Load coverage tests and stored results into memory

Syntax

```
[cvtos, cvdos] = cvload(filename)
[cvtos, cvdos] = cvload(filename, restoretotal)
```

Description

The `cvload` command loads into memory the coverage tests and results stored in a file by the `cvsave` command.

`[cvtos, cvdos] = cvload(filename)` loads the tests and data stored in the text file `filename.cvt`.

`[cvtos, cvdos] = cvload(filename, restoretotal)` restores or clears the cumulative results from prior runs, depending on the value of `restoretotal`.

cvload Special Considerations

The following are special considerations for using the `cvload` command:

- If a model with the same name exists in the coverage database, only the compatible results that reference the existing model are loaded to prevent duplication.
- If the Simulink models referenced from the file are open but do not exist in the coverage database, the coverage tool resolves the links to the existing models.
- When you are loading several files that reference the same model, only the results that are consistent with the earlier files are loaded.

Inputs

`filename`	The name of the file (`filename.cvt`) containing the tests (in `cvtest` objects) and data (in `cvdata` objects).
`restoretotal`	If `restoretotal` is 1, `cvload` restores the cumulative results from prior runs. If `restoretotal` is unspecified or 0, `cvload` clears the model's cumulative results.

Outputs

`cvtos`	The `cvtest` objects that are successfully loaded are returned in `cvtos`, a cell array of `cvtest` objects.
`cvdos`	The `cvdata` objects that are successfully loaded are returned in `cvdos`, a cell array of `cvdata` objects. `cvdos` has the same size as `ctvos`, but can contain empty elements if a particular test has no results.

Examples

Store coverage results:

```
(test_objects, data_objects) = cvload(test_results, 1);
```

See Also

- "Loading Stored Coverage Test Results with cvload" on page 5-66
- `cvsave`

cvmodelview

Purpose

Display model coverage results with model coloring

Syntax

```
cvmodelview(cvdo)
```

Description

`cvmodelview(cvdo)` displays coverage results from the `cvdata` object `cvdo` by coloring the Simulink model.

Inputs

`cvdo`	`cvdo` is a `cvdata` object.

Examples

Open the `slvnvdemo_cv_small_controller` demo model, create the test specification object `testObj`, and execute `testObj`:

```
mdl = 'slvnvdemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
data = cvsim(testObj)
```

Afterward, issue the following command to display the model coverage results by coloring the block diagram:

```
cvmodelview(data)
```

Alternatives

In the Coverage Settings dialog box:

1. Select **Coverage for this model** on the **Coverage** tab.
2. Select **Display coverage results using model coloring** on the **Results** tab.

See Also

- "Enabling the Colored Diagram Display" on page 5-57
- "Displaying Model Coverage with Model Coloring" on page 5-58

Purpose

Save coverage tests and results to file

Syntax

```
cvsave(filename, model)
cvsave(filename, cvto1, cvto2, ...)
cvsave(filename, cvdo1, cvdo2, ...)
```

Description

Save the coverage tests and results from simulations to a file with the function `cvsave`.

`cvsave(filename, model)` saves all the tests (`cvtest` objects) and results (`cvdata` objects) related to the model `model` in the text file `filename.cvt`.

`cvsave(filename, cvto1, cvto2, ...)` saves the tests in the `cvtest` objects `cvto1, cvto2, ...` in the text file `filename.cvt`. Information about the referenced models is also saved.

`cvsave(filename, cvdo1, cvdo2, ...)` saves the tests, test results, and referenced models' structure for `cvdata` objects `cvdo1`, `cvdo2`, ... to the text file `filename.cvt`.

Inputs

`filename`	Name of the text file (to be appended with the `.cvt` extension) in which to store the tests and results
`model`	Name of a Simulink model
`cvto`	`cvtest` object
`cvdo`	`cvdata` object

Examples

Use `cvhtml` to display the coverage results for two tests and the cumulative coverage, compute cumulative coverage with the + operator, and save the results:

```
cvhtml('ratelim_report',dataObj1,dataObj2);
cumulative = dataObj1+dataObj2;
cvsave('ratelim_testdata',cumulative);
```

Alternatives

In the Coverage Settings dialog box, on the Results tab:

- Select **Save cumulative results in workspace variable**.
- Select **Save last run in workspace variable**.

See Also

- "Saving Test Runs to a File with cvsave" on page 5-66
- `cvload`

Purpose

Simulate and return model coverage results for test objects

Syntax

```
cvdo = cvsim(cvto)
[cvdo,t,x,y] = cvsim(cvto)
[cvdo,t,x,y] = cvsim(cvto, timespan, options)
[cvdo,t,x,y] = cvsim(cvto, label, setupcmd)
[cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)
```

Description

You simulate a test specification object (a `cvtest` object) with the `cvsim` command.

Note You do not have to enable model coverage reporting for the model to use the `cvsim` command.

`cvdo = cvsim(cvto)` executes the `cvtest` object `cvto` by starting a simulation run for the corresponding model. The results are returned in the `cvdata` object `cvdo`. But when recording coverage for multiple models in a hierarchy, `cvsim` returns its results in a `cv.cvdatagroup` object.

`[cvdo,t,x,y] = cvsim(cvto)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation.

`[cvdo,t,x,y] = cvsim(cvto, timespan, options)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation, and overrides default simulation values with the values for `timespan` and `options`.

`[cvdo,t,x,y] = cvsim(cvto, label, setupcmd)` creates the `cvtest` object `cvto` and simulates it in one command. The arguments `label` and `setupcmd` are passed directly to the `cvtest` function, which creates the `cvtest` object `cvto`.

`[cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)` executes the `cvtest` objects `cvto1, cvto2, ...` and returns the results in the set of `cvdata` objects `cvdo1, cvdo2, ....`

Inputs

`cvto`	`cvtest` object
`timespan`	Simulation start and stop time. Specify as: • `tFinal` to specify the stop time. The start time is 0. • `[tStart tFinal]` to specify the start and stop times. • `[tStart OutputTimes tFinal]` to specify the start and stop times and time points to be returned in t. Generally, `t` includes more time points. `OutputTimes` is equivalent to specifying **Configuration Parameters > Data Import/Export > Output options > Produce specified output only**.
`options`	Optional simulation parameters specified as a structure created by the `simset` command.
`label`	Label for test object (passed to `cvtest`)
`setupcmd`	Setup command used to create test object (passed to `cvtest`)

Outputs

`cvdo`	`cvdata` object
`t`	The simulation's time vector

`x`	The simulation's state matrix consisting of continuous states followed by discrete states
`y`	The simulation's output matrix. Each column contains the output of a root-level Outport block, in port number order. If any Outport block has a vector input, its output takes the appropriate number of columns.

Examples

```
[dataObj1,T,X,Y] = cvsim(testObj1,[0 2]);
[dataObj2,T,X,Y] = cvsim(testObj2,[0 2]);
```

See Also

- "Creating and Running Test Cases" on page 5-8
- `simset`
- `cvtest`

cvsimref

Purpose

Simulate and return model coverage results for referenced models

Syntax

```
cvdg = cvsimref(topModelName)
cvdg = cvsimref(topModelName, cvtg)
[cvdg,t,x,y] = cvsimref(topModelName, cvtg)
[cvdg,t,x,y] = cvsimref(topModelName, cvtg, timespan,
   options)
[cvdg1, cvdg2, ...] = cvsimref(topModelName, cvtg1, cvtg2,
   ...)
```

Description

Use the `cvsimref` function to record coverage for referenced models in a hierarchy.

Note You do not have to enable model coverage reporting for any of the models in a model hierarchy to use the `cvsimref` command.

`cvdg = cvsimref(topModelName)` simulates the top model in the hierarchy, collects model coverage data, and returns the results in the `cv.cvdatagroup` object `cvdg`.

`cvdg = cvsimref(topModelName, cvtg)` executes the `cv.cvtestgroup` object `cvtg` by starting a simulation run for the corresponding top model, `topModelName`. The results are returned in `cvdg`.

`[cvdg,t,x,y] = cvsimref(topModelName, cvtg)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation.

`[cvdg,t,x,y] = cvsimref(topModelName, cvtg, timespan, options)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation, and overrides default simulation values with the values for `timespan` and `options`.

`[cvdg1, cvdg2, ...] = cvsimref(topModelName, cvtg1, cvtg2, ...)` executes the `cv.cvtestgroup` objects `cvtg1, cvtg2, ...` and returns the results in the set of `cv.cvdatagroup` objects `cdvg1, cvdg2, ...`.

Inputs

`topModelName`	Name of the top model in the hierarchy
`cvtg`	`cv.cvtestgroup` object
`timespan`	Simulation start and stop time. Specify as: • `tFinal` to specify the stop time. The start time is 0. • `[tStart tFinal]` to specify the start and stop times. • `[tStart OutputTimes tFinal]` to specify the start and stop times and time points to be returned in t. Generally, `t` includes more time points. `OutputTimes` is equivalent to specifying **Configuration Parameters > Data Import/Export > Output options > Produce specified output only**.
`options`	Optional simulation parameters specified as a structure created by the `simset` command.

Outputs

`cvdg`	`cv.cvdatagroup` object
`t`	The simulation's time vector
`x`	The simulation's state matrix consisting of continuous states followed by discrete states
`y`	The simulation's output matrix. Each column contains the output of a root-level Outport block, in port number order. If any Outport block has a vector input, its output takes the appropriate number of columns.

See Also

- "Using Model Coverage Commands for Referenced Models" on page 5-69
- "Creating and Running Test Cases" on page 5-8
- `simset`
- `cvsim`
- `cvtest`
- `cv.cvdatagroup`
- `cv.cvtestgroup`

Purpose

Create model coverage test specification object

Syntax

```
cvto = cvtest(root)
cvto = cvtest(root, label)
cvto = cvtest(root, label, setupcmd)
```

Description

The `cvtest` command creates a test specification object that you simulate with the `cvsim` command.

`cvto = cvtest(root)` creates a test object with the handle `cvto`.

`cvto = cvtest(root, label)` creates a test object with the label `label`, which is used for reporting results.

`cvto = cvtest(root, label, setupcmd)` creates a test object with the setup command `setupcmd` and labels it with `label`.

Inputs

`root`	The name of, or a handle to, a Simulink model or a subsystem of a model. Only the specified model or subsystem and its descendants are subject to model coverage testing.
`label`	Label for test object.
`setupcmd`	Setup command for creating test object. The setup command is executed in the base MATLAB workspace just prior to running the instrumented simulation. This command is useful for loading data prior to a test.

Outputs

`cvto` A test object with the following structure.

Field	Description
`id`	Read-only internal ID
`modelcov`	Read-only internal ID
`rootPath`	Name of system or subsystem for analysis
`label`	String used when reporting results
`setupCmd`	Command executed in base workspace prior to simulation
`settings.condition`	Set to 1 for condition coverage.
`settings.decision`	Set to 1 for decision coverage.
`settings.mcdc`	Set to 1 for MC/DC coverage.
`settings.sigrange`	Set to 1 for signal range coverage.
`settings.tableExec`	Set to 1 for lookup table coverage.
`modelRefSettings.enable`	• `off` — Disable coverage for all referenced models. • `all` or `on` — Enable coverage for all referenced models. • `filtered` — Enable coverage only for referenced models not listed in the `excludedModels` subfield.
`modelRefSettings.excludeTopModel`	Set to 1 to exclude coverage for the top model.

`modelRefSettings.excludedModels`	String specifying a comma-separated list of referenced models for which coverage is disabled.
`emlSettings.enableExternal`	Set to 1 to enable coverage for external M-files called by Embedded MATLAB functions in your model.

Examples

Create a `cvtest` object of the Adjustable Rate Limiter block in the demo model `slvnvdemo_ratelim_harness`:

```
mdl = 'slvnvdemo_ratelim_harness';
testObj1 = cvtest([mdl, '/Adjustable Rate Limiter']);
testObj1.label = 'Gain within slew limits';
testObj1.setupCmd = 'load(''within_lim.mat'');';
testObj1.settings.mcdc = 1;
```

See Also

- "Creating Tests with cvtest" on page 5-62
- `cv.cvtestgroup`

decisioninfo

Purpose

Display decision coverage information for model object

Syntax

```
coverage = decisioninfo(cvdo, object)
coverage = decisioninfo(cvdo, object, ignore_descendants)
[coverage, description] = decisioninfo(cvdo, object)
```

Description

`coverage = decisioninfo(cvdo, object)` returns decision coverage results from the `cvdata` object `cvdo` for the model component specified by `object`.

`coverage = decisioninfo(cvdo, object, ignore_descendants)` returns decision coverage results for `object`, depending on the value of `ignore_descendants`.

`[coverage, description] = decisioninfo(cvdo, object)` returns decision coverage results and textual descriptions of decision points associated with `object`.

Inputs

`cvdo`

`cvdata` object

`object`

The `object` argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for `object` include the following:

Object Specification	Description
`BlockPath`	Full path to a Simulink model or block
`BlockHandle`	Handle to a Simulink model or block
`slObj`	Handle to a Simulink API object
`sfID`	Stateflow ID

`sfObj`	Handle to a Stateflow API object
`{BlockPath, sfID}`	Cell array with the path to a Stateflow block and the ID of an object contained in that chart
`{BlockPath, sfObj}`	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
`[BlockHandle, sfID]`	Array with a Stateflow block handle and the ID of an object contained in that chart

`ignore_descendants` Specifies to ignore the coverage of descendant objects if `ignore_descendants` is set to `1`.

Outputs

`coverage` The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — the number of decision outcomes satisfied for `object`
- `total_outcomes` — the total number of decision outcomes for `object`

Note `coverage` is empty if `cvdo` does not contain decision coverage results for `object`.

`description` — `description` is a structure array containing the following fields:

- `decision.text` — String describing a decision point, e.g., `'U > LL'`
- `decision.outcome.text` — String describing a decision outcome, i.e., `'true'` or `'false'`
- `decision.outcome.executionCount` — Number of times a decision outcome occurred in a simulation

Examples

Open the `slvnvdemo_cv_small_controller` demo model, create the test specification object `testObj`, enable decision coverage for `testObj`, and execute `testObj`:

```
mdl = 'slvnvdemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.decision = 1;
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the decision coverage results for the Saturation block and determine its percentage of decision outcomes covered:

```
blk_handle = get_param([mdl, '/Saturation'], 'Handle');
cov = decisioninfo(data, blk_handle)
percent_cov = 100 * cov(1) / cov(2)
```

Alternatives

In the Coverage Settings dialog box, on the **Coverage** tab, under **Coverage Metrics**, select **Decision Coverage**.

See Also

- "Condition Coverage (CC)" on page 5-3
- `conditioninfo`
- `mcdcinfo`

cv.cvdatagroup.get

Purpose

Get `cvdata` object

Syntax

```
get(cvdg, model_name)
```

Description

`get(cvdg, model_name)` returns the `cvdata` object in the `cv.cvdatagroup` object `cvdg` that corresponds to the model specified in `model_name`.

Example

Get a `cvdata` object from the specified Simulink model:

```
get(cvdg, 'slvnvdemo_cv_small_controller');
```

Purpose

Get `cvtest` objects

Syntax

```
get(cvtg, model_name)
```

Description

`get(cvtg, model_name)` returns the `cvtest` object in the `cv.cvtestgroup` object `cvtg` that corresponds to the model specified in `model_name`.

Example

Get a `cvtest` object from the specified Simulink model:

```
get(cvtg, 'slvnvdemo_cv_small_controller');
```

See Also

`cvsimref`, `cvtest`

cv.cvdatagroup.getAll

Purpose

Get all `cvdata` objects

Syntax

```
getAll(cvdo)
```

Description

`getAll(cvdo)` returns all `cvdata` objects in the `cv.cvdatagroup` object `cvdo`.

Example

Return all `cvdata` object from the specified Simulink model:

```
getAll(cvdg, 'slvnvdemo_cv_small_controller');
```

Purpose

Get table cell contents

Syntax

```
content = getEntry(table, row, column)
```

Description

`content = getEntry(table, row, column)` gets the contents of the specified cell.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying the row
`column`	An integer specifying the column

Outputs

`content`	An element object or object array specifying the content of the table entry

Example

Get the content of the table cell in the third column, third row:

```
table1 = ModelAdvisor.Table(4, 4);
.
.
.
content = getEntry(table1, 3, 3);
```

ModelAdvisor.Check.getID

Purpose

Return check identifier

Syntax

```
id = getID(check_obj)
```

Description

`id = getID(check_obj)` returns the ID of the check `check_obj`. `id` is a unique string that identifies the check.

You create this unique identifier when you create the check. This unique identifier is the equivalent of the `ModelAdvisor.Check ID` property.

See Also

"Defining Custom Checks" on page 6-13 — Describes how to create custom actions

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Display modified condition/decision coverage information for model object

Syntax

```
coverage = mcdcinfo(cvdo, object)
coverage = mcdcinfo(cvdo, object, ignore_descendants)
[coverage, description] = mcdcinfo(cvdo, object)
```

Description

`coverage = mcdcinfo(cvdo, object)` returns modified condition/decision coverage results from the `cvdata` object `cvdo` for the model component specified by `object`.

`coverage = mcdcinfo(cvdo, object, ignore_descendants)` returns modified condition/decision coverage results for `object`, depending on the value of `ignore_descendants`.

`[coverage, description] = mcdcinfo(cvdo, object)` returns modified condition/decision coverage results and textual descriptions of each condition/decision in `object`.

Inputs

`cvdo` — `cvdata` object

`object` — The `object` argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for `object` include the following:

Object Specification	Description
`BlockPath`	Full path to a Simulink model or block
`BlockHandle`	Handle to a Simulink model or block
`slObj`	Handle to a Simulink API object
`sfID`	Stateflow ID

`sfObj`	Handle to a Stateflow API object
`{BlockPath, sfID}`	Cell array with the path to a Stateflow block and the ID of an object contained in that chart
`{BlockPath, sfObj}`	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
`[BlockHandle, sfID]`	Array with a Stateflow block handle and the ID of an object contained in that chart

`ignore_descendants` — Specifies to ignore the coverage of descendant objects if `ignore_descendants` is set to `1`.

Outputs

`coverage` — The value of `coverage` is a two-element vector of the form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — Number of condition/decision outcomes satisfied for `object`
- `total_outcomes` — Total number of condition/decision outcomes for `object`

Note `coverage` is empty if `cvdo` does not contain modified condition/decision coverage results for `object`.

`description`

`description` is a structure array containing the following fields:

- `text` — String denoting whether the condition/decision is associated with a block output or Stateflow transition
- `condition.text` — String describing a condition/decision or the block port to which it applies
- `condition.achieved` — Logical array indicating whether a condition case has been fully covered
- `condition.trueRslt` — String representing a condition case expression that produces a true result
- `condition.falseRslt` — String representing a condition case expression that produces a false result

See "MC/DC Analysis" on page 5-36 for more information about the data contained in these fields.

Examples

The following commands open the `slvnvdemo_cv_small_controller` demo model, create the test specification object `testObj`, enable modified condition/decision coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvnvdemo_cv_small_controller';
```

```
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.mcdc = 1;
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the modified condition/decision coverage results for the Logic block (in the Gain subsystem) and determine its percentage of condition/decision outcomes covered.

```
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');
cov = mcdcinfo(data, blk_handle)
percent_cov = 100 * cov(1) / cov(2)
```

Alternatives

In the Coverage Settings dialog box, on the **Coverage** tab, under **Coverage Metrics**, select **MCDC Coverage**.

See Also

- "Modified Condition/Decision Coverage (MC/DC)" on page 5-4
- `conditioninfo`
- `mcdcinfo`

Purpose

Add actions to custom checks

Description

Instances of this class define actions you take when the Model Advisor checks do not pass. Users access actions by clicking the **Action** button that you define in the Model Advisor window.

Construction

ModelAdvisor.Action	Add actions to custom checks

Methods

setCallbackFcn	Specify action callback function

Properties

Description	Message in **Action** box
Name	Action button label

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Example

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
myAction.Name='Fix block fonts';
myAction.Description=...
   'Click the button to update all blocks with specified font';
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

ModelAdvisor.Action

Purpose

Add actions to custom checks

Syntax

```
action_obj = ModelAdvisor.Action
```

Description

`action_obj = ModelAdvisor.Action` creates a handle to an action object.

Note

- Include an action definition in a check definition.
- Each check can contain only one action.

Example

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Create custom checks

Description

The `ModelAdvisor.Check` class creates a Model Advisor check object. All checks must have an associated `ModelAdvisor.Task` object to be displayed in the Model Advisor tree.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** is displayed in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When you use checks in task definitions, the following rules apply:

- If you define the properties of the check in the check definition and the task definition, the task definition takes precedence. The Model Advisor displays the information contained in the task definition. For example, if you define the name of the check in the task definition using the `ModelAdvisor.Task.DisplayName` property and in the check definition using the `ModelAdvisor.Check.Title` property, the Model Advisor displays the information provided in `ModelAdvisor.Task.DisplayName`.
- If you define the properties of the check in the check definition but not the task definition, the task uses the properties from the check. For example, if you define the name of the check in the check definition using the `ModelAdvisor.Check.Title` property, and you register the check using a task definition, the Model Advisor displays the information provided in `ModelAdvisor.Check.Title`.
- If you define the properties of the check in the task definition but not the check definition, the Model Advisor displays the information correctly as long as you register the task with the Model Advisor instead of the check. For example, if you define the name of the check in the task definition using the `ModelAdvisor.Task.DisplayName` property instead of the `ModelAdvisor.Check.Title` property, and you register the check

using a task definition, the Model Advisor displays the information provided in `ModelAdvisor.Task.DisplayName`.

Construction

ModelAdvisor.Check	Create custom checks

Methods

getID	Return check identifier
setAction	Specify action for check
setCallbackFcn	Specify callback function for check
setInputParameters	Specify input parameters for check
setInputParametersLayoutGrid	Specify layout grid for input parameters

Properties

CallbackContext	Model or subsystem context
CallbackHandle	Callback function handle for check
CallbackStyle	Callback function type
Enable	Indicate whether user can enable or disable check
ID	Identifier for check
LicenseName	Cell array of product license names
ListViewVisible	Status of button
Result	Results cell array
Title	Name of check

TitleTips	Description of check
Value	Status of check
Visible	Indicate to display or hide check

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

ModelAdvisor.Check

Purpose

Create custom checks

Syntax

```
check_obj = ModelAdvisor.Check(check_ID)
```

Description

`check_obj = ModelAdvisor.Check(check_ID)` creates a check object, `check_obj`, and assigns it a unique identifier, `check_ID`. `check_ID` must remain constant. To display checks in the Model Advisor tree, all checks must have an associated `ModelAdvisor.Task` or `ModelAdvisor.Root` object.

Note You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution appears** in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

Example

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Define subfolder in **By Task** folder

Description

The `ModelAdvisor.FactoryGroup` class defines a new subfolder to add to the **By Task** folder.

Construction

ModelAdvisor.FactoryGroup	Define subfolder in **By Task** folder

Methods

addCheck	Add check to folder

Properties

Description	Description of folder
DisplayName	Name of folder
ID	Identifier for folder
MAObj	Model Advisor object

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Example

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

ModelAdvisor.FactoryGroup

Purpose

Define subfolder in **By Task** folder

Syntax

```
fg_obj = ModelAdvisor.FactoryGroup(fg_ID)
```

Description

`fg_obj = ModelAdvisor.FactoryGroup(fg_ID)` creates a handle to a factory group object, `fg_obj`, and assigns it a unique identifier, `fg_ID`. `fg_ID` must remain constant.

Example

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Template for formatting Model Advisor analysis results

Description

Use the `ModelAdvisor.FormatTemplate` class to format the result of a check in the analysis result pane of the Model Advisor for a uniform look and feel among the checks you create. There are two formats for the analysis result:

- Table
- List

Construction

ModelAdvisor.FormatTemplate	Construct template object for formatting Model Advisor analysis results

Methods

addRow	Add row to table
setCheckText	Add description of check to result
setColTitles	Add column titles to table
setInformation	Add description of subcheck to result
setListObj	Add list of hyperlinks to model objects
setRecAction	Add Recommended Action section and text
setRefLink	Add See Also section and links
setSubBar	Add line between subcheck results
setSubResultStatus	Add status to check or subcheck result
setSubResultStatusText	Add text below status in result

setSubTitle	Add title for subcheck in result
setTableInfo	Add data to table
setTableTitle	Add title to table

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

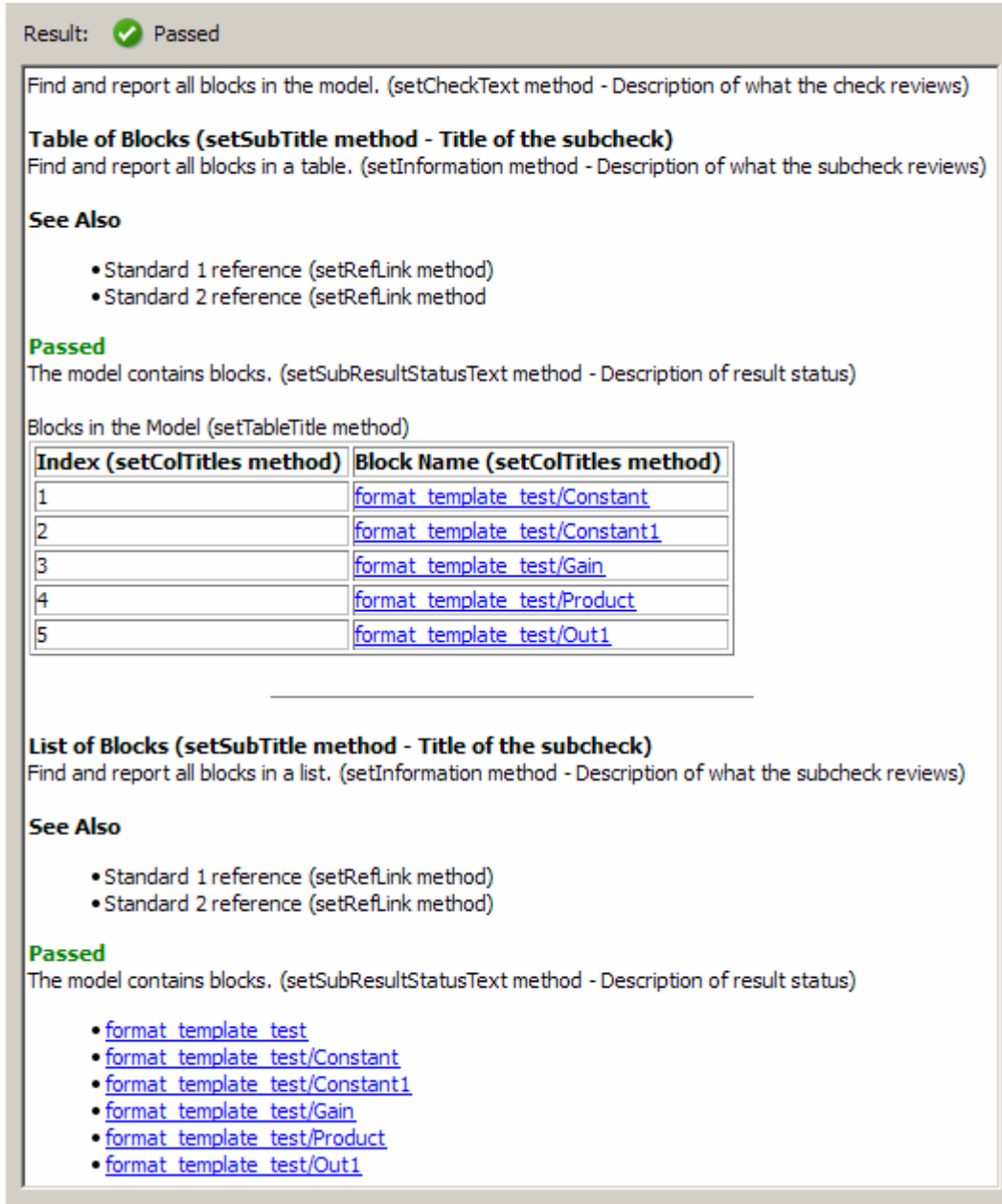
The following code creates two template objects, `ft1` and `ft2`, and uses them to format the result of running the check in a table and a list. The result identifies the blocks in the model. The graphics following the code display the output as it appears in the Model Advisor when the check passes and fails.

```
% Sample Check With Subchecks Callback Function
function ResultDescription = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

%Initialize variables
ResultDescription={};
ResultStatus = false; % Default check status is 'Warning'
mdladvObj.setCheckResultStatus(ResultStatus);

% Create FormatTemplate object for first subcheck, specify table format
ft1 = ModelAdvisor.FormatTemplate('TableTemplate');

% Add information describing the overall check
setCheckText(ft1, ['Find and report all blocks in the model. '...
    '(setCheckText method - Description of what the check reviews)']);

% Add information describing the subcheck
setSubTitle(ft1, 'Table of Blocks (setSubTitle method - Title of the subcheck)');
setInformation(ft1, ['Find and report all blocks in a table. '...
    '(setInformation method - Description of what the subcheck reviews)']);

% Add See Also section for references to standards
```

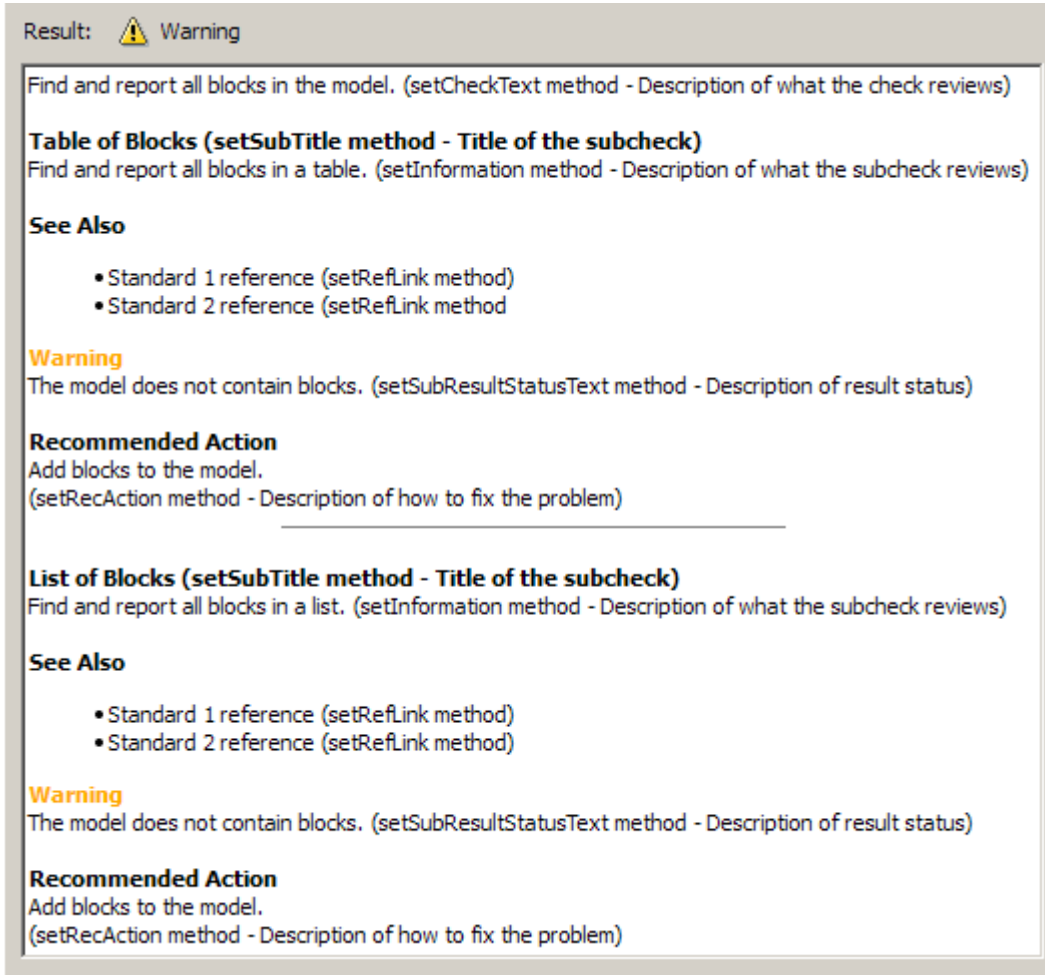
```
setRefLink(ft1, {{'Standard 1 reference (setRefLink method)'},
    {'Standard 2 reference (setRefLink method'}});

% Add information to the table
setTableTitle(ft1, {'Blocks in the Model (setTableTitle method)'});
setColTitles(ft1, {'Index (setColTitles method)',
    'Block Name (setColTitles method)'});

% Perform the check actions
allBlocks = find_system(system);
if length(find_system(system)) == 1
    % Add status for subcheck
    setSubResultStatus(ft1, 'Warn');
    setSubResultStatusText(ft1, ['The model does not contain blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    setRecAction(ft1, {'Add blocks to the model. '...
        '(setRecAction method - Description of how to fix the problem)'});
    ResultStatus = false;
else
    % Add status for subcheck
    setSubResultStatus(ft1, 'Pass');
    setSubResultStatusText(ft1, ['The model contains blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    for inx = 2 : length(allBlocks)
        % Add information to the table
        addRow(ft1, {inx-1,allBlocks(inx)});
    end
    ResultStatus = true;
end

% Pass table template object for subcheck to Model Advisor
ResultDescription{end+1} = ft1;

% Create FormatTemplate object for second subcheck, specify list format
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');

% Add information describing the subcheck
```

```
setSubTitle(ft2, 'List of Blocks (setSubTitle method - Title of the subcheck)');
setInformation(ft2, ['Find and report all blocks in a list. '...
    '(setInformation method - Description of what the subcheck reviews)']);

% Add See Also section for references to standards
setRefLink(ft2, {{'Standard 1 reference (setRefLink method)'},
    {'Standard 2 reference (setRefLink method)'}});

% Last subcheck, supress line
setSubBar(ft2, false);

% Perform the subcheck actions
if length(find_system(system)) == 1
    % Add status for subcheck
    setSubResultStatus(ft2, 'Warn');
    setSubResultStatusText(ft2, ['The model does not contain blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    setRecAction(ft2, {'Add blocks to the model. '...
        '(setRecAction method - Description of how to fix the problem)'});
    ResultStatus = false;
else
    % Add status for subcheck
    setSubResultStatus(ft2, 'Pass');
    setSubResultStatusText(ft2, ['The model contains blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    % Add information to the list
    setListObj(ft2, allBlocks);
end

% Pass list template object for the subcheck to Model Advisor
ResultDescription{end+1} = ft2;
% Set overall check status
mdladvObj.setCheckResultStatus(ResultStatus);
```

The following graphic displays the output as it appears in the Model Advisor when the check passes.



The following graphic displays the output as it appears in the Model Advisor when the check fails.



Alternatives

Use the Model Advisor Formatting API to format check analysis results, however The MathWorks recommends that you use the

`ModelAdvisor.FormatTemplate` class for a uniform look and feel among the checks you create.

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate

Purpose

Construct template object for formatting Model Advisor analysis results

Syntax

```
obj = ModelAdvisor.FormatTemplate('type')
```

Description

`obj = ModelAdvisor.FormatTemplate('type')` creates a handle, *obj*, to an object of the `ModelAdvisor.FormatTemplate` class. *type* is a string identifying the format type of the template, either list or table. Valid values are `ListTemplate` and `TableTemplate`.

You must return the result object to the Model Advisor to display the formatted result in the analysis result pane.

Note Use the `ModelAdvisor.FormatTemplate` class in check callbacks.

Examples

Create a template object, `ft`, and use it to create a list template:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Define custom folder

Description

The `ModelAdvisor.Group` class defines a folder that is displayed in the Model Advisor tree. Use folders to consolidate checks by functionality or usage.

Construction

ModelAdvisor.Group	Define custom folder

Methods

addGroup	Add subfolder to folder
addTask	Add task to folder

Properties

Description	Description of folder
DisplayName	Name of folder
ID	Identifier for folder
MAObj	Model Advisor object

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

ModelAdvisor.Group

Purpose

Define custom folder

Syntax

```
group_obj = ModelAdvisor.Group(group_ID)
```

Description

`group_obj = ModelAdvisor.Group(group_ID)` creates a handle to a group object, `group_obj`, and assigns it a unique identifier, `group_ID`. `group_ID` must remain constant.

Examples

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Include image in Model Advisor output

Description

The `ModelAdvisor.Image` class adds an image to the Model Advisor output.

Construction

ModelAdvisor.Image	Create custom checks

Methods

setHyperlink	Specify hyperlink location
setImageSource	Specify image location

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.Image

Purpose

Create custom checks

Syntax

```
object = ModelAdvisor.Image
```

Description

`object = ModelAdvisor.Image` creates a handle to an image object, `object`, that the Model Advisor displays in the output. The Model Advisor supports many image formats, including, but not limited to, JPEG, BMP, and GIF.

Examples

```
image_obj = ModelAdvisor.Image;
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Add input parameters to custom checks

Description

Instances of the `ModelAdvisor.InputParameter` class specify the input parameters a custom check uses in analyzing the model. Access input parameters in the Model Advisor window.

Construction

ModelAdvisor.InputParameter	Add input parameters to custom checks

Methods

setColSpan	Specify number of columns for input parameter
setRowSpan	Specify rows for input parameter

Properties

Description	Description of input parameter
Entries	Drop-down list entries
Name	Input parameter name
Type	Input parameter type
Value	Value of input parameter

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

ModelAdvisor.InputParameter

Purpose

Add input parameters to custom checks

Syntax

```
input_param = ModelAdvisor.InputParameter
```

Description

`input_param = ModelAdvisor.InputParameter` creates a handle to an input parameter object, `input_param`.

Note You must include input parameter definitions in a check definition.

Example

Note The following example is a fragment of code from the `sl_customization.m` file for the demo model, `slvnvdemo_mdladv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.setInputParametersLayoutGrid([3 2]);
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

ModelAdvisor.LineBreak class

Purpose

Insert line break

Description

Use instances of the `ModelAdvisor.LineBreak` class to insert line breaks in the Model Advisor outputs.

Construction

ModelAdvisor.LineBreak	Insert line break

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Insert line break

Syntax

```
ModelAdvisor.LineBreak
```

Description

`ModelAdvisor.LineBreak` inserts a line break into the Model Advisor output.

Example

Add a line break between two lines of text:

```
result = ModelAdvisor.Paragraph;
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.List class

Purpose

Create list class

Description

Use instances of the `ModelAdvisor.List` class to create list-formatted outputs.

Construction

ModelAdvisor.List	Create list class

Methods

addItem	Add item to list
setType	Specify list type

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Create list class

Syntax

```
list = ModelAdvisor.List
```

Description

`list = ModelAdvisor.List` creates a list object, `list`.

Example

```
subList = ModelAdvisor.List();
setType(subList, 'numbered')
addItem(subList, ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
addItem(subList, ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.ListViewParameter class

Purpose

Add list view parameters to custom checks

Description

The Model Advisor uses list view parameters to populate the Model Advisor Result Explorer. Access the information in list views by clicking **Explore Result** in the Model Advisor window.

Construction

ModelAdvisor.ListViewParameter	Add list view parameters to custom checks

Properties

Attributes	Attributes to display in Model Advisor Report Explorer
Data	Objects in Model Advisor Result Explorer
Name	Drop-down list entry

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Example

Note The following example is a fragment of code from the `sl_customization.m` file for the demo model, `slvnvdemo_mdladv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object')';
```

```
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

ModelAdvisor.ListViewParameter

Purpose

Add list view parameters to custom checks

Syntax

```
lv_param = ModelAdvisor.ListViewParameter
```

Description

`lv_param = ModelAdvisor.ListViewParameter` defines a list view, `lv_param`.

Note Include list view parameter definitions in a check definition.

See Also

"Defining Model Advisor Result Explorer Views" on page 6-19 — Describes how to create check list views

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Batch-Fixing Warnings or Failures" — Describes how to use list views in the Model Advisor

"Demo and Code Example" on page 6-49 — Describes how to run a demo that shows how to customize the Model Advisor

"getListViewParameters" — Describes how to get list view parameters of a check

"setListViewParameters" — Describes how to set list view parameters of a check

Purpose

Create and format paragraph

Description

The `ModelAdvisor.Paragraph` class creates and formats a paragraph object.

Construction

ModelAdvisor.Paragraph	Create and format paragraph

Methods

addItem	Add item to paragraph
setAlign	Specify paragraph alignment

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Example

```
% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '...
'optimization settings:']);
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

“Formatting Model Advisor Results” on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.Paragraph

Purpose

Create and format paragraph

Syntax

```
para_obj = ModelAdvisor.Paragraph
```

Description

`para_obj = ModelAdvisor.Paragraph` defines a paragraph object `para_obj`.

Example

```
% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '...
'optimization settings:']);
```

Purpose

Identify root node

Description

The `ModelAdvisor.Root` class returns the root object.

Construction

ModelAdvisor.Root	Identify root node

Methods

publish	Publish object in Model Advisor root
register	Register object in Model Advisor root

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

ModelAdvisor.Root

Purpose

Identify root node

Syntax

```
root_obj = ModelAdvisor.Root
```

Description

`root_obj = ModelAdvisor.Root` creates a handle to the root object, `root_obj`.

Example

```
mdladvRoot = ModelAdvisor.Root;
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Create table

Description

Instances of the `ModelAdvisor.Table` class create and format a table. Specify the number of rows and columns in a table, excluding the table title and table heading row.

Construction

ModelAdvisor.Table	Create table

Methods

getEntry	Get table cell contents
setColHeading	Specify table column title
setColHeadingAlign	Specify column title alignment
setColWidth	Specify column widths
setEntry	Add cell to table
setEntryAlign	Specify table cell alignment
setHeading	Specify table title
setHeadingAlign	Specify table title alignment
setRowHeading	Specify table row title
setRowHeadingAlign	Specify table row title alignment

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.Table

Purpose

Create table

Syntax

```
table = ModelAdvisor.Table(row, column)
```

Description

`table = ModelAdvisor.Table(row, column)` creates a table object (`table`). The Model Advisor displays the table object containing the specified number of rows (`row`) and columns (`column`).

Examples

In the following example, you create two table objects, `table1` and `table2`. The Model Advisor displays `table1` in the results as a table with 1 row and 1 column. The Model Advisor display `table2` in the results as a table with 2 rows and 3 columns.

```
table1 = ModelAdvisor.Table(1,1);
table2 = ModelAdvisor.Table(2,3);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Define custom tasks

Description

The `ModelAdvisor.Task` class is a wrapper for a check so that you can access the check with the Model Advisor.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** is displayed in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Construction

ModelAdvisor.Task	Define custom tasks

Methods

setCheck	Specify check used in task

Properties

Description	Description of task
DisplayName	Name of task
Enable	Indicate if user can enable and disable task
ID	Identifier for task
LicenseName	Cell array of product license names required to enable check
MAObj	Model Advisor object
Value	Status of task
Visible	Indicate to display or hide task

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

Purpose

Define custom tasks

Syntax

```
task_obj = ModelAdvisor.Task(task_ID)
```

Description

`task_obj = ModelAdvisor.Task(task_ID)` creates a task object, `task_obj`, with a unique identifier, `task_ID`. `task_ID` must remain constant. If you do not specify `task_ID`, the Model Advisor assigns a random `task_ID` to the task object.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution appears** in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Examples

In the following example, you create three task objects, `MAT1`, `MAT2`, and `MAT3`.

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

ModelAdvisor.Text class

Purpose

Create Model Advisor text output

Description

Instances of `ModelAdvisor.Text` class create formatted text for the Model Advisor output.

Construction

ModelAdvisor.Text	Create Model Advisor text output

Methods

setBold	Specify bold text
setColor	Specify text color
setHyperlink	Specify hyperlinked text
setItalic	Italicize text
setRetainSpaceReturn	Retain spacing and returns in text
setSubscript	Specify subscripted text
setSuperscript	Specify superscripted text
setUnderlined	Underline text

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
t1 = ModelAdvisor.Text('This is some text');
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks
“Formatting Model Advisor Results” on page 6-44 — Describes how to format Model Advisor results

Purpose

Create Model Advisor text output

Syntax

```
text = ModelAdvisor.Text(content, attribute)
```

Description

`text = ModelAdvisor.Text(content, attribute)` creates a text object for the Model Advisor output.

Inputs

`content` — Optional string specifying the content of the text object. If `content` is empty, empty text is output.

`attribute` — Optional string specifying the formatting of the content. If no attribute is specified, the output text has default coloring with no formatting. Possible formatting options include:

- `normal` (default) — Text is default color and style.
- `bold` — Text is bold.
- `italic` — Text is italicized.
- `underlined` — Text is underlined.
- `pass` — Text is green.
- `warn` — Text is yellow.
- `fail` — Text is red.
- `keyword` — Text is blue.
- `subscript` — Text is subscripted.
- `superscript` — Text is superscripted.
- `retainspacereturn` — Text retains spacing and returns.

Outputs

`text`	The text object you create

Example

```
text = ModelAdvisor.Text('Sub entry 1', {'pass','bold'})
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

“Formatting Model Advisor Results” on page 6-44 — Describes how to format Model Advisor results

Purpose

Publish object in Model Advisor root

Syntax

```
publish(root_obj, check_obj, location)
publish(root_obj, group_obj)
publish(root_obj, fg_obj)
```

Description

`publish(root_obj, check_obj, location)` specifies where the Model Advisor places the check in the Model Advisor tree. `location` is either one of the subfolders in the **By Product** folder, or the name of a new subfolder to put in the **By Product** folder. Use a pipe-delimited string to indicate multiple subfolders. For example, to add a check to the **Simulink Verification and Validation > Modeling Standards** folder, use the following string: `'Simulink Verification and Validation|Modeling Standards'`.

`publish(root_obj, group_obj)` specifies the `ModelAdvisor.Group` object to publish as a folder in the **Model Advisor Task Manager** folder.

`publish(root_obj, fg_obj)` specifies the `ModelAdvisor.FactoryGroup` object to publish as a subfolder in the **By Task** folder.

Example

```
% publish check into By Product > Demo group.
mdladvRoot.publish(rec, 'Demo');
```

See Also

- "Defining Where Custom Checks Appear" on page 6-16 in the Simulink® Verification and Validation™ User's Guide on page 1
- "Defining Where Tasks Appear" on page 6-24 in the Simulink® Verification and Validation™ User's Guide on page 1
- "Defining Where Custom Folders Appear" on page 6-26 in the Simulink® Verification and Validation™ User's Guide on page 1

ModelAdvisor.Root.register

Purpose

Register object in Model Advisor root

Syntax

```
register(MAobj, obj)
```

Description

`register(MAobj, obj)` registers the object, *`obj`*, in the root object `MAobj`.

In the Model Advisor memory, the `register` method registers the following types of objects:

- `ModelAdvisor.Check`
- `ModelAdvisor.Task`
- `ModelAdvisor.Group`
- `ModelAdvisor.FactoryGroup`

The `register` method places objects in the Model Advisor memory that you use in other functions. The `register` method does not place objects in the Model Advisor tree.

Example

```
mdladvRoot = ModelAdvisor.Root;

MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task with input parameter and auto-fix ability';
MAT1.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT1);

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';
MAT2.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT2);

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
MAT3.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT3)
```

Purpose

Interact with Requirements Management Interface

Syntax

```
rmi setup
reqlinks = rmi('createempty')
reqlinks = rmi('get', object)
reqlinks = rmi('get', object, group)
rmi('set', object, reqlinks)
rmi('set', object, reqlinks, group)
rmi('cat', object, reqlinks)
cnt = rmi('count', object)
rmi('clearall', object)
rmi('clearAll', object, 'deep')
rmi register linktypename
rmi unregister linktypename
rmi linktypelist
cmdstr = rmi('navcmd', object)
[cmdstr, titlestr] = rmi('navcmd', object)
guidstr = rmi('guidget', object)
object = rmi('guidlookup', model, guidstr)
rmi('highlightModel', object)
rmi('unhighlightModel', object)
rmi('view', object, index)
dialog = rmi('edit', object)
rmi('copyObj', object)
```

Description

Use the `rmi` command to interact programmatically with the Requirements Management Interface (RMI).

RMI Setup

`rmi setup` configures the RMI for use with your computer and installs the interface for use with the Telelogic DOORS software, if needed. See "Configuring the Requirements Management Interface" on page 2-3 for more information about using this command to set up the RMI.

Requirement Link Management

`reqlinks = rmi('createempty')` creates an empty instance of the requirement links data structure.

`reqlinks = rmi('get', object)` returns the requirement links data structure for `object`.

`reqlinks = rmi('get', object, group)` returns the requirement links data structure for the Signal Builder group specified by the index `group`. In this case, `object` is the name or handle of a Signal Builder block whose signal groups are associated with requirements.

`rmi('set', object, reqlinks)` sets the requirement links data structure `reqlinks` to `object`.

`rmi('set', object, reqlinks, group)` sets the requirement links data structure `reqlinks` to the Signal Builder group specified by the index `group`. In this case, `object` is the name or handle of a Signal Builder block whose signal groups you want to associate with requirements.

`rmi('cat', object, reqlinks)` appends the requirement links data structure `reqlinks` to the end of the existing structure associated with `object`. If no structure exists, the RMI sets `reqlinks` to `object`.

`cnt = rmi('count', object)` returns the number of requirement links associated with `object`.

`rmi('clearall', object)` removes the requirement links data structure associated with `object`, deleting its requirements.

`rmi('clearAll', object, 'deep')` deletes all requirements links in the model containing `object`.

Managing Custom Link Types

`rmi register linktypename` registers the custom link type specified by the M-file function `linktypename`.

`rmi unregister linktypename` removes the custom link type specified by the M-file function `linktypename`.

`rmi linktypelist` displays a list of the currently registered link types. The list indicates whether each link type is built-in or custom and provides the path to the M-file function used for its registration.

Note See "Linking to Custom Types of Requirements Documents" on page 2-30 for more information.

Navigation and Display Options

`cmdstr = rmi('navcmd', object)` returns the MATLAB command string used to navigate to `object`. `object` is the name or handle of a Simulink or Stateflow object with which requirements can be associated.

`[cmdstr, titlestr] = rmi('navcmd', object)` returns the MATLAB command string `cmdstr` and the title string `titlestr` that provides descriptive text for `object`.

`guidstr = rmi('guidget', object)` returns the globally unique identifier for `object`. A globally unique identifier is created for `object` if it lacks one.

`object = rmi('guidlookup', model, guidstr)` returns the object name in `model` that has the globally unique identifier specified by `guidstr`.

`rmi('highlightModel', object)` highlights all of the objects in the parent model of `object` that have requirement links.

`rmi('unhighlightModel', object)` removes highlighting of objects in the parent model of `object` that have requirement links.

`rmi('view', object, index)` accesses the requirement numbered `index` in the requirements document associated with `object`. `index` is an integer that represents the *n*th requirement linked to `object`.

`dialog = rmi('edit', object)` displays the Requirements dialog box for `object` and returns the handle of the dialog box.

`rmi('copyObj', object)` resets the globally unique identifier for `object`, preserving its requirement links.

Inputs

`method` — "Description" on page 9-93 lists the valid method for this argument.

`object` — `object` is the name or handle of a Simulink or Stateflow object with which requirements can be associated.

`group` — Signal Builder group index

`reqlinks` — Requirement links are represented using a MATLAB structure array with the following fields:

- `doc` — String identifying requirements document
- `id` — String defining location in requirements document. The first character specifies the identifier type:

First Character	Identifier	Example
`?`	Search text, the first occurrence of which is located in requirements document	`'?Requirement 1'`
`@`	Named item, such as bookmark in a Microsoft Word file or an anchor in an HTML file	`'@my_req'`
`#`	Page or item number	`'#21'`
`>`	Line number	`'>3156'`
`$`	Worksheet range in a spreadsheet	`'$A2:C5'`

- `linked` — Boolean value specifying whether the requirement link is accessible for report generation

and highlighting. The default is `1` (true), specifying that the RMI can highlight the model object and include its requirement link in generated reports.

- `description` — String describing the requirement.
- `keywords` — Optional string supplementing `description`.
- `reqsys` — String identifying the link type registration name, `'other'` for built-in link types.

`guidstr`	Globally unique model identifier
`index`	Integer that represents the *n*th requirement linked to `object`

Outputs

`reqlinks`	See "Inputs" on page 9-96.
`cnt`	Number of requirement links associated with `object`
`cmdstr`	MATLAB command string
`titlestr`	Descriptive text for `object`
`guidstr`	Globally unique model identifier
`object`	`object` is the name or handle of a Simulink or Stateflow object with which requirements can be associated.
`dialog`	Handle for `object`

Examples

This example uses the `rmi` command to get the requirements from a block, change the description, and save it to the corresponding block:

```
>> rmi setup
Registered the requirements Active-X controls
>> slvnvdemo_fuelsys_htmreq
>> blk_with_req =  ['slvnvdemo_fuelsys_htmreq/fuel rate' 10 'controller/Airflow calculation'];
```

```
>> reqts = rmi('get',blk_with_req)

reqts =

           doc: 'fuelsys_requirements.htm'
            id: 'REQ1'
        linked: 1
   description: 'Using a few parameters to estimate the mass airflow through the engine'
      keywords: 'REQ1'
        reqsys: 'HTML'

>> reqts.description = 'Mass airflow estimation'

reqts =

           doc: 'fuelsys_requirements.htm'
            id: 'REQ1'
        linked: 1
   description: 'Mass airflow estimation'
      keywords: 'REQ1'
        reqsys: 'HTML'

>> rmi('set', blk_with_req, reqts)
>> rmi('get', blk_with_req)

ans =

           doc: 'fuelsys_requirements.htm'
            id: 'REQ1'
        linked: 1
   description: 'Mass airflow estimation'
      keywords: 'REQ1'
        reqsys: 'HTML'
```

The next example adds a new requirement to the block used in the previous example:

```
>> new_req = rmi('createempty')

new_req =

            doc: ''
             id: ''
         linked: 1
    description: ''
       keywords: ''
         reqsys: 'other'

>> new_req.doc = 'fuelsys_requirements2.htm'

new_req =

            doc: 'fuelsys_requirements2.htm'
             id: ''
         linked: 1
    description: ''
       keywords: ''
         reqsys: 'other'

>> new_req.description = 'A new requirement'

new_req =

            doc: 'fuelsys_requirements2.htm'
             id: ''
         linked: 1
    description: 'A new requirement'
       keywords: ''
         reqsys: 'other'

>> rmi('cat',blk_with_req, new_req)

ans =
```

```
2x1 struct array with fields:
    doc
    id
    linked
    description
    keywords
    reqsys
```

See Also

- Chapter 2, “Managing Model Requirements”

Purpose

Start Requirements Management Interface

Syntax

```
rminav
```

Description

`rminav` starts the Requirements Management Interface Navigator window.

If you specified `reqsys = 'OTHERS'` in the MATLAB M-file `reqmgropts.m`, the standard version of the Requirements Management Interface Navigator window opens. You can associate requirements documents written in HTML or Microsoft Word and Excel software with Simulink models, Stateflow charts, and MATLAB M-files.

If you specified `reqsys = 'DOORS'` in `reqmgropts.m`, the DOORS software version of the Requirements Management Interface Navigator window opens. You can associate requirements in the Telelogic DOORS software with Simulink models, Stateflow charts, and MATLAB M-files.

To associate requirements in the DOORS software with MATLAB objects, you must start the MATLAB software with the `/automation` option.

See Also

- `rmi`

ModelAdvisor.Check.setAction

Purpose

Specify action for check

Syntax

```
setAction(check_obj, action_obj)
```

Description

`setAction(check_obj, action_obj)` returns the action object `action.obj` to use in the check `check_obj`. The `setAction` method identifies the action you want to use in a check.

See Also

`ModelAdvisor.Action` — Create custom actions

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

Purpose

Specify paragraph alignment

Syntax

```
setAlign(paragraph, alignment)
```

Description

`setAlign(paragraph, alignment)` specifies the alignment of text. Possible values are:

- `'left'` (default)
- `'right'`
- `'center'`

Example

```
report_paragraph = ModelAdvisor.Paragraph;
setAlign(report_paragraph, 'center');
```

ModelAdvisor.Text.setBold

Purpose

Specify bold text

Syntax

`setBold(text, mode)`

Description

`setBold(text, mode)` specifies whether `text` should be formatted in bold font.

Inputs

`text`	Instantiation of the ModelAdvisor.Text class
`mode`	A Boolean value indicating bold formatting of text: • `true` — Format the text in bold font. • `false` — Do not format the text in bold font.

Example

```
t1 = ModelAdvisor.Text('This is some text');
setBold(t1, 'true');
```

Purpose

Specify action callback function

Syntax

```
setCallbackFcn(action_obj, @handle)
```

Description

`setCallbackFcn(action_obj, @handle)` specifies the handle to the callback function, `handle`, to use with the action object, `action_obj`.

Example

Note The following example is a fragment of code from the `sl_customization.m` file for the demo model, `slvnvdemo_mdladv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

```
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
modifyAction.setCallbackFcn(@modifyOptmizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
                            ' settings that can impact safety'];
modifyAction.Enable = true;
rec.setAction(modifyAction);
```

See Also

"Defining Check Actions" on page 6-21 — Describes how to create custom actions
Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks
"setActionEnable" — Set enable/disable status for check action

ModelAdvisor.Check.setCallbackFcn

Purpose

Specify callback function for check

Syntax

```
setCallbackFcn(check_obj, @handle, context, style)
```

Description

`setCallbackFcn(check_obj, @handle, context, style)` specifies the callback function to use with the check, `check_obj`.

Inputs

`check_obj`	Instantiation of the `ModelAdvisor.Check` class
`handle`	Handle to a check callback function
`context`	Context for checking the model or subsystem: • `'None'` — No special requirements. • `'PostCompile'` — The model must be compiled.
`style`	Type of callback function: • `'StyleOne'` — Simple check callback function, for formatting results using template • `'StyleTwo'` — Detailed check callback function • `'StyleThree'` — Check callback functions with hyperlinked results

Example

```
% --- sample check 1
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback,'None','StyleThree');
```

See Also

"Creating Callback Functions and Results" on page 6-28 — Describes how to create check callback functions

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

ModelAdvisor.Task.setCheck

Purpose

Specify check used in task

Syntax

```
setCheck(task, check_ID)
```

Description

`setCheck(task, check_ID)` specifies the check to use in the task.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution appears** in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Inputs

`task`	Instantiation of the `ModelAdvisor.Task` class
`check_ID`	A unique string that identifies the check to use in the task

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
setCheck(MAT1, 'com.mathworks.sample.Check1');
```

Purpose

Add description of check to result

Syntax

`setCheckText(ft_obj, 'text')`

Description

`setCheckText(ft_obj, 'text')` is an optional method that adds text as the first item in the report. Use this method to add information describing the overall check. *`ft_obj`* is a handle to a template object. *`text`* is a string that appears as the first line in the analysis result.

Examples

Create a list object, `ft`, and add a line of text to the result:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setCheckText(ft, ['Identify unconnected lines, input ports,'...
    'and output ports in the model']);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.Table.setColHeading

Purpose

Specify table column title

Syntax

```
setColHeading(table, column, heading)
```

Description

`setColHeading(table, column, heading)` specifies that the column header of `column` is set to `heading`.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`column`	An integer specifying the column number
`heading`	A string, element object, or object array specifying the table column title

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setColHeading(table1, 1, 'Header 1');
setColHeading(table1, 2, 'Header 2');
setColHeading(table1, 3, 'Header 3');
```

Purpose

Specify column title alignment

Syntax

```
setColHeadingAlign(table, column, alignment)
```

Description

`setColHeadingAlign(table, column, alignment)` specifies the alignment of the column heading.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`column`	An integer specifying the column number
`alignment`	Alignment of the column heading. *`alignment`* can have one of the following values: • `left` (default) • `right` • `center`

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setColHeading(table1, 1, 'Header 1');
setColHeadingAlign(table1, 1, 'center');
setColHeading(table1, 2, 'Header 2');
setColHeadingAlign(table1, 2, 'center');
setColHeading(table1, 3, 'Header 3');
setColHeadingAlign(table1, 3, 'center');
```

ModelAdvisor.Text.setColor

Purpose

Specify text color

Syntax

```
setColor(text, color)
```

Description

`setColor(text, color)` sets the `text` color to *`color`*.

Inputs

`text`	Instantiation of the `ModelAdvisor.Text` class
`color`	An enumerated string specifying the color of the text. Possible formatting options include: • `normal` (default) — Text is default color. • `pass` — Text is green. • `warn` — Text is yellow. • `fail` — Text is red. • `keyword` — Text is blue.

Example

```
t1 = ModelAdvisor.Text('This is a warning');
setColor(t1, 'warn');
```

Purpose

Specify number of columns for input parameter

Syntax

```
setColSpan(input_param, [start_col end_col])
```

Description

`setColSpan(input_param, [start_col end_col])` specifies the number of columns that the parameter occupies. Use the `setColSpan` method to specify where you want an input parameter located in the layout grid when there are multiple input parameters.

Inputs

`input_param`	Instantiation of the `ModelAdvisor.InputParameter` class
`start_col`	A positive integer representing the first column that the input parameter occupies in the layout grid
`end_col`	A positive integer representing the last column that the input parameter occupies in the layout grid

Example

```
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
```

ModelAdvisor.FormatTemplate.setColTitles

Purpose

Add column titles to table

Syntax

```
setColTitles(ft_obj, {'col_title_1', 'col_title_2', ...})
```

Description

`setColTitles(ft_obj, {'col_title_1', 'col_title_2', ...})` is method you must use when you create a template object that is a table type. Use it to specify the titles of the columns in the table. *`ft_obj`* is a handle to a table template object. The second input is a cell of strings specifying the column titles. The order of the inputs determines which column the title is in. If you do not add data to the table, the Model Advisor does not display the table in the result.

Note Before adding data to a table, you must specify column titles.

Examples

Create a table object, `ft`, and specify two column titles:

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setColTitle(ft, {'Index', 'Block Name'});
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Specify column widths

Syntax

```
setColWidth(table, column, width)
```

Description

`setColWidth(table, column, width)` specifies the `column`.

The `setColWidth` method specifies the table column widths relative to the entire table width. If column widths are [1 2 3], the second column is twice the width of the first column, and the third column is three times the width of the first column. Unspecified columns have a default width of 1. For example:

```
setColWidth(1, 1);
setColWidth(3, 2);
```

specifies [1 1 2] column widths.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`column`	An integer specifying column number
`width`	An integer or array of integers specifying the column widths, relative to the entire table width

Example

```
table1 = ModelAdvisor.Table(2, 3)
setColWidth(table1, 1, 1);
setColWidth(table1, 3, 2);
```

ModelAdvisor.Table.setEntry

Purpose

Add cell to table

Syntax

```
setEntry(table, row, column, string)
setEntry(table, row, column, content)
```

Description

`setEntry(table, row, column, string)` adds a string to a cell in a table.

`setEntry(table, row, column, content)` adds an object specified by `content` to a cell in a table.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying the row
`column`	An integer specifying the column
`string`	A string representing the contents of the entry
`content`	An element object or object array specifying the content of the table entries

Example

Create two tables and insert `table2` into the first cell of `table1`:

```
table1 = ModelAdvisor.Table(1, 1);
table2 = ModelAdvisor.Table(2, 3);
.
.
.
setEntry(table1, 1, 1, table2);
```

Purpose

Specify table cell alignment

Syntax

```
setEntryAlign(table, row, column, alignment)
```

Description

`setEntryAlign(table, row, column, alignment)` specifies the cell alignment of the designated cell.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying row number
`column`	An integer specifying column number
`alignment`	A string specifying the cell alignment. Possible values are: • `left` (default) • `right` • `center`

Example

```
table1 = ModelAdvisor.Table(2,3);
setHeading(table1, 'New Table');
.
.
.
setEntry(table1, 1, 1, 'First Entry');
setEntryAlign(table1, 1, 1, 'center');
```

ModelAdvisor.Table.setHeading

Purpose

Specify table title

Syntax

```
setHeading(table, title)
```

Description

`setHeading(table, title)` specifies the table title.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`title`	A string, element object, or object array that specifies the table title

Example

```
table1 = ModelAdvisor.Table(2, 3);
setHeading(table1, 'New Table');
```

Purpose

Specify table title alignment

Syntax

```
setHeadingAlign(table, alignment)
```

Description

`setHeadingAlign(table, alignment)` specifies the alignment for the table title.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`alignment`	A string specifying the table title alignment. Possible values are: • `left` (default) • `right` • `center`

Example

```
table1 = ModelAdvisor.Table(2, 3);
setHeading(table1, 'New Table');
setHeadingAlign(table1, 'center');
```

ModelAdvisor.Image.setHyperlink

Purpose

Specify hyperlink location

Syntax

```
setHyperlink(image, url)
```

Description

`setHyperlink(image, url)` specifies the target location of the hyperlink associated with `image`.

Inputs

`image`	Instantiation of the `ModelAdvisor.Image` class
`url`	A string specifying the target URL

Example

```
matlab_logo=ModelAdvisor.Image;
setHyperlink(matlab_logo, 'http://www.mathworks.com');
```

Purpose

Specify hyperlinked text

Syntax

```
setHyperlink(text, url)
```

Description

`setHyperlink(text, url)` creates a hyperlink from the text to the specified URL.

Inputs

`text`	Instantiation of the `ModelAdvisor.Text` class
`url`	A string that specifies the target location of the URL

Examples

```
t1 = ModelAdvisor.Text('MathWorks home page');
setHyperlink(t1, 'http://www.mathworks.com');
```

ModelAdvisor.Image.setImageSource

Purpose

Specify image location

Syntax

```
setImageSource(image_obj, source)
```

Description

`setImageSource(image_obj, source)` specifies the location of the image.

Inputs

`image_obj`	Instantiation of the `ModelAdvisor.Image` class
source	A string specifying the location of the image file

Purpose

Add description of subcheck to result

Syntax

`setInformation(ft_obj, 'text')`

Description

`setInformation(ft_obj, 'text')` is an optional method that adds text as the first item after the subcheck title. Use this method to add information describing the subcheck. *`ft_obj`* is a handle to a template object. *`text`* is a string that describes the subcheck. The Model Advisor displays the text after the title of the subcheck.

Examples

Create a list object, `ft`, and specify a subcheck title and description:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft, ['Check for constructs in the model '...
    'that are not supported when generating code']);
setInformation(ft, ['Identify blocks that should not '...
    'be used for code generation.']);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.Check.setInputParameters

Purpose

Specify input parameters for check

Syntax

```
setInputParameters(check_obj, params)
```

Description

`setInputParameters(check_obj, params)` specifies `ModelAdvisor.InputParameter` objects (`params`) to be used as input parameters to a check (`check_obj`).

Inputs

`check_obj`	Instantiation of the `ModelAdvisor.Check` class
`params`	A cell array of `ModelAdvisor.InputParameters` objects

Examples

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
inputParam1 = ModelAdvisor.InputParameter;
inputParam2 = ModelAdvisor.InputParameter;
inputParam3 = ModelAdvisor.InputParameter;
setInputParameters(rec, {inputParam1,inputParam2,inputParam3});
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

`ModelAdvisor.InputParameter`— Add input parameters to custom checks

Purpose

Specify layout grid for input parameters

Syntax

```
setInputParametersLayoutGrid(check_obj, [row col])
```

Description

`setInputParametersLayoutGrid(check_obj, [row col])` specifies the layout grid for input parameters in the Model Advisor. Use the `setInputParametersLayoutGrid` method if there are multiple input parameters.

Inputs

`check_obj`	Instantiation of the `ModelAdvisor.Check` class
`row`	Number of rows in the layout grid
`col`	Number of columns in the layout grid

Example

```
% --- sample check 1
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback,'None','StyleThree');
rec.setInputParametersLayoutGrid([3 2]);
```

See Also

`ModelAdvisor.InputParameter` — Add input parameters to custom checks

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

ModelAdvisor.Text.setItalic

Purpose

Italicize text

Syntax

```
setItalic(text, mode)
```

Description

`setItalic(text, mode)` specifies whether `text` should be italicized.

Inputs

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating italic formatting of text: • `true` — Italicize the text. • `false` — Do not italicize the text.

Example

```
t1 = ModelAdvisor.Text('This is some text');
setItalic(t1, 'true');
```

Purpose

Add list of hyperlinks to model objects

Syntax

`setListObj(ft_obj, {model_obj})`

Description

`setListObj(ft_obj, {model_obj})` is an optional method that generates a bulleted list of hyperlinks to model objects. *`ft_obj`* is a handle to a list template object. *`model_obj`* is a cell array of handles or full paths to blocks, or model objects that the Model Advisor displays as a bulleted list of hyperlinks in the report.

Examples

Create a list object, `ft`, and add a list of the blocks found in the model:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');

% Find all the blocks in the system
allBlocks = find_system(system);

% Add the blocks to a list
setListObj(ft, allBlocks);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setRecAction

Purpose

Add Recommended Action section and text

Syntax

`setRecAction(ft_obj, {'text'})`

Description

`setRecAction(ft_obj, {'text'})` is an optional method that adds a Recommended Action section to the report. Use this method to describe how to fix the check. *`ft_obj`* is a handle to a template object. `{'text'}` is a cell array of strings describing the recommended action to fix the issues reported by the check. The Model Advisor displays the recommended action as a separate section below the list or table in the report.

Examples

Create a list object, `ft`, find Gain blocks in the model, and recommend changing them:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
% Find all Gain blocks
gainBlocks = find_system(gcs, 'BlockType','Gain');

% Find Gain blocks with expression evaluates to 1
for idx = 1:length(gainBlocks)
    gainObj = get_param(gainBlocks(idx), 'Object');
    resGain = slResolve(gainObj.Gain, gainObj.getFullName);
    if ~isempty(resGain)
        % Find the first index that computes to 1
        if ~isempty(find(resGain == 1, 1))
            setRecAction(ft, {'If you are using these blocks '...
                'as buffers, you should replace them with '...
                'Signal Conversion blocks'});
        end
    end
end
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Add See Also section and links

Syntax

```
setRefLink(ft_obj, {{'standard'}})
setRefLink(ft_obj, {{'url', 'standard'}})
```

Description

`setRefLink(ft_obj, {{'standard'}})` is an optional method that adds a See Also section above the table or list in the result. Use this method to add references to standards. *`ft_obj`* is a handle to a template object. *`standard`* is a cell array of strings that you want to display in the result. If you include more than one cell, the Model Advisor displays the strings in a bulleted list.

`setRefLink(ft_obj, {{'url', 'standard'}})` generates a list of links in the See Also section. *`url`* is a string that indicates the location to link to. You must provide the full link including the protocol. For example, `http:\\www.mathworks.com` is a valid link, while `www.mathworks.com` is not a valid link. You can create a link to any protocol that is valid URL, such as a web site address, a full path to a file, or a relative path to a file.

Note `setRefLink` expects a cell array of cell arrays for the second input.

Examples

Create a list object, `ft`, and add a related standard:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setRefLink(ft, {{'IEC 61508-3, Table A.3 (3) ''Language subset'''}});
```

Create a list object, `ft`, and add a list of related standards:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setRefLink(ft, {
    {'IEC 61508-3, Table A.3 (2) ''Strongly typed programming language'''},...
    {'IEC 61508-3, Table A.3 (3) ''Language subset'''}});
```

See Also

Chapter 6, “Customizing the Model Advisor” — Describes how to create custom checks

“Formatting Model Advisor Results” on page 6-44 — Describes how to format Model Advisor results

Purpose

Retain spacing and returns in text

Syntax

```
setRetainSpaceReturn(text, mode)
```

Description

`setRetainSpaceReturn(text, mode)` specifies whether the text must retain the spaces and carriage returns.

Inputs

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating whether to preserve spaces and carriage returns in the text: • `true` (default) — Preserve spaces and carriage returns. • `false` — Do not preserve spaces and carriage returns.

Example

```
t1 = ModelAdvisor.Text('MathWorks home page');
setRetainSpaceReturn(t1, 'true');
```

ModelAdvisor.Table.setRowHeading

Purpose

Specify table row title

Syntax

```
setRowHeading(table, row, heading)
```

Description

`setRowHeading(table, row, heading)` specifies a title for the designated table row.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying row number
`heading`	A string, element object, or object array specifying the table row title

Example

```
table1 = ModelAdvisor.Table(2,3);
setRowHeading(table1, 1, 'Row 1 Title');
setRowHeading(table1, 2, 'Row 2 Title');
setRowHeading(table1, 3, 'Row 3 Title');
```

Purpose

Specify table row title alignment

Syntax

```
setRowHeadingAlign(table, row, alignment)
```

Description

`setRowHeadingAlign(table, row, alignment)` specifies the alignment for the designated table row.

Inputs

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying row number.
`alignment`	A string specifying the cell alignment. Possible values are: • `left` (default) • `right` • `center`

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setRowHeading(table1, 1, 'Row 1 Title');
setRowHeadingAlign(table1, 1, 'center');
setRowHeading(table1, 2, 'Row 2 Title');
setRowHeadingAlign(table1, 2, 'center');
setRowHeading(table1, 3, 'Row 3 Title');
setRowHeadingAlign(table1, 3, 'center');
```

ModelAdvisor.InputParameter.setRowSpan

Purpose

Specify rows for input parameter

Syntax

```
setRowSpan(input_param, [start_row end_row])
```

Description

`setRowSpan(input_param, [start_row end_row])` specifies the number of rows that the parameter occupies. Specify where you want an input parameter located in the layout grid when there are multiple input parameters.

Inputs

`input_param`	The input parameter object
`start_row`	A positive integer representing the first row that the input parameter occupies in the layout grid
`end_row`	A positive integer representing the last row that the input parameter occupies in the layout grid

Examples

```
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
```

Purpose

Add line between subcheck results

Syntax

`setSubBar(ft_obj, value)`

Description

`setSubBar(ft_obj, value)` is an optional method that adds lines between results for subchecks. *`ft_obj`* is a handle to a template object. *`value`* is a boolean value that specifies when the Model Advisor includes a line between subchecks in the check results. By default, the value is `true`, and the Model Advisor displays the bar. The Model Advisor does not display the bar when you set the value to `false`.

Examples

Create a list object, `ft`, turn off the subbar:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubBar(ft, false);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setSubResultStatus

Purpose

Add status to check or subcheck result

Syntax

```
setSubResultStatus(ft_obj, 'status')
```

Description

`setSubResultStatus(ft_obj, 'status')` is an optional method that displays the status in the result. Use this method to display the status of the check or subcheck in the result. *`ft_obj`* is a handle to a template object. *`status`* is a string identifying the status of the check. Valid strings are:

```
Pass
Warn
Fail
```

Examples

Create a list object, `ft`, and add a passing status:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubResutlStatus(ft, 'Pass');
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setSubResultStatusText

Purpose

Add text below status in result

Syntax

`setSubResultStatusText(`*`ft_obj`*`, '`*`message`*`')`

Description

`setSubResultStatusText(`*`ft_obj`*`, '`*`message`*`')` is an optional method that displays text below the status in the result. Use this method to describe the status. *`ft_obj`* is a handle to a template object. *`message`* is a string that the Model Advisor displays below the status in the report.

Examples

Create a list object, `ft`, add a passing status and a description of why the check passed:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubResutlStatus(ft, 'Pass');
setSubResultStatusText(ft, ['Constructs that are not supported when '...
    'generating code were not found in the model or subsystem']);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.Text.setSubscript

Purpose

Specify subscripted text

Syntax

```
setSubscript(text, mode)
```

Description

`setSubscript(text, mode)` indicates whether to make `text` subscript.

Inputs

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating subscripted formatting of text: • `true` — Make the text subscript. • `false` — Do not make the text subscript.

Example

```
t1 = ModelAdvisor.Text('This is some text');
setSubscript(t1, 'true');
```

Purpose

Specify superscripted text

Syntax

```
setSuperscript(text, mode)
```

Description

`setSuperscript(text, mode)` indicates whether to make `text` subscript.

Inputs

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating superscripted formatting of text: • `true` — Make the text superscript. • `false` — Do not make the text superscript.

Example

```
t1 = ModelAdvisor.Text('This is some text');
setSuperscript(t1, 'true');
```

ModelAdvisor.FormatTemplate.setSubTitle

Purpose

Add title for subcheck in result

Syntax

`setSubTitle(ft_obj, 'text')`

Description

`setSubTitle(ft_obj, 'text')` is an optional method that adds a subcheck result title. Use this method when you create subchecks to distinguish between them in the result. *`ft_obj`* is a handle to template object. *`text`* is a string specifying the title of the subcheck.

Examples

Create a list object, `ft`, and add a subcheck title:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft, ['Check for constructs in the model '...
    'that are not supported when generating code']);
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Add data to table

Syntax

```
setTableInfo(ft_obj, {data})
```

Description

`setTableInfo(ft_obj, {data})` is an optional method that creates a table. *`ft_obj`* is a handle to a table template object. *`data`* is a cell array of strings or objects specifying the information in the body of the table. The Model Advisor creates hyperlinks to objects. If you do not add data to the table, the Model Advisor does not display the table in the result.

Note Before creating a table, you must specify column titles using the `setColTitle` method.

Examples

Create a table object, `ft`, add column titles, and add data to the table:

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setColTitle(ft, {'Index', 'Block Name'});
setTableInfo(ft, {'1', 'Gain'});
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setTableTitle

Purpose

Add title to table

Syntax

`setTableTitle(ft_obj, 'title')`

Description

`setTableTitle(ft_obj, 'title')` is an optional method that adds a title to a table. *`ft_obj`* is a handle to a table template object. *`title`* is a string indicating the title of the table. The string appears above the table. If you do not add data to the table, the Model Advisor does not display the table and title in the result.

Examples

Create a table object, `ft`, and add a table title:

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setTitleTitle(ft, 'Table of fonts and styles used in model');
```

See Also

Chapter 6, "Customizing the Model Advisor" — Describes how to create custom checks

"Formatting Model Advisor Results" on page 6-44 — Describes how to format Model Advisor results

Purpose

Specify list type

Syntax

`setType(list_obj, listType)`

Description

`setType(list_obj, listType)` specifies the type of list the `ModelAdvisor.List` constructor creates.

Inputs

`list_obj`	Instantiation of the `ModelAdvisor.List` class
`listType`	Specifies the list type: • `numbered` • `bulleted`

Example

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

ModelAdvisor.Text.setUnderlined

Purpose

Underline text

Syntax

`setUnderlined(text, mode)`

Description

`setUnderlined(text, mode)` indicates whether to underline `text`.

Inputs

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating underlined formatting of text: • `true` — Underline the text. • `false` — Do not underline the text.

Example

```
t1 = ModelAdvisor.Text('This is some text');
setUnderlined(t1, 'true');
```

Purpose

Display signal range coverage information for model object

Syntax

```
[min, max] = sigrangeinfo(cvdo, object)
[min, max] = sigrangeinfo(cvdo, object, portID)
```

Description

`[min, max] = sigrangeinfo(cvdo, object)` returns the minimum and maximum signal values output by the model component `object` within the `cvdata` object `cvdo`.

`[min, max] = sigrangeinfo(cvdo, object, portID)` returns the minimum and maximum signal values associated with the output port `portID` of the Simulink block `object`.

Inputs

`cvdo`

`cvdata` object

`object`

The `object` argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for `object` include the following:

Object Specification	Description
`BlockPath`	Full path to a Simulink model or block
`BlockHandle`	Handle to a Simulink model or block
`slObj`	Handle to a Simulink API object
`sfID`	Stateflow ID
`sfObj`	Handle to a Stateflow API object

`{BlockPath, sfID}`	Cell array with the path to a Stateflow block and the ID of an object contained in that chart
`{BlockPath, sfObj}`	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
`[BlockHandle, sfID]`	Array with a Stateflow block handle and the ID of an object contained in that chart

`portID` Output port of the Simulink block `object`

Outputs

`min` Minimum signal values output by the model component `object` within the `cvdata` object `cvdo`. If `object` outputs a vector, `min` and `max` are also vectors.

`max` Maximum signal values output by the model component `object` within the `cvdata` object `cvdo`. If `object` outputs a vector, `min` and `max` are also vectors.

Example

The following commands open the `slvnvdemo_cv_small_controller` demo model, create the test specification object `testObj`, enable signal range coverage for `testObj`, and execute `testObj`:

```
mdl = 'slvnvdemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.sigrange = 1;
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the signal range for the Product block.

```
blk_handle = get_param([mdl, '/Product'], 'Handle');
[minVal, maxVal] = sigrangeinfo(data, blk_handle)
```

tableinfo

Purpose

Display lookup table coverage information for model object

Syntax

```
coverage = tableinfo(cvdo, object)
coverage = tableinfo(cvdo, object, ignore_descendants)
[coverage, exeCounts] = tableinfo(cvdo, object)
[coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)
```

Description

`coverage = tableinfo(cvdo, object)` returns lookup table coverage results from the `cvdata` object `cvdo` for the model component specified by `object`.

`coverage = tableinfo(cvdo, object, ignore_descendants)` returns lookup table coverage results for `object`, depending on the value of `ignore_descendants`.

`[coverage, exeCounts] = tableinfo(cvdo, object)` returns lookup table coverage results and the execution count for each interpolation/extrapolation interval in the lookup table block specified by `object`.

`[coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)` returns lookup table coverage results, the execution count for each interpolation/extrapolation interval, and the execution counts for breakpoint equality.

Inputs

`cvdo`	`cvdata` object
`object`	Full path or handle to a Simulink lookup table block or a model containing such a block.
`ignore_descendants`	Specifies to ignore the coverage of descendant objects if `ignore_descendants` is set to `1`.

Outputs

`coverage`

The value of `coverage` is a two-element vector of form `[covered_intervals total_intervals]`, the elements of which are:

- `covered_intervals` — Number of interpolation/extrapolation intervals satisfied for `object`
- `total_intervals` — Total number of interpolation/extrapolation intervals for `object`

Note `coverage` is empty if `cvdo` does not contain lookup table coverage results for `object`.

`execounts`

`exeCounts` is an array having the same dimensionality as the lookup table block; however, its size has been extended to allow for the lookup table extrapolation intervals.

`brkEquality`

`brkEquality` is a cell array containing vectors that identify the number of times in a simulation the lookup table block input was equivalent to a breakpoint value. Each vector represents the breakpoints along a different lookup table dimension.

Example

The following commands open the `slvnvdemo_cv_small_controller` demo model, create the test specification object `testObj`, enable lookup table coverage for `testObj`, and execute `testObj`:

```
mdl = 'slvnvdemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
```

```
testObj.settings.tableExec = 1;
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the lookup table coverage results for the Gain Table block (in the Gain subsystem) and determine its percentage of interpolation/extrapolation intervals covered:

```
blk_handle = get_param([mdl, '/Gain/Gain Table'], 'Handle');
cov = tableinfo(data, blk_handle)
percent_cov = 100 * cov(1) / cov(2)
```

Alternatives

In the Coverage Settings dialog box, on the **Coverage** tab, under **Coverage Metrics**, select **Look-up Table Coverage**.

See Also

- "Lookup Table Coverage (LUT)" on page 5-5
- `conditioninfo`
- `decisioninfo`
- `mcdcinfo`
- `sigrangeinfo`

ModelAdvisor.ListViewParameter.Attributes property

Purpose

Attributes to display in Model Advisor Report Explorer

Values

Cell array

Default: `{}` (empty cell array)

Description

The `Attributes` property specifies the attributes to display in the center pane of the Model Advisor Results Explorer.

Example

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object')';
myLVParam.Attributes = {'FontName'}; % name is default property
```

ModelAdvisor.Check.CallbackContext property

Purpose

Model or subsystem context

Values

`'PostCompile'`
`'None'` (default)

Description

The `CallbackContext` property specifies the context for checking the model or subsystem.

`'None'`	No special requirements for the model before checking.
`'Postcompile'`	The model must be compiled.

Purpose

Callback function handle for check

Values

Function handle.

An empty handle `[ ]` is the default.

Description

The `CallbackHandle` property specifies the handle to the check callback function.

ModelAdvisor.Check.CallbackStyle property

Purpose

Callback function type

Values

`'StyleOne'` (default)
`'StyleTwo'`
`'StyleThree'`

Description

The `CallbackStyle` property specifies the type of the callback function.

`'StyleOne'`	Simple check callback function
`'StyleTwo'`	Detailed check callback function
`'StyleThree'`	Check callback function with hyperlinked results

Purpose

Objects in Model Advisor Result Explorer

Values

Array of Simulink objects

Default: `[]` (empty array)

Description

The `Data` property specifies the objects displayed in the Model Advisor Result Explorer.

Example

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object')';
```

ModelAdvisor.Action.Description property

Purpose

Message in **Action** box

Values

String

Default:`''` (null string)

Description

The `Description` property specifies the message displayed in the Action box.

Example

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
%Specify a callback function for the action
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
myAction.Description=...
   'Click the button to update all blocks with specified font';
```

Purpose

Description of folder

Values

String

Default: `''` (null string)

Description

The `Description` property provides information about the folder. Details about the folder are displayed in the right pane of the Model Advisor.

Example

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.Description='Demo Factory Group';
```

ModelAdvisor.Group.Description property

Purpose

Description of folder

Values

String

Default: `''` (null string)

Description

The `Description` property provides information about the folder. Details about the folder are displayed in the right pane of the Model Advisor.

Example

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.Description='This is my group';
```

ModelAdvisor.InputParameter.Description property

Purpose

Description of input parameter

Values

String.

Default: `''` (null string)

Description

The `Description` property specifies a description of the input parameter. Details about the check are displayed in the right pane of the Model Advisor.

Example

```
% define input parameters
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
```

ModelAdvisor.Task.Description property

Purpose

Description of task

Values

String

Default: `''` (null string)

Description

The `Description` property is a description of the task that the Model Advisor displays in the **Analysis** box.

When adding checks as tasks, the Model Advisor uses the task `Description` property instead of the check `TitleTips` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task 1';
MAT1.Description='This is the first example task.'

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';
MAT2.Description='This is the second example task.'

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
MAT3.Description='This is the third example task.'
```

ModelAdvisor.FactoryGroup.DisplayName property

Purpose

Name of folder

Values

String

Default:'' (null string)

Description

The `DisplayName` specifies the name of the folder that is displayed in the Model Advisor.

Examples

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='Demo Factory Group';
```

ModelAdvisor.Group.DisplayName property

Purpose

Name of folder

Values

String

Default:'' (null string)

Description

The `DisplayName` specifies the name of the folder that is displayed in the Model Advisor.

Examples

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.DisplayName='My Group';
```

Purpose

Name of task

Values

String

Default: `''` (null string)

Description

The `DisplayName` property specifies the name of the task. The Model Advisor displays each custom task in the tree using the name of the task. Therefore, you should specify a unique name for each task. When you specify the same name for multiple tasks, the Model Advisor generates a warning.

When adding checks as tasks, the Model Advisor uses the task `DisplayName` property instead of the check `Title` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task with input parameter and auto-fix ability';

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
```

ModelAdvisor.Check.Enable property

Purpose

Indicate whether user can enable or disable check

Values

`true` (default)
`false`

Description

The `Enable` property specifies whether the user can enable or disable the check.

`true`	Display the check box control
`false`	Hide the check box control

Purpose

Indicate if user can enable and disable task

Values

`true` (default)
`false`

Description

The `Enable` property specifies whether the user can enable or disable a task.

`true` (default)	Display the check box control for task
`false`	Hide the check box control for task

When adding checks as tasks, the Model Advisor uses the task `Enable` property instead of the check `Enable` property.

Example

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.Enable ='false';
```

ModelAdvisor.InputParameter.Entries property

Purpose

Drop-down list entries

Values

Depends on the value of the `Type` property.

Description

The `Entries` property is valid only when the `Type` property is one of the following:

- `Enum`
- `ComboBox`
- `PushButton`

Examples

```
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
```

Purpose

Identifier for check

Values

String

Default: `''` (null string)

Description

The `ID` property specifies a permanent, unique identifier for the check. Note the following about the `ID` property:

- You must specify this property.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Tasks and factory group definitions must refer to checks by `ID`.

ModelAdvisor.FactoryGroup.ID property

Purpose

Identifier for folder

Values

String

Description

The `ID` property specifies a permanent, unique identifier for the folder.

Note

- You must specify this field.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Group definitions must refer to other groups by `ID`.

Purpose

Identifier for folder

Values

String

Description

The `ID` property specifies a permanent, unique identifier for the folder.

Note

- You must specify this field.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Group definitions must refer to other groups by `ID`.

ModelAdvisor.Task.ID property

Purpose

Identifier for task

Values

String

Default: `''` (null string)

Description

The `ID` property specifies a permanent, unique identifier for the task.

Note

- The Model Advisor automatically assigns a string to `ID` if you do not specify it.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Group definitions must refer to tasks using `ID`.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.ID='Task_ID_1234';
```

Purpose

Cell array of product license names

Values

Cell array of product license names

{}(empty cell array) (default)

Description

The `LicenseName` property specifies a cell array of names for product licenses required to enable the check. The Model Advisor does not display the check if you do not meet the license requirements.

Tip To find the correct text for license strings, type `help license` at the MATLAB command line.

ModelAdvisor.Task.LicenseName property

Purpose

Cell array of product license names required to enable check

Values

Cell array of product license names

Default: `{}` (empty cell array)

Description

The `LicenseName` property specifies a cell array of names for product licenses required to enable the check. The Model Advisor does not display the check if you do not meet the license requirements. If you specify `ModelAdvisor.Check.LicenseName`, the Model Advisor displays the check when the union of both properties is true.

Tip To find the correct text for license strings, type `help license` at the MATLAB command line.

Purpose

Status of **Explore Result** button

Values

`false` (default)
`true`

Description

The `ListViewVisible` property is a Boolean value that sets the status of the **Explore Result** button.

`true`	Display the **Explore Result** button.
`false`	Hide the **Explore Result** button.

Example

```
% add 'Explore Result' button
rec.ListViewVisible = true;
```

ModelAdvisor.FactoryGroup.MAObj property

Purpose

Model Advisor object

Values

Handle to a `Simulink.ModelAdvisor` object

Description

The `MAObj` property specifies a handle to the current Model Advisor object.

Purpose

Model Advisor object

Values

Handle to `Simulink.ModelAdvisor` object

Description

The `MAObj` property specifies a handle to the current Model Advisor object.

ModelAdvisor.Task.MAObj property

Purpose

Model Advisor object

Values

Handle to a `Simulink.ModelAdvisor` object

Description

The `MAObj` property specifies the current Model Advisor object.

When adding checks as tasks, the Model Advisor uses the task `MAObj` property instead of the check `MAObj` property.

Purpose

`cv.cvdatagroup` object name

Values

`name`

Description

The `name` property specifies the name of the `cv.cvdatagroup` object.

Examples

```
cvdg = cvsimref(topModelName, cvtg);
cvdg.name = 'My_Data_Group';
```

cv.cvtestgroup.name property

Purpose

`cv.cvtestgroup` object name

Value

`name`

Description

The `name` property specifies the name of the `cv.cvtestgroup` object.

Examples

```
cvto1 = cvtest('TopModel');
cvto2 = cvtest('SubModel1');
cvto3 = cvtest('SubModel2');
cvtg = cv.cvtestgroup(cvto1, cvto2, cvto3);
cvtg.name = 'My_Test_Group';
```

See Also

`cvtest`

Purpose

Action button label

Values

String

Default: `''` (null string)

Description

The `Name` property specifies the label for the action button. This property is required.

Example

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
%Specify a callback function for the action
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
```

ModelAdvisor.InputParameter.Name property

Purpose

Input parameter name

Values

String.

Default: `''` (null string)

Description

The `Name` property specifies the name of the input parameter in the custom check.

Examples

```
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
```

ModelAdvisor.ListViewParameter.Name property

Purpose

Drop-down list entry

Values

String

Default: `''` (null string)

Description

The `Name` property specifies an entry in the **Show** drop-down list in the Model Advisor Result Explorer.

Examples

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
```

ModelAdvisor.Check.Result property

Purpose

Results cell array

Values

Cell array

Default: `{}` (empty cell array)

Description

The `Result` property specifies the cell array for storing the results that are returned by the callback function specified in `CallbackHandle`.

Tip To set the icon associated with the check, use the `Simulink.ModelAdvisor` `setCheckResultStatus` and `setCheckErrorSeverity` methods.

Purpose

Name of check

Values

String

Default: `''` (null string)

Description

The `Title` property specifies the name of the check in the Model Advisor. The Model Advisor displays each custom check in the tree using the title of the check. Therefore, you should specify a unique title for each check. When you specify the same title for multiple checks, the Model Advisor generates a warning.

Example

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
```

ModelAdvisor.Check.TitleTips property

Purpose

Description of check

Values

String

Default: `''` (null string)

Description

The `TitleTips` property specifies a description of the check. Details about the check are displayed in the right pane of the Model Advisor.

Example

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
```

Purpose

Input parameter type

Values

String.

Default: `''` (null string)

Description

The `Type` property specifies the type of input parameter.

Use the `Type` property with the `Value` and `Entries` properties to define input parameters.

Valid values are listed in the following table.

Type	Data Type	Default Value	Description
`Bool`	Boolean	`false`	A check box
`String`	String	`''` (null string)	A text box
`Enum`	Cell array	First entry in the list	A drop-down menu • Use `Entries` to define the entries in the list. • Use `Value` to indicate a specific entry in the list.
`ComboBox`	Cell array	First entry in the list	A drop-down list that allows you to choose a value or enter a value. • Use `Entries` to define the entries in the list. • Use `Value` to indicate a specific entry in the menu or to enter a value not in the list.
`PushButton`	N/A	N/A	A button When you click the button, the callback function

Type	Data Type	Default Value	Description
			specified by `Entries` is called.

Examples

```
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
```

Purpose

Status of check

Values

`'true'` (default)
`'false'`

Description

The `Value` property specifies the initial status of the check.

`'true'`	Check is enabled
`'false'`	Check is disabled

Examples

```
% hide all checks that do not belong to Demo group
if ~(strcmp(checkCellArray{i}.Group, 'Demo'))
    checkCellArray{i}.Visible = false;
    checkCellArray{i}.Value = false;
end
```

ModelAdvisor.InputParameter.Value property

Purpose

Value of input parameter

Values

Depends on the `Type` property.

Description

The `Value` property specifies the initial value of the input parameter. This property is valid only when the `Type` property is one of the following:

- `'Bool'`
- `'String'`
- `'Enum'`
- `'ComboBox'`

Example

```
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
```

Purpose

Status of task

Values

`'true'` (default) — Initial status of task is enabled
`'false'` — Initial status of task is disabled

Description

The `Value` property indicates the initial status of a task—whether it is enabled or disabled.

When adding checks as tasks, the Model Advisor uses the task `Value` property instead of the check `Value` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.Value ='false';
```

ModelAdvisor.Check.Visible property

Purpose

Indicate to display or hide check

Values

`'true'` (default)
`'false'`

Description

The `Visible` property specifies whether the Model Advisor displays the check.

`'true'`	Display the check
`'false'`	Hide the check

Examples

```
% hide all checks that do not belong to Demo group
if ~(strcmp(checkCellArray{i}.Group, 'Demo'))
    checkCellArray{i}.Visible = false;
    checkCellArray{i}.Value = false;
end
```

Purpose

Indicate to display or hide task

Values

`'true'` (default) — Display task in the Model Advisor
`'false'` — Hide task

Description

The `Visible` property specifies whether the Model Advisor displays the task.

Caution

When adding checks as tasks, you cannot specify both the task and check `Visible` properties, you must specify one or the other. If you specify both properties, the Model Advisor generates an error when the check `Visible` property is `false`.

Example

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.Visible ='false';
```

ModelAdvisor.Task.Visible

10

Block Reference

System Requirements

Purpose

List system requirements in Simulink diagrams

Library

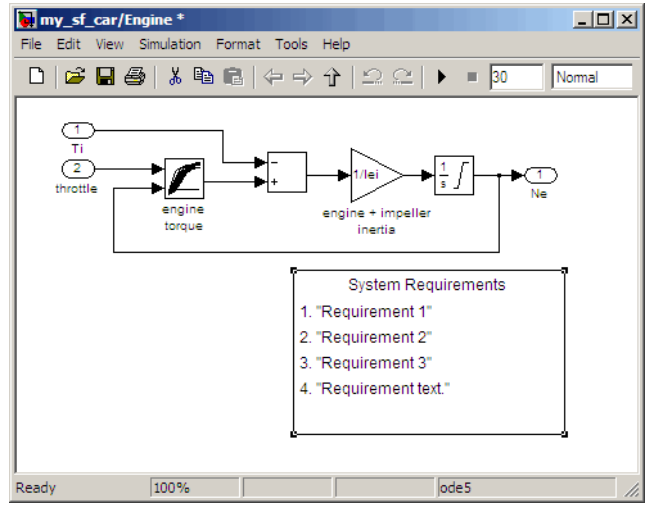
Simulink Verification and Validation

Description

The System Requirements block lists all the system requirements associated with the model or subsystem depicted in the current diagram. It does not list requirements associated with individual blocks in the diagram.

You can place this block anywhere in a diagram. It is not connected to other Simulink blocks. You can only have one System Requirements block in a diagram.

When you drag the System Requirements block from the Library Browser into your Simulink diagram, it is automatically populated with the system requirements, as shown.



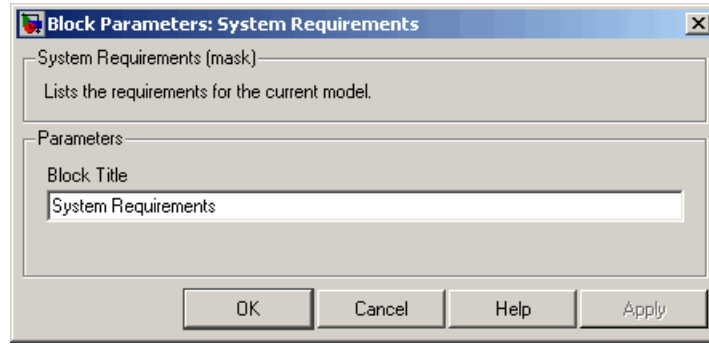
Each of the listed requirements is an active link to the actual requirements document. When you double-click on a requirement name, the associated requirements document opens in its editor window, scrolled to the target location.

If the System Requirements block exists in a diagram, it automatically updates the requirements listing as you add, modify, or delete requirements for the model or subsystem.

For more information on using the System Requirements block, see "Displaying the System Requirements in a Diagram" on page 2-52.

Dialog Box and Parameters

To access the Block Parameters dialog box for the System Requirements block, right-click on the System Requirements block and, from the resulting pop-up menu, select **Mask Parameters**. The Block Parameters dialog box opens, as shown.



The Block Parameters dialog box for the System Requirements block contains one parameter.

Block Title

The title of the system requirements list in the diagram. The default title is `System Requirements`. You can type a customized title, for example, `Engine Requirements`.

11

Model Advisor Checks

Simulink Verification and Validation Checks

In this section...

Simulink Verification and Validation Checks Overview

Simulink Verification and Validation checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements, modeling guidelines, or requirements consistency.

For descriptions of the modeling standards checks, see

- "DO-178B Checks" on page 11-4
- "IEC 61508 Checks" on page 11-60
- "MathWorks Automotive Advisory Board Checks" on page 11-75

For descriptions of the requirements consistency checks, see "Requirements Consistency Checks" on page 11-142.

See Also

- "Consulting the Model Advisor" in the Simulink documentation
- "Simulink Checks" in the Simulink reference documentation
- "Real-Time Workshop® Checks" in the Real-Time Workshop documentation

Modeling Standards Checks Overview

Modeling standards checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements or MathWorks Automotive Advisory Board (MAAB) modeling guidelines.

For descriptions of the modeling standards checks, see

- "DO-178B Checks" on page 11-4
- "IEC 61508 Checks" on page 11-60
- "MathWorks Automotive Advisory Board Checks" on page 11-75

See Also

- Consulting the Model Advisor in the Simulink documentation
- Simulink Checks in the Simulink reference documentation
- Real-Time Workshop Checks in the Real-Time Workshop documentation

DO-178B Checks

In this section...

In this section...

DO-178B Checks Overview

DO-178B checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements.

See Also

- "Consulting the Model Advisor" in the Simulink documentation
- "Simulink Checks" in the Simulink reference documentation
- "Real-Time Workshop Checks" in the Real-Time Workshop documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related optimization settings

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Results and Recommended Actions

Condition	Recommended Action
Block reduction optimization is on. This optimization can remove blocks from generated code, resulting in requirements with no associated code and violations for traceability requirements. (See DO-178B, Section 6.3.4e—Source code is traceable to low-level requirements.)	Clear the **Block reduction** check box on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `BlockReduction` to `off`.
Conditional input branch execution is on. Because the model coverage tool does not account for this optimization, the optimization can result in the tool reporting 100% model coverage while coverage for the code using the same test cases can be less than 100%. (See DO-178B, Section 6.4.4.2—Test coverage of software structure is achieved.)	Clear the **Conditional input branch execution** check box on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `ConditionallyExecuteInputs` to `off`.
Implementation of logic signals as Boolean data is off. Strong data typing is recommended for safety-related code. (See DO-178B, Section 6.3.1e—High-level requirements conform to standards, DO-178B, Section 6.3.2e—Low-level requirements conform to standards, and MISRA C 2004, Rule 12.6.)	Select **Implement logic signals as boolean data (vs. double)** on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `BooleanDataType` to `on`.

Condition	Recommended Action
The model includes blocks that depend on elapsed or absolute time and is configured to minimize the amount of memory allocated for the timers. Such a configuration limits the number of days the application can execute before a timer overflow occurs. Many aerospace products are powered on continuously and timers should not assume a limited lifespan. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 12.11.)	Set **Application lifespan (days)** on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `LifeSpan` to `inf`.
The optimization that ignores integer downcasts in folded expressions is on. This optimization can remove blocks that typecast data from generated code, resulting in requirements with no associated code and violations for traceability requirements. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 10.1.)	Clear the **Ignore integer downcasts in folded expressions** check box on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `EnforceIntegerDowncast` to `off`.
The optimization that suppresses the generation of initialization code for root-level inports and outports that are set to zero is on. For safety-related code, you should explicitly initialize all variables. (See DO-178B, Section 6.3.3b—Software architecture is consistent and MISRA C 2004, Rule 9.1.)	Clear the **Remove root level I/O zero initialization** check box on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `ZeroExternalMemoryAtStartup` to `on`. Alternatively, integrate external, hand-written code that initializes all I/O variables to zero explicitly.

Condition	Recommended Action
The optimization that suppresses the generation of initialization code for internal work structures, such as block states and block outputs that are set to zero, is on. For safety-related code, you should explicitly initialize all variables. (See DO-178B, Section 6.3.3b—Software architecture is consistent and MISRA C 2004, Rule 9.1.)	Clear the **Remove internal data zero initialization** check box on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `ZeroInternalMemoryAtStartup` to `on`. Alternatively, integrate external, hand-written code that initializes all state variables to zero explicitly.
The optimization that suppresses generation of code resulting from floating-point to integer conversions that wrap out-of-range values is off. You must avoid overflows for safety-related code. When this optimization is off and your model includes blocks that disable the **Saturate on overflow** parameter, the code generator wraps out-of-range values for those blocks. This can result in unreachable and, therefore, untestable code. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 12.11.)	Select **Remove code from floating-point to integer conversions that wraps out-of-range values** on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `EfficientFloat2IntCast` to `on`.
The optimization that specifies whether to generate code that guards against division by zero for fixed-point data is on. You must avoid division-by-zero exceptions in safety-related code. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Clear the **Remove code that protects against division arithmetic exceptions** check box on the **Optimization** pane of the Configuration Parameters dialog box or set the parameter `NoFixptDivByZeroProtection` to `off`.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

See Also

- Optimization Pane in the Simulink graphical user interface documentation
- Optimizing a Model for Code Generation in the Real-Time Workshop documentation
- Tips for Optimizing the Generated Code in the Real-Time Workshop Embedded Coder documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for solvers

Check model configuration for diagnostic settings that apply to solvers and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to solvers are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic for detecting automatic breakage of algebraic loops is set to `none` or `warning`. The breaking of algebraic loops can affect the predictability of the order of block execution. For safety-related applications, a model developer needs to know when such breaks occur. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)	Set **Algebraic loop** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box or set the parameter `AlgebraicLoopMsg` to `error`. Consider breaking such loops explicitly with Unit Delay blocks to ensure that execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.
The diagnostic for detecting automatic breakage of algebraic loops for Model blocks, atomic subsystems, and enabled subsystems is set to `none` or `warning`. The breaking of algebraic loops can affect the predictability of the order of block execution. For safety-related applications, a model developer needs to know when such breaks occur. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)	Set **Minimize algebraic loop** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box or set the parameter `ArtificialAlgebraicLoopMsg` to `error`. Consider breaking such loops explicitly with Unit Delay blocks to ensure that execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.

Condition	Recommended Action
The diagnostic for detecting potential conflict in block execution order is set to `none` or `warning`. For safety-related applications, block execution order must be predictable. A model developer needs to know when conflicting block priorities exist. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Block priority violation** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box or set the parameter `BlockPriorityViolationMsg` to `error`.
The diagnostic for detecting whether a model contains an S-function that has not been specified explicitly to inherit sample time is set to `none` or `warning`. These settings can result in unpredictable behavior. A model developer needs to know when such an S-function exists in a model so it can be modified to produce predictable behavior. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)	Set **Unspecified inheritability of sample times** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box or set the parameter `UnknownTslnhSupMsg` to `error`.
The diagnostic for detecting whether the Simulink software automatically modifies the solver, step size, or simulation stop time is set to `none` or `warning`. Such changes can affect the operation of generated code. For safety-related applications, it is better to detect such changes so a model developer can explicitly set the parameters to known values. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)	Set **Automatic solver parameter selection** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box or set the parameter `SolverPrmCheckMsg` to `error`.
The diagnostic for detecting when a name is used for more than one state in the model is set to `none`. State names within a model should be unique. For safety-related applications, it is better to detect name clashes so a model developer can correct them. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **State name clash** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box or set the parameter `StateNameClashWarn` to `warning`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to solvers and that can impact safety.

See Also

- Diagnostics Pane: Solver in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for sample time

Check model configuration for diagnostic settings that apply to sample time and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to sample times are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic for detecting when a source block, such as a Sine Wave block, inherits a sample time (specified as -1) is set to `none` or `warning`. The use of inherited sample times for a source block can result in unpredictable execution rates for the source block and blocks connected to it. For safety-related applications, source blocks should have explicit sample times to prevent incorrect execution sequencing. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)	Set **Source block specifies -1 sample time** on the **Diagnostics > Sample Time** pane of the Configuration Parameters dialog box or set the parameter `InheritedTslnSrcMsg` to `error`.
The diagnostic for detecting whether the input for a discrete block, such as the Unit Delay block, is a continuous signal is set to `none` or `warning`. Signals with continuous sample times should not be used for embedded real-time code. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)	Set **Discrete used as continuous** on the **Diagnostics > Sample Time** pane of the Configuration Parameters dialog box or set the parameter `DiscreteInheritContinuousMsg` to `error`.

Condition	Recommended Action
The diagnostic for detecting invalid rate transitions between two blocks operating in multitasking mode is set to `none` or `warning`. Such rate transitions should not be used for embedded real-time code. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Multitask rate transition** on the **Diagnostics > Sample Time** pane of the Configuration Parameters dialog box or set the parameter `MultiTaskRateTransMsg` to `error`.
The diagnostic for detecting subsystems that can cause data corruption or nondeterministic behavior is set to `none` or `warning`. This diagnostic detects whether conditionally executed multirate subsystems (enabled, triggered, or function-call subsystems) operate in multitasking mode. Such subsystems can corrupt data and behave unpredictably in real-time environments that allow preemption. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Multitask conditionally executed subsystem** on the **Diagnostics > Sample Time** pane of the Configuration Parameters dialog box or set the parameter `MultiTaskCondExecSysMsg` to `error`.
The diagnostic for checking sample time consistency between a Signal Specification block and the connected destination block is set to `none` or `warning`. An over-specified sample time can result in an unpredictable execution rate. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)	Set **Enforce sample times specified by Signal Specification blocks** on the **Diagnostics > Sample Time** pane of the Configuration Parameters dialog box or set the parameter `SigSpecEnsureSampleTimeMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to sample time and that can impact safety.

See Also

- Diagnostics Pane: Sample Time in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for signal data

Check model configuration for diagnostic settings that apply to signal data and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal data are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that specifies how the Simulink software resolves signals associated with `Simulink.Signal` objects in the MATLAB workspace is set to `Explicit and implicit` or `Explicit and warn implicit`. For safety-related applications, model developers should be required to define signal resolution explicitly. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Signal resolution** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `SignalResolutionControl` to `Explicit only`. This provides predictable operation by requiring users to define each signal and block setting that must resolve to `Simulink.Signal` objects in the workspace.
The Product block diagnostic that detects a singular matrix while inverting one of its inputs in matrix multiplication mode is set to `none` or `warning`. Division by a singular matrix can result in numeric exceptions when executing generated code. This is not acceptable in safety-related systems. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Set **Division by singular matrix** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `CheckMatrixSingularityMsg` to `error`.

Condition	Recommended Action
The diagnostic that detects when the Simulink software cannot infer the data type of a signal during data type propagation is set to `none` or `warning`. For safety-related applications, model developers must ensure that all data types are specified correctly. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards, DO-178B and Section 6.3.2e – Low-level requirements conform to standards.)	Set **Underspecified data types** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `UnderSpecifiedDataTypeMsg` to `error`.
The diagnostic that detects whether the value of a signal or parameter is too large to be represented by the signal or parameter's data type is set to `none` or `warning`. Undetected numeric overflows can result in incorrect and unsafe application behavior. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Set **Detect overflow** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `IntegerOverflowMsg` to `error`.
The diagnostic that detects when the value of a block output signal is `Inf` or `NaN` at the current time step is set to `none` or `warning`. When this type of block output signal condition occurs, numeric exceptions can result, and numeric exceptions are not acceptable in safety-related applications. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Set **Inf or NaN block output** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `SignalInfNanChecking` to `error`.

Condition	Recommended Action
The diagnostic that detects Simulink object names that begin with `rt` is set to `none` or `warning`. This diagnostic prevents name clashes with generated signal names that have an `rt` prefix. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards, and DO-178B, Section 6.3.2e – Low-level requirements conform to standards.)	Set **"rt" prefix for identifiers** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `RTPrefix` to `error`.
The diagnostic that detects simulation range checking is set to `none` or `warning`. This diagnostic detects when signals exceed their specified ranges during simulation. Simulink compares the signal values that a block outputs with the specified range and the block data type. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Set **Simulation range checking** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `SignalRangeChecking` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to signal data and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for parameters

Check model configuration for diagnostic settings that apply to parameters and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to parameters are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects when a parameter downcast occurs is set to `none` or `warning`. A downcast to a lower signal range can result in numeric overflows of parameters, resulting in incorrect and unsafe behavior. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Set **Detect downcast** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `ParameterDowncastMsg` to `error`.
The diagnostic that detects when a parameter underflow occurs is set to `none` or `warning`. When the data type of a parameter does not have sufficient resolution, the parameter value is zero instead of the specified value. This can lead to incorrect operation of generated code. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Set **Detect underflow** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `ParameterUnderflowMsg` to `error`.

Condition	Recommended Action
The diagnostic that detects when a parameter overflow occurs is set to `none` or `warning`. Numeric overflows can result in incorrect and unsafe application behavior and should be detected and corrected in safety-related applications. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)	Set **Detect overflow** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `ParameterOverflowMsg` to `error`.
The diagnostic that detects when a parameter loses precision is set to `none` or `warning`. Not detecting such errors can result in a parameter being set to an incorrect value in the generated code. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rules 10.1, 10.2, 10.3, and 10.4.)	Set **Detect precision loss** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `ParameterPrecisionLossMsg` to `error`.
The diagnostic that detects when an expression with tunable variables is reduced to its numerical equivalent is set to `none` or `warning`. This can result in a tunable parameter unexpectedly not being tunable in generated code. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)	Set **Detect loss of tunability** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `ParameterTunabilityLossMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to parameters and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation

- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for data used for debugging

Check model configuration for diagnostic settings that apply to data used for debugging and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to debugging are set optimally for generating code for a safety-related application.

See

- DO-178B, Section 6.3.1e – High-level requirements conform to standards
- DO-178B and Section 6.3.2e – Low-level requirements conform to standards

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that enables model verification blocks is set to `Use local settings` or `Enable all`. Such blocks should be disabled because they are assertion blocks, which are for verification only. Model developers should not use assertions in embedded code.	Set **Model Verification block enabling** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `AssertControl` to `Disable All`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to data used for debugging and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for data store memory

Check model configuration for diagnostic settings that apply to data store memory and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to data store memory are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether the model attempts to read data from a data store in which it has not stored data in the current time step is set to a value other than `Enable all as errors`. Reading data before it is written can result in use of stale data or data that is not initialized.	Set **Detect read before write** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `ReadBeforeWriteMsg` to `Enable all as errors`.
The diagnostic that detects whether the model attempts to store data in a data store after previously reading data from it in the current time step is set to a value other than `Enable all as errors`. Writing data after it is read can result in use of stale or incorrect data.	Set **Detect write after read** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `WriteAfterReadMsg` to `Enable all as errors`.

Condition	Recommended Action
The diagnostic that detects whether the model attempts to store data in a data store twice in succession in the current time step is set to a value other than `Enable all as errors`. Writing data twice in one time step can result in unpredictable data.	Set **Detect write after write** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `WriteAfterWriteMsg` to `Enable all as errors`.
The diagnostic that detects when one task reads data from a Data Store Memory block to which another task writes data is set to `none` or `warning`. Reading or writing data in different tasks in multitask mode can result in corrupted or unpredictable data.	Set **Multitask data store** on the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box or set the parameter `MultiTaskDSMMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to data store memory and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for type conversions

Check model configuration for diagnostic settings that apply to type conversions and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to type conversions are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects Data Type Conversion blocks used where no type conversion is necessary is set to `none`. The Simulink software might remove unnecessary Data Type Conversion blocks from generated code. This warning can result in requirements with no corresponding code. The removal of such blocks need to be detected so model developers can remove the unnecessary blocks explicitly. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)	Set **Unnecessary type conversions** on the **Diagnostics > Type Conversion** pane of the Configuration Parameters dialog box or set the parameter `UnnecessaryDatatypeConvMsg` to `warning`.

Condition	Recommended Action
The diagnostic that detects vector-to-matrix or matrix-to-vector conversions at block inputs is set to `none` or `warning`. When the Simulink software automatically converts between vector and matrix dimensions, unintended operations or unpredictable behavior can occur. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)	Set **Vector/matrix block input conversion** on the **Diagnostics > Type Conversion** pane of the Configuration Parameters dialog box or set the parameter `VectorMatrixConversionMsg` to `error`.
The diagnostic that detects when a 32-bit integer value is converted to a floating-point value is set to `none`. This type of conversion can result in a loss of precision due to truncation of the least significant bits for large integer values. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rules 10.1, 10.2, 10.3, and 10.4.)	Set **32-bit integer to single precision float conversion** on the **Diagnostics > Type Conversion** pane of the Configuration Parameters dialog box or set the parameter `Int32ToFloatConvMsg` to `warning`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to type conversions and that can impact safety.

See Also

- Diagnostics Pane: Type Conversion in the Simulink graphical user interface documentation
- Data Type Conversion block in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for signal connectivity

Check model configuration for diagnostic settings that apply to signal connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal connectivity are set optimally for generating code for a safety-related application.

See

- DO-178B, Section 6.3.1e – High-level requirements conform to standards
- DO-178B,Section 6.3.2e – Low-level requirements conform to standards

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects virtual signals that have a common source signal but different labels is set to `none` or `warning`. This diagnostic pertains to virtual signals only and has no effect on generated code. However, signal label mismatches can lead to confusion during model reviews.	Set **Signal label mismatch** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `SignalLabelMismatchMsg` to `error`.
The diagnostic that detects when the model contains a block with an unconnected input signal is set to `none` or `warning`. This must be detected because code is not generated for unconnected block inputs.	Set **Unconnected block input ports** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `UnconnectedInputMsg` to `error`.

Condition	Recommended Action
The diagnostic that detects when the model contains a block with an unconnected output signal is set to `none` or `warning`. This must be detected because dead code can result from unconnected block output signals.	Set **Unconnected block output ports** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `UnconnectedOutputMsg` to `error`.
The diagnostic that detects unconnected signal lines and unmatched Goto or From blocks is set to `none` or `warning`. This error must be detected because code is not generated for unconnected lines.	Set **Unconnected line** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `UnconnectedLineMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to signal connectivity and that can impact safety.

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- Signal Basics in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for bus connectivity

Check model configuration for diagnostic settings that apply to bus connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to bus connectivity are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether a Model block's root Outport block is connected to a bus but does not specify a bus object is set to `none` or `warning`. For a bus signal to cross a model boundary, the signal must be defined as a bus object to ensure compatibility with higher level models that use a model as a reference model.	Set **Unspecified bus object at root Outport block** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `RootOutportRequireBusObject` to `error`.
The diagnostic that detects whether the name of a bus element matches the name specified by the corresponding bus object is set to `none` or `warning`. This diagnostic prevents the use of incompatible buses in a bus-capable block such that the output names are inconsistent.	Set **Element name mismatch** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `BusObjectLabelMismatch` to `error`.
The diagnostic that detects when some blocks treat a signal as a mux/vector, while other blocks treat the signal as a bus, is set to `none` or `warning`. When the Simulink software automatically converts a muxed signal to a bus, it is possible for	Set **Mux blocks used to create bus signals** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `StrictBusMsg` to `error`. You can use the Model Advisor or the `sl_replace_mux` utility

Condition	Recommended Action
an unintended operation or unpredictable behavior to occur.	function to replace all Mux blocks used as bus creators with a Bus Creator block.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to bus connectivity and that can impact safety.

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- `Simulink.Bus` in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings that apply to function-call connectivity

Check model configuration for diagnostic settings that apply to function-call connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to function-call connectivity are set optimally for generating code for a safety-related application.

DO-178B, Section 6.3.3b – Software architecture is consistent

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects incorrect use of a function-call subsystem is set to `none` or `warning`. If this condition is undetected, incorrect code might be generated.	Set **Invalid function-call connection** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `InvalidFcnCallConMsg` to `error`.
The diagnostic that specifies whether the Simulink software has to compute a function-call subsystem's inputs directly or indirectly while executing the subsystem is set to `Use local settings` or `Disable all`. This diagnostic detects unpredictable data coupling between a function-call subsystem and the subsystem's inputs.	Set **Context-dependent inputs** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box or set the parameter `FcnCallInpInsideContextMsg` to `Enable all`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to function-call connectivity and that can impact safety.

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for compatibility

Check model configuration for diagnostic settings that affect compatibility and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to compatibility are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA C 2004, Rule 9.1.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects when a block has not been upgraded to use features of the current release is set to `none` or `warning`. An S-function written for an earlier version might not be compatible with the current version and generated code could operate incorrectly.	Set **S-function upgrades needed** on the **Diagnostics > Compatibility** pane of the Configuration Parameters dialog box or set the parameter `SFcnCompatibilityMsg` to `error`.
The **Check undefined subsystem initial output** diagnostic is `off`. This diagnostic specifies whether the Simulink software displays a warning if the model contains a conditionally executed subsystem in which a block with a specified initial condition drives an Outport block with an undefined initial condition. A conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.	Set **Check undefined subsystem initial output** on the **Diagnostics > Compatibility** pane of the Configuration Parameters dialog box or set the parameter `CheckSSInitialOutputMsg` to `on`.

Condition	Recommended Action
The diagnostic that detects potential initial output differences from earlier releases is set to `off`. An output of a conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.	Set **Check preactivation output of execution context** on the **Diagnostics > Compatibility** pane of the Configuration Parameters dialog box or set the parameter `CheckExecutionContextPreStartOutputMsg` to `on`.
The diagnostic that detects potential output differences from earlier releases is set to `off`. An output of a conditionally executed subsystem could have an output that is not initialized and feeds into a block with a tunable parameter. If undetected, this condition can cause the behavior of the downstream block to be nondeterministic.	Set **Check runtime output of execution context** on the **Diagnostics > Compatibility** pane of the Configuration Parameters dialog box or set the parameter `CheckExecutionContextRuntimeOutputMsg` to `on`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that affect compatibility and that can impact safety.

See Also

- Diagnosing Simulation Errors in the Simulink documentation
- Diagnostics Pane: Compatibility in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for model referencing

Check model configuration for diagnostic settings that apply to model referencing and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to model referencing are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects a mismatch between the version of the model that creates or refreshes a Model block and the referenced model's current version is set to `none` or `warning`. The detection occurs during load and update operations. Get the latest version of the referenced model from the software configuration management system, rather than using an older version. Using an older version can produce incorrect simulation results and mismatches between simulation and target code operation. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Model block version mismatch** on the **Diagnostics > Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `ModelReferenceVersionMismatchMessage` to `error`.
The diagnostic that detects port and parameter mismatches during model loading and updating is set to `none` or `warning`. If undetected, such mismatches can lead to incorrect simulation results because the parent and referenced models have different interfaces. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Port and parameter mismatch** on the **Diagnostics > Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `ModelReferenceIOMismatchMessage` to `error`.

Condition	Recommended Action
The **Model configuration mismatch** diagnostic is set to `none` or `error`. This diagnostic checks whether the configuration parameters of a model referenced by the current model match the current model's configuration parameters or are inappropriate for a referenced model. Some diagnostics for referenced models are not supported in simulation mode. Setting this diagnostic to `error` can prevent simulations from running. Some differences in configurations can lead to incorrect simulation results and mismatches between simulation and target code generation. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Model configuration mismatch** on the **Diagnostics > Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `ModelReferenceCSMismatchMessage` to `warning`.
The diagnostic that detects invalid internal connections to the current model's root-level Inport and Outport blocks is set to `none` or `warning`. When this condition is detected, the Simulink software might automatically insert hidden blocks into the model to correct the condition. The hidden blocks can result in generated code that has no traceable requirements. Setting the diagnostic to `error` forces model developers to correct the referenced models manually. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Invalid root Inport/Outport block connection** on the **Diagnostics > Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `ModelReferenceIOMessage` to `error`.
The diagnostic that detects whether To Workspace or Scope blocks are logging data in a referenced model is set to `none` or `warning`. Because To Workspace and Scope blocks are for debugging and not for embedded code, it is best to detect the condition so model developers can correct it. (See DO-178B, Section 6.3.1d – High-level requirements are verifiable and DO-178B,	Set **Unsupported data logging** on the **Diagnostics > Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `ModelReferenceDataLoggingMessage` to `error`.

Condition	Recommended Action
Section 6.3.2d – Low-level requirements are verifiable.)	

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to model referencing and that can impact safety.

See Also

- Diagnosing Simulation Errors in the Simulink documentation
- Diagnostics Pane: Model Referencing in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related model referencing settings

Check model configuration for model referencing settings that can impact safety.

Description

This check verifies that model configuration parameters for model referencing are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The referenced model is configured such that its target is rebuilt whenever you update, simulate, or generate code for the model, or if the Simulink software detects any changes in known dependencies. These configuration settings can result in unnecessary regeneration of the code, resulting in changing only the date of the file and slowing down the build process when using model references. (See DO-178B, Section 6.3.1b – High-level requirements are accurate and consistent and DO-178B, Section 6.3.2b – Low-level requirements are accurate and consistent.)	Set **Rebuild options** on the **Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `UpdateModelReferenceTargets` to `Never` or `If any changes detected`.
The diagnostic that detects whether a target needs to be rebuilt is set to `None` or `Warn if targets require rebuild`. For safety-related applications, an error should alert model developers that the parent and referenced models are inconsistent. This diagnostic parameter is available only if **Rebuild options** is set to `Never`. (See DO-178B, Section 6.3.1b – High-level requirements are accurate and consistent and DO-178B, Section 6.3.2b – Low-level requirements are accurate and consistent.)	Set **Never rebuild targets diagnostic** on the **Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `CheckModelReferenceTargetMessage` to `Error if targets require rebuild`.

Condition	Recommended Action
The ability to pass scalar root input by value is on. This capability should be off because scalar values can change during a time step and result in unpredictable data. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Pass scalar root inputs by value** on the **Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `ModelReferencePassRootInputsByReference` to `off`.
The model is configured to minimize algebraic loop occurrences. This configuration is incompatible with the recommended setting of **Single output/update function** for embedded systems code. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set **Minimize algebraic loop occurrences** on the **Model Referencing** pane of the Configuration Parameters dialog box or set the parameter `ModelReferenceMinAlgLoopOccurrences` to `off`.

Action Results

Clicking **Modify Settings** configures model referencing settings that can impact safety.

See Also

- Model Dependencies in the Simulink documentation
- Model Referencing Pane in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related code generation settings

Check model configuration for code generation settings that can impact safety.

Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The option to include comments in the generated code is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Include comments** on the **Real-Time Workshop > Comments** > pane of the Configuration Parameters dialog box or set the parameter `GenerateComments` to `on`.
The option to include comments that describe the code for blocks is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Simulink block / Stateflow object comments** on the **Real-Time Workshop > Comments** pane of the Configuration Parameters dialog box or set the parameter `SimulinkBlockComments` to `on`.
The option to include comments that describe the code for blocks eliminated from a model is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Show eliminated blocks** on the **Real-Time Workshop > Comments** pane of the Configuration Parameters dialog box or set the parameter `ShowEliminatedStatement` to `on`.

Condition	Recommended Action
The option to include the names of parameter variables and source blocks as comments in the model parameter structure declaration in *model*_prm.h is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Verbose comments for SimulinkGlobal storage class** on the **Real-Time Workshop > Comments** pane of the Configuration Parameters dialog box or set the parameter `ForceParamTrailComments` to `on`.
The option to include requirement descriptions assigned to Simulink blocks as comments is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Requirements in block comments** on the **Real-Time Workshop > Comments** pane of the Configuration Parameters dialog box or set the parameter `ReqsInCode` to `on`.
The option to generate nonfinite data and operations is on. Support for nonfinite numbers is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **Support: non-finite numbers** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `SupportNonFinite` to `off`.
The option to generate and maintain integer counters for absolute and elapsed time is on. Support for absolute time is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **Support: absolute time** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `SupportAbsoluteTime` to `off`.

Condition	Recommended Action
The option to generate code for blocks that use continuous time is on. Support for continuous time is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **Support: continuous time** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `SupportContinuousTime` to `off`.
The option to generate code for noninlined S-functions is on. This option requires support of nonfinite numbers, which is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **Support: non-inlined S-functions** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `SupportNonInlinedSFcns` to `off`.
The option to generate model function calls compatible with the main program module of the GRT target is on. This option is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **GRT compatible call interface** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `GRTInterface` to `off`.

Condition	Recommended Action
The option to generate the *model*_update function is off. Having a single call to the output and update functions simplifies the interface to the real-time operating system (RTOS) and simplifies verification of the generated code. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **Single output/update function** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `CombineOutputUpdateFcns` to `on`.
The option to generate the *model*_terminate function is on. This function deallocates dynamic memory, which is not appropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **Terminate function required** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `IncludeMdlTerminateFcn` to `off`.
The option to log or monitor error status is off. If you do not select this option, the Real-Time Workshop product generates extra code that might not be reachable for testing. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **Suppress error status in real-time model data structure** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `SuppressErrorStatus` to `on`.

Condition	Recommended Action
MAT-file logging is enabled. This option adds extra code for logging test points to a MAT-file, which is not supported by embedded targets. Use this option only in test harnesses. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)	Set **MAT-file logging** on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box or set the parameter `MatFileLogging` to `on`.
The option that specifies the style for parenthesis usage is set to `Minimum (Rely on C/C++ operators precedence)` or to `Nominal (Optimize for readability)`. For safety-related applications, explicitly specify precedence with parentheses. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer, DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer, and MISRA C 2004, Rule 12.1.)	Set **Parenthesis level** on the **Real-Time Workshop > Code** pane of the Configuration Parameters dialog box or set the parameter `ParenthesisStyle` to `Maximum (Specify precedence with parentheses)`.
The option that specifies whether to preserve operand order is off. This option increases the traceability of the generated code. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Preserve operand order in expression** on the **Real-Time Workshop > Code** pane of the Configuration Parameters dialog box or set the parameter `PreserveExpressionOrder` to `on`.

Condition	Recommended Action
The option that specifies whether to preserve empty primary condition expressions in `if` statements is off. This option increases the traceability of the generated code. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Preserve condition expression in if statement** on the **Real-Time Workshop > Code** pane of the Configuration Parameters dialog box or set the parameter `PreserveIfCondition` to `on`.
The minimum number of characters specified for generating name mangling strings is less than four. You can use this option to minimize the likelihood that parameter and signal names will change during code generation when the model changes. Use of this option assists with minimizing code differences between file versions, decreasing the effort to perform code reviews. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)	Set **Minimum mangle length** on the **Real-Time Workshop > Symbols** pane of the Configuration Parameters dialog box or set the parameter `MangleLength` to a value of 4 or greater.

Action Results

Clicking **Modify Settings** configures model code generation settings that can impact safety.

See Also

- Real-Time Workshop Pane: Comments in the Real-Time Workshop reference documentation
- Real-Time Workshop Pane: Symbols in the Real-Time Workshop reference documentation
- Real-Time Workshop Pane: Interface in the Real-Time Workshop reference documentation
- Real-Time Workshop Pane: Code Style in the Real-Time Workshop Embedded Coder reference documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for saving

Check model configuration for diagnostic settings that apply to saving model files

Description

This check verifies that model configuration parameters are set optimally for saving a model for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether a model contains disabled library links before the model is saved is set to `none` or `warning`. If this condition is undetected, incorrect code might be generated. (See DO-178B, Section 6.3.3b - Software architecture is consistent.)	Set **Block diagram contains disabled library links** on the **Diagnostics > Saving**> pane of the Configuration Parameters dialog box or set the parameter `SaveWithDisabledLinkMsg` to `error`.
The diagnostic that detects whether a model contains library links that are using parameters not in a mask before the model is saved is set to `none` or `warning`. If this condition is undetected, incorrect code might be generated. (See DO-178B, Section 6.3.3b - Software architecture is consistent.)	Set **Block diagram contains parameterized library links** on the **Diagnostics > Saving**> pane of the Configuration Parameters dialog box or set the parameter `SaveWithParameterizedLinkMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to saving a model file.

See Also

- Disabling Links to Library Blocks in the Simulink documentation
- Identifying Disabled Library Links in the Simulink documentation
- Saving a Model in the Simulink documentation

- Model Parameters in the Simulink documentation
- Diagnostics Pane: Saving in the Simulink documentation

Check for blocks that do not link to requirements

Check whether blocks link to a requirements document.

Description

This check verifies that functional blocks link to a document with engineering requirements for traceability.

Analysis Results and Recommended Actions

Condition	Recommended Action
Blocks do not link to a requirements document. (See DO-178B, Section 6.3.1f - High-level requirements trace to system requirements, Section 6.3.2f - Low-level requirements trace to high-level requirements.)	Link to requirements document. See "Adding Requirement Links to an Object" on page 2-7.

Limitations

When you run this check, the Model Advisor does not follow library links or look under masks. The Model Advisor reviews all top-level blocks in the system of interest.

Tips

Run this check from the top level model or subsystem that you want to check.

See Also

Chapter 2, "Managing Model Requirements"

Check for proper usage of Math blocks

Check whether math operators require nonfinite number support.

Description

This check verifies that Math Function blocks do not use math operations that need nonfinite number support with real-time embedded targets.

Analysis Results and Recommended Actions

Condition	Recommended Action
Math Function blocks using `log` (natural logarithm), `log10` (base 10 logarithm), and `rem` (Remainder) operators that require nonfinite number support. (See DO-178B, Section 6.3.1g - Algorithms are accurate, Section 6.3.2g - Algorithms are accurate, and MISRA 2004, Rule 21.1)	When using the Math Function block with a `log` or `log10` function, you must protect the input to the block in the model such that it is not less then or equal to zero. Otherwise, the output can produce a `NaN` or `Inf` and result in a run-time error in the generated code. When using the Math Function block with a `rem` function, you must protect the second input to the block such that it is not equal to zero. Otherwise the output can produce a `Inf` or `Inf` and result in a run-time error in the generated code.

Tips

With embedded systems, you must take care when using blocks that could produce nonfinite outputs such as `NaN`, `Inf` or `Inf`. Your design must protect the inputs to these blocks in order to avoid run-time errors in the embedded system.

See Also

Math Function block in the Simulink documentation

Check for proper usage of For Iterator blocks

Check for For Iterator blocks that have variable loops.

Description

This check verifies that a model does not use variable loops with For Iterator blocks.

See

- DO-178B Section 6.3.1e – High-level requirements conform to standards
- DO-178B Section 6.3.2e – Low-level requirements conform to standards
- MISRA C 2004, Rule 13.6

Results and Recommended Actions

Condition	Recommended Action
The model combines the use of variable iteration values with a For Iterator block. The use of variable `for` loops can lead to unpredictable execution time and, in the case of external iteration variables, infinite loops.	To avoid the use of variable `for` loops, do one of the following: • Set the **Iteration limit source** parameter of the For Iterator block to `internal`. • If the **Iteration limit source** parameter of the For Iterator block must be `external`, use a Constant, Probe, or Width block as the source. • Avoid selecting the **Set next i (iteration variable) externally** parameter of the For Iterator block.

See Also

- For Iterator Subsystem block in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check for proper usage of While Iterator blocks

Check for While Iterator blocks that cause infinite loops.

Description

This check verifies that a model does not include infinite loops with While Iterator blocks.

See

- DO-178B Section 6.3.1e – High-level requirements conform to standards
- DO-178B Section 6.3.2e – Low-level requirements conform to standards
- MISRA C 2004, Rule 21.1

Results and Recommended Actions

Condition	Recommended Action
The model combines the use of a While Iterator block with an unlimited number of iterations. An unlimited number of iterations can lead to infinite loops in real-time code, which can lead to execution time overruns.	To avoid infinite loops: • Set the **Maximum number of iterations** parameter of the While Iterator block to a positive integer value. • Consider selecting the **Show iteration number port** parameter of the While Iterator block and observe the iteration value during simulation to determine whether the maximum number of iterations is being reached. If the loop reaches the maximum number of iterations, verify whether the output values of the While Iterator block are correct.

See Also

- While Iterator Subsystem block in the Simulink reference documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Display model version information

Display model version information in your report.

Description

This check displays the following information for the current model:

- Version number
- Author
- Date
- Model checksum

Results and Recommended Actions

Condition	Recommended Action
Could not retrieve model version and checksum information.	This summary is provided for your information. No action is required.

See Also

- Validating Generated Code in the Real-Time Workshop documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check for proper usage of blocks that compute absolute values

Check for absolute value blocks that have unreachable code or produce overflows.

Description

This check verifies whether the model includes a block that attempts to compute the absolute value of a Boolean or unsigned integer value.

See

- DO-178B Section 6.3.1d – High-level requirements are verifiable
- DO-178B Section 6.3.2d – Low-level requirements are verifiable
- DO-178B Section 6.3.1g – Algorithms are accurate
- DO-178B Section 6.3.2g – Algorithms are accurate
- MISRA C 2004, Rule 14.1
- MISRA C 2004, Rule 21.1

Results and Recommended Actions

Condition	Recommended Action
The model includes a block that: • Computes an absolute value and the input signal of the block is a Boolean value or an unsigned integer. Use of Boolean and unsigned data types might result in code that is unreachable and cannot be tested. • Computes an absolute value of a signed integer and **Saturate on integer overflow** is not selected for that block. Taking the absolute value of full scale negative integer value results in an overflow.	• To avoid unreachable code, change the input to the Absolute Value block to a signed input type. • To avoid overflows, select the **Saturate on integer overflow** check box of the Absolute Value block.

See Also

- Abs block in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check for proper usage of Relational Operator blocks

Check for relational operator blocks that compare data types or equate floating-point types.

Description

This check verifies that a model does not use the == or ~= operator with a relational operator block to compare floating-point signals.

See

- DO-178B Section 6.3.1g – Algorithms are accurate
- DO-178B Section 6.3.2g – Algorithms are accurate
- MISRA C 2004, Rule 12.6
- MISRA C 2004, Rule 13.3

Results and Recommended Actions

Condition	Recommended Action
The model includes a relational operator block that uses the == or ~= operator to compare floating-point signals. Because of floating-point precision issues, the use of these operators on floating-point signals is unreliable.	Change the data type of the signal or rework the model to eliminate the need to use the relational operator block with the == or ~= operator.

See Also

Descriptions of the following blocks in the Simulink reference documentation

- Relational Operator block in the Simulink reference documentation
- Compare To Constant block in the Simulink documentation
- Compare To Zero block in the Simulink documentation
- Detect Change block in the Simulink documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

IEC 61508 Checks

In this section...

IEC 61508 Checks Overview

IEC 61508 checks facilitate designing and troubleshooting Simulink models and subsystems and the code that you generate from it for applications that need to comply with IEC 61508-3.

Tip

If your model uses model referencing, run the IEC 61508 checks on all referenced models before running them on the top-level model.

See Also

- IEC 61508–3 Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 3: Software requirements
- Developing Models and Code That Comply with the IEC 16508 Standard in the Real-Time Workshop Embedded Coder documentation
- "Consulting the Model Advisor" in the Simulink documentation
- "Simulink Checks" in the Simulink reference documentation
- "Real-Time Workshop Checks" in the Real-Time Workshop documentation

Display model metrics and complexity report

Display number of elements and name, level, and depth of subsystems for the model or subsystem.

Description

The IEC 61508 standard recommends the usage of size and complexity metrics to assess the software under development. This check provides model metrics information for the model. The provided information can be used to inspect whether the size or complexity of the model or subsystem exceeds given limits. The check displays:

- A block count for each Simulink block type contained in the given model.
- The maximum subsystem depth of the given model.
- A count of Stateflow constructs in the given model (if applicable).
- Name, level, and depth of the subsystems contained in the given model (if applicable).

See IEC 61508-3, Table A.9 (5) – Software complexity metrics.

Results and Recommended Actions

Condition	Recommended Action
N/A	This summary is provided for your information. No action is required.

See Also

- `sldiagnostics` in the Simulink documentation
- Cyclomatic Complexity in the Stateflow documentation

Check for unconnected objects

Identify unconnected lines, input ports, and output ports in the model.

Description

Unconnected objects are likely to cause problems propagating signal attributes such as data, type, sample time, and dimensions.

Ports connected to Ground or Terminator blocks pass this check.

See IEC 61508-3, Table A.3 (3) — Language subset.

Results and Recommended Actions

Condition	Recommended Action
There are unconnected lines, input ports, or output ports in the model or subsystem.	• Double-click an element in the list of unconnected items to locate the item in the model diagram. • Properly connect the objects identified in the results.

See Also

"Working with Signals" in the Simulink documentation

Check for fully defined interface

Identify root model Inport blocks that do not have fully defined attributes.

Description

Using root model Inport blocks that do not have fully define dimensions, sample time, or data type can lead to undesired simulation results. Simulink back-propagates dimensions, sample times, and data types from downstream blocks unless you explicitly assign these values.

See IEC 61508-3, Table B.9 (5) – Fully defined interface.

Results and Recommended Actions

Condition	Recommended Action
The model has root-level Inport blocks that have undefined attributes, such as an inherited sample time, data type, or port dimension.	Explicitly define all root-level Inport block attributes identified in the results. Double-click an element from the list of underspecified items to locate the condition.

Tip

The following configurations pass this check:

- Inport blocks with inherited port dimensions in conjunction with the usage of bus objects
- Inport blocks with automatically inherited data types in conjunction with bus objects
- Inport blocks with inherited sample times in conjunction with the **Periodic sample time constraint** menu set to `Ensure sample time independent`

See Also

- Working with Data Types in the Simulink documentation
- Determining Output Signal Dimensions in the Simulink documentation

- Specifying Sample Time in the Simulink documentation

Check for questionable constructs

Identify blocks not supported by code generation or not recommended for deployment.

Description

This check partially identifies model constructs that are not suited for code generation or not recommended for production code generation as identified in the Simulink Block Support tables for Real-Time Workshop and Real-Time Workshop Embedded Coder. If you are using blocks with support notes for code generation, review the information and follow the given advice.

See IEC 61508-3, Table A.3 (3) – Language subset.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for code generation.	Consider replacing the blocks listed in the results. Double-click an element from the list of questionable items to locate condition.
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Double-click an element from the list of questionable items to locate condition.
The model or subsystem contains Gain blocks whose value equals 1.	If you are using Gain blocks as buffers, consider replacing them with Signal Conversion blocks. Double-click an element from the list of questionable items to locate condition.

Limitation

This check might not identify all instances of noncompliance with the Real-Time Workshop and Real-Time Workshop Embedded Coder Simulink Block Support tables.

See Also

- “Simulink Block Support” tables in the Real-Time Workshop documentation for Real-Time Workshop and Real-Time Workshop Embedded Coder
- “Requirements and Restrictions for ERT-Based Simulink Models” in the Real-Time Workshop Embedded Coder documentation

Check usage of Stateflow constructs

Identify usage of Stateflow constructs that might impact safety.

Description

This check identifies instances of Stateflow software being used in a way that can impact an application's safety, including

- Use of strong data typing
- Port name mismatches
- Scope of data objects and events
- Formatting of state action statements

See

- IEC 61508-3, Table A.3 (2) – Strongly typed programming language
- IEC 61508-3, Table A.3 (3) – Language subset
- IEC 61508-3, Table B.9 (2) – Information hiding/encapsulation
- MISRA C:2004, Rule 10.1
- MISRA C:2004, Rule 10.2
- MISRA C:2004, Rule 10.3
- MISRA C:2004, Rule 10.4
- MAAB Control Algorithm Modeling Guidelines, db_0122: Stateflow and Simulink interface signals and parameters
- MAAB Control Algorithm Modeling Guidelines, db_0123: Stateflow port names
- MAAB Control Algorithm Modeling Guidelines, db_0125: Scope of internal signals and local auxiliary variables
- MAAB Control Algorithm Modeling Guidelines, db_0126: Scope of Events
- MAAB Control Algorithm Modeling Guidelines, jc_0501: Format of entries in a state block

Results and Recommended Actions

Condition	Recommended Action
A Stateflow chart is not configured for strong data typing on boundaries between a Simulink model and theStateflow chart.	Enable the option **Use Strong Data Typing with Simulink I/O** for the Stateflow chart. When you enable this option, the Stateflow chart accepts input signals of any data type that Simulink models support, provided that the type of the input signal matches the type of the corresponding Stateflow input data object.
Signals have names that differ from those of their corresponding Stateflow ports.	• Check whether the ports are connected properly and, if not, correct the connections. • Change the names of the signals or the Stateflow ports so that the names match.
Events are not defined in the Stateflow hierarchy at the chart level or below.	Define events at the chart level or below.
Local data is not defined in the Stateflow hierarchy at the chart level or below.	Define local data at the chart level or below.
A new line is missing from a state action after • An `entry` (`en`), `during` (`du`), or `exit` (`ex`) statement • The semicolon (;) at the end of an assignment statement	Add missing new lines.

See Also

See the following topics in the Stateflow documentation

- “Strong Data Typing with Simulink I/O”
- “Property Fields”
- “Defining Events”
- “Defining Data”
- “Labeling States”

Display configuration management data

Display model configuration and checksum information.

Description

This informer check displays the following information for the current model:

- Model version number
- Model author
- Date
- Model checksum

See IEC 61508-3, Table A.8 (5) – Software configuration management.

Results and Recommended Actions

Condition	Recommended Action
Could not retrieve model version and checksum information.	This summary is provided for your information. No action is required.

See Also

- "How Simulink Helps You Manage Model Versions" in the Simulink documentation
- Model Change Log in the Simulink® Report Generator™ documentation
- Simulink.BlockDiagram.getChecksum in the Simulink documentation
- Simulink.SubSystem.getChecksum in the Simulink documentation

Check usage of Simulink constructs

Identify usage of Simulink constructs that might impact safety.

Description

Blocks that you use incorrectly can result in unreachable code, incorrect or unpredictable results, infinite loops, and unpredictable execution times in generated code.

This check inspects your model for proper usage of:

- Abs blocks
- Blocks that compute relational operators including Relational Operator, Compare To Constant, Compare To Zero, and Detect Change blocks
- While Iterator blocks
- For Iterator blocks

See

- IEC 61508-3, Table A.3 (2) – Strongly typed programming language
- IEC 61508-3, Table A.3 (3) – Language subset
- IEC 61508-3, Table A.4 (3) – Defensive programming
- IEC 61508-3, Table B.8 (3) – Control Flow Analysis
- MISRA C:2004, Rule 13.3
- MISRA C:2004, Rule 13.6
- MISRA C:2004, Rule 14.1
- MISRA C:2004, Rule 21.1

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains an Abs block that is operating on a Boolean or an unsigned input data type. This condition results in unreachable simulation pathways through the model and might result in unreachable code.	• Change the input of the Abs block to a signed input type. • Remove the Absfrom the model.
The model or subsystem contains an Abs block that is operating on a signed integer value, and the **Saturate on integer overflow** check box is cleared. For signed data types, the absolute value of the most negative value is problematic since it is not representable by the data type. This condition results in an overflow in the generated code.	Select the **Saturate on integer overflow** check box of the specified Abs blocks.
The model or subsystem contains a block computing a relational operator that is operating on different data types. The condition can lead to unpredictable results in the generated code.	For the specified blocks, use common data types as inputs.
The model or subsystem contains a block computing a relational operator that is not generating Boolean data as its output. This condition violates strong data typing rules and can lead to unpredictable results in the generated code.	Set the **Output data type** to `boolean` in the **Block Parameters > Signal Attributes** pane for the specified blocks.

Condition	Recommended Action
The model or subsystem contains a block computing a relational operator that uses the == or ~= operator to compare floating-point signals. The use of these operators on floating-point signals is unreliable and unpredictable because of floating-point precision issues, and can lead to unpredictable results in the generated code.	For the specified blocks, do one of the following: • Change the signal data type. • Rework the model to eliminate the need to use == or ~= operators on floating-point signals.
The model or subsystem contains a While Iterator block that has unlimited iterations. This condition can lead to infinite loops in the generated code.	For the specified While Iterator blocks: • Set the **Maximum number of iterations (-1 for unlimited)** parameter to a positive integer value. • Consider selecting the **Show iteration number port** check box and observe the iteration value during simulation.
The model or subsystem contains a For Iterator block that has variable iterations. This condition can lead to unpredictable execution times or infinite loops in the generated code.	For the specified For Iterator blocks, do one of the following: • Set the **Iteration limit source** parameter to `internal`. • If the **Iteration limit source** parameter must be `external`, use a Constant, Probe, or Width block as the source. • Clear the **Set next i (iteration variable) externally** check box. • Consider selecting the **Show iteration variable** check box and

Condition	Recommended Action
	observe the iteration value during simulation.

See Also

Descriptions of the following blocks in the Simulink reference documentation:

- Abs block
- Relational Operator block
- Compare To Constant block
- Compare To Zero block
- Detect Change block
- While Iterator block
- For Iterator block

MathWorks Automotive Advisory Board Checks

In this section...

In this section...

In this section...

MathWorks Automotive Advisory Board Checks Overview

MathWorks Automotive Advisory Board (MAAB) checks facilitate designing and troubleshooting models from which code is generated for automotive applications.

See Also

- "Consulting the Model Advisor" in the Simulink documentation
- "Simulink Checks" in the Simulink reference documentation
- "Real-Time Workshop Checks" in the Real-Time Workshop documentation
- The MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow – Version 2.0

Check for difference in font and font sizes

Check for difference in font and font sizes.

Description

With the exception of free text annotations within a model, text elements, such as block names, block annotations, and signal labels, must have the same font style and font size. Select a font style and font size that is legible and portable (convertible between platforms), such as Arial or Helvetica 12 point.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline db_0043: Simulink font and font size.

Input Parameters

Font Name

Apply the specified font to all text elements. Available fonts include `Helvetica` (default), `Arial`, `Arial Black`, `Mangal`, or `Modern`.

Font Size

Apply the specified font size to all text elements. Available sizes include `-1`, `6`, `8`, `9`, `10` (default), `12`, `14`, `16`, `18`, `20`, `22`, and `24`.

Font Angle

Apply the specified font angle to all text elements. Available angles include `auto` (default), `normal`, `italic`, and `oblique`.

Font Weight

Apply the specified font weight to all text elements. Available weights include `auto` (default), `normal`, `light`, `demi` , and `bold`.

Results and Recommended Actions

Condition	Recommended Action
The fonts or font sizes for text elements in the model are not consistent or portable.	Specify values for the font parameters and click **Modify all Fonts**, or manually change the fonts and font sizes of text elements in the model such that they are consistent and portable.

Action Results

Clicking **Modify all Fonts** changes the font and font size of all text elements in the model according to the values you specify for the font parameters.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check transition orientations in flow charts

Check transition orientations in flow charts.

Description

The following rules apply to transitions in flow charts:

- Draw transition conditions horizontally.
- Draw transitions with a condition action vertically.

Loop constructs are exceptions to these rules.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0132: Transitions in Flowcharts.

Results and Recommended Actions

Condition	Recommended Action
The model includes a transition with a condition that is not drawn horizontally or a transition action that is not drawn vertically.	Modify the model.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for display of nondefault block attributes

Check for display of nondefault block attributes.

Description

Model diagrams should display block parameters that have values other than default values. One way of displaying this information is by using the **Block Annotation** tab in the Block Properties dialog box.

This guideline facilitates

- Readability
- Verification and validation

See MAAB guideline db_0140: Display of basic block parameters.

Results and Recommended Actions

Condition	Recommended Action
Block parameters that have values other than default values do not appear in the model display.	Use the **Block Annotation** tab in the Block Properties dialog to add block parameter annotations.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for proper labeling on signal lines

Check for proper labeling on signal lines.

Description

You should use a label to identify:

- Signals originating from the following blocks (the block icon exception noted below applies to all blocks listed except Inport, Bus Selector, Demux, and Selector):

  Bus Selector block (tool forces labeling)
  Chart block (Stateflow)
  Constant block
  Data Store Read block
  Demux block
  From block
  Inport block
  Selector block
  Subsystem block

  Block Icon Exception If a signal label is visible in the display of the icon for the originating block, you do not have to display a label for the connected signal unless the signal label is needed elsewhere due to a rule for signal destinations.

- Signals connected to one of the following destination blocks (directly or indirectly with a basic block that performs an operation that is not transformative):

  Bus Creator block
  Chart block (Stateflow)
  Data Store Write block
  Goto block
  Mux block
  Outport block
  Subsystem block

- Any signal of interest.

This guideline facilitates

- Readability
- Workflow
- Verification and validation
- Code generation

See MAAB guideline na_0008: Display of labels on signals.

Results and Recommended Actions

Condition	Recommended Action
Signals coming from Bus Selector, Chart, Constant, Data Store Read, Demux, From, Inport, or Selector blocks are not labeled.	Double-click the line that represents the signal. After the text cursor appears, enter a name and click anywhere outside the label to exit label editing mode.

See Also

- Signal Labels in the Simulink documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for propagated labels on signal lines

Check for propagated labels on signal lines.

Description

You should propagate a signal label from its source rather than enter the signal label explicitly (manually) if the signal originates from:

- An Inport block in a nested subsystem. However, if the nested subsystem is a library subsystem, you can explicitly label the signal coming from the Inport block to accommodate reuse of the library block.
- A basic block that performs a nontransformative operation.
- A Subsystem or Stateflow Chart block. However, if the connection originates from the output of an instance of the library block, you can explicitly label the signal to accommodate reuse of the library block.

This guideline facilitates

- Readability
- Workflow
- Verification and validation
- Code generation

See MAAB guideline na_0009: Entry versus propagation of signal labels.

Results and Recommended Actions

Condition	Recommended Action
The model includes signal labels that were entered explicitly, but should be propagated.	Use the open angle bracket (<) character to mark signal labels that should be propagated and remove the labels that were entered explicitly.

See Also

- Signal Labels in the Simulink documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check default transition placement in Stateflow charts

Check default transition placement in Stateflow charts.

Description

In a Stateflow chart, you should connect the default transition at the top of the state and place the destination state of the default transition above other states in the hierarchy.

Properly position the default transition and its destination state for:

- Readability

See MAAB guideline jc_0531: Placement of the default transition.

Results and Recommended Actions

Condition	Recommended Action
The default transition for a Stateflow chart is not connected at the top of the state.	Move the default transition to the top of the state chart.
The destination state of a Stateflow chart's default transition is lower than other states in the same hierarchy.	Adjust the position of the default transition's destination state such that the state is above other states in the same hierarchy.

See Also

- "Defining Transitions Between States" in the Stateflow documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check setting Stateflow graphical function return value

Check setting Stateflow graphic function return value.

Description

The return value from a Stateflow graphical function must be set in only one place.

This guideline facilitates

- Workflow
- Code generation

See MAAB guideline jc_0511: Setting the return value from a graphical function.

Results and Recommended Actions

Condition	Recommended Action
The return value from a Stateflow graphical function is set in multiple places.	Modify the function such that its return value is set in one place.

See Also

- “Graphical Functions” in the Stateflow documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for blocks not using one-based indexing

Check for blocks that do not use one-based indexing.

Description

One-based indexing ([1, 2, 3,...]) is used for the following:

Product	Items
MATLAB	• Workspace variables and structures • Local variables of MATLAB functions • Global variables
Simulink	• Signal vectors and matrices • Parameter vectors and matrices • S-function input and output signal vectors and matrices in M-code • S-function parameter vectors and matrices in M-code • S-function local variables in M-code
Stateflow	• Input and output signal vectors and matrices • Parameter vectors and matrices • Local variables

Zero-based indexing ([0, 1, 2, ...]) is used for the following:

Product	Items
Simulink	• Signal vectors and matrices • S-function input and output signal vectors and matrices in C code • S-function input parameters in C code • S-function parameter vectors and matrices in C code • S-function local variables in C code
Stateflow	• Variables and structures in custom C code
C code	• Local variables and structures • Global variables

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline db_0112: Indexing.

Results and Recommended Actions

Condition	Recommended Action
Blocks in your model are not configured for one-based indexing.	Using block parameters, configure all blocks for one-based indexing.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for invalid file names

Check for files residing in the same directory as the model that have illegal file names.

Description

This guideline facilitates

- Readability
- Workflow

See MAAB guideline .

Results and Recommended Actions

Condition	Recommended Action
The file name contains illegal characters.	Rename the file. Allowed characters are a–z, A–Z, 0–9. and underscore (_).
The file name starts with a number.	Rename the file.
The file name starts with an underscore ("_").	Rename the file.
The file name ends with an underscore ("_").	Rename the file.
The file extension contains one (or more) underscores.	Change the file extension.
The file name has consecutive underscores.	Rename the file.
The file name contains more than one dot (".").	Rename the file.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for invalid model directory names

Checks model directory and subdirectory names for invalid characters.

Description

This guideline facilitates

- Readability
- Workflow

See MAAB guideline .

Results and Recommended Actions

Condition	Recommended Action
The directory name contains illegal characters.	Rename the directory. Allowed characters are a–z, A–Z, 0–9. and underscore (_).
The directory name starts with a number.	Rename the directory.
The directory name starts with an underscore ("_").	Rename the directory.
The directory name ends with an underscore ("_").	Rename the directory.
The directory name has consecutive underscores.	Rename the directory.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for blocks that are not discrete

Check for blocks that are not discrete.

Description

You cannot include continuous blocks in controller models.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jm_0001: Prohibited Simulink standard blocks inside controllers.

Results and Recommended Actions

Condition	Recommended Action
Continuous blocks — Derivative, Integrator, State-Space, Transfer Fcn, Transfer Delay, Variable Time Delay, Variable Transport Delay, and Zero-Pole — are not permitted in models representing discrete controllers.	Replace continuous blocks with the equivalent blocks discretized in the s-domain by using the Discretizing library, as explain in "How to Discretize Blocks from the Simulink Model" in the Simulink documentation.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for prohibited sink blocks

Check for prohibited Simulink sink blocks.

Description

You must design controller models from discrete blocks. Sink blocks, such as the Scope block, are not allowed.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline hd_0001: Prohibited Simulink sinks.

Results and Recommended Actions

Condition	Recommended Action
Sink blocks are not permitted in discrete controllers.	Remove sink blocks from the model.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for invalid port positioning and configuration

Check whether the model contains ports with invalid position and configuration.

Description

In models, ports must comply with the following rules:

- Place Inport blocks on the left side of the diagram. Move the Inport block right only to prevent signal crossings.
- Place Outport blocks on the right side of the diagram. Move the Outport block left only to prevent signal crossings.
- Avoid using duplicate Inport blocks at the subsystem level if possible.
- Do not use duplicate Inport blocks at the root level.

This guideline facilitates

- Readability

See MAAB guideline db_0042: Port block in Simulink models.

Results and Recommended Actions

Condition	Recommended Action
Inport blocks are too far to the right and result in left-flowing signals.	Move the specified Inport blocks to the left.
Outport blocks are too far to the left and result in right-flowing signals.	Move the specified Output blocks to the right.

Condition	Recommended Action
Ports do not have the default orientation.	Modify the model diagram such that signal lines for output ports enter the side of the block and signal lines for input ports exit the right side of the block.
Ports are duplicate Inport blocks.	• If the duplicate Inport blocks are in a subsystem, remove them where possible. • If the duplicate Inport blocks are at the root level, remove them.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for mismatches between names of ports and corresponding signals

Check for mismatches between names of ports and corresponding signals.

Description

Use matching names for ports and their corresponding signals.

This guideline facilitates

- Readability
- Workflow
- Simulation

See MAAB guideline jm_0010: Port block names in Simulink models.

Results and Recommended Actions

Condition	Recommended Action
Ports have names that differ from their corresponding signals.	Change the port name or the signal name to match the correct name for the signal.

Limitations

Prerequisite MAAB guidelines for this check are:

- db_0042: Port block in Simulink models
- na_0005: Port block name visibility in Simulink models

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether block names do not appear below blocks

Check whether block names do not appear below blocks.

Description

If shown, the name of all blocks should appear below the blocks.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline db_0142: Position of block names.

Results and Recommended Actions

Condition	Recommended Action
Blocks have names that do not appear below the blocks.	Set the name of the block to appear below the blocks.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for systems that mix primitive blocks and subsystems

Check for systems that mix primitive blocks and subsystems.

Description

You must design every level of a model with building blocks of the same type, for example, only subsystems or only primitive (basic) blocks.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0143: Similar block types on the model levels.

Results and Recommended Actions

Condition	Recommended Action
A level in the model includes both subsystem blocks and primitive blocks.	• Move nonvirtual blocks into the subsystem. • If possible, replace blocks at the identified level of the model hierarchy with blocks that you can place at any module level. Such blocks include Inport, Outport, Enable (not at highest model level), Trigger (not at highest model level), Mux, Demux, Bus Selector, Bus Creator, Selector, Ground, Terminator, From, Goto, Switch, Multiport Switch, Merge, Unit Delay, Rate Transition, Type Conversion, Data Store Memory, If, and Switch Case.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether model has unconnected block input ports, output ports, or signal lines

Check whether model has unconnected input ports, output ports, or signal lines.

Description

All unconnected inputs should be connected to ground blocks. All unconnected outputs should be connected to terminator blocks. Respecting the guideline eliminates error messages.

See MAAB guideline db_0081: Unconnected signals, block inputs and block outputs.

Results and Recommended Actions

Condition	Recommended Action
Blocks have unconnected inputs or outputs.	Connect unconnected lines to blocks specified by the design or to Ground or Terminator blocks.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for improperly positioned Trigger and Enable blocks

Check for improperly positioned Trigger and Enable blocks.

Description

Locate blocks that define subsystems as conditional or iterative at the top of the subsystem diagram.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0146: Triggered, enabled, conditional Subsystems.

Results and Recommended Actions

Condition	Recommended Action
Trigger , Enable, and Action Port blocks are not centered in the upper third of the model diagram.	Move the Trigger, Enable, and Action Port blocks to the correct area of the model diagram.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether annotations have drop shadows

Check whether annotations have drop shadows.

Description

Annotations should not have a drop shadow for readability.

This guideline facilitates

- Readability

See MAAB guideline jm_0013: Annotations.

Results and Recommended Actions

Condition	Recommended Action
Annotations display drop shadows.	Clear the **Format > Show Drop Shadow** menu option.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether tunable parameters specify expressions, data type conversions, or indexing operations

Check whether tunable parameters specify expressions, data type conversions, or indexing operations.

Description

To ensure that a parameter is tunable, you must enter the basic block without the use of MATLAB calculations or scripting. For example, omit

- Expressions
- Data type conversions
- Selections of rows or columns

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline db_0110: Tunable parameters in basic blocks.

Results and Recommended Actions

Condition	Recommended Action
Blocks have a tunable parameter that specifies an expression, data type conversion, or indexing operation.	In each case, move the calculation outside of the block, for example, by performing the calculation with a series of Simulink blocks, or precompute the value in the base workspace as a new variable.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether Stateflow events are defined at the chart level or below

Check whether Stateflow events are defined at the chart level or below.

Description

All events of a Stateflow chart must be defined at the chart level or lower. Events cannot be at the machine level; that is, charts cannot interact with local events.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0126: Scope of events.

Results and Recommended Actions

Condition	Recommended Action
An event in a chart is not defined at the chart level or below.	Define the event at the chart level or below.

See Also

- "Defining Events" in the Stateflow documentation.
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether Stateflow data objects with local scope are defined at the chart level or below

Check whether Stateflow data objects with local scope are defined at the chart level or below.

Description

You must define all local data of a Stateflow block on the chart level or below in the object hierarchy. You cannot define local variables on the machine level; however, parameters and constants are allowed at the machine level.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0125: Scope of internal signals and local auxiliary variables.

Results and Recommended Actions

Condition	Recommended Action
Local data is not defined in the Stateflow hierarchy at the chart level or below.	Define local data at the chart level or below.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check interface signals and parameters

Check whether labeled Stateflow and Simulink input and output signals are strongly typed.

Description

Strong data typing between Stateflow and Simulink input and output signals is required.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0122: Stateflow and Simulink interface signals and parameters.

Results and Recommended Actions

Condition	Recommended Action
A Stateflow chart does not use strong data typing with Simulink.	Select the **Use Strong Data Typing with Simulink I/O** check box for the specified block.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for exclusive states, default states, and substate validity

Check states in state machines.

Description

In state machines:

- There must be at least two exclusive states.
- A state cannot have only one substate.
- The initial state of a hierarchical level with exclusive states is clearly defined by a default transition.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0137: States in state machines.

Prerequisite

A prerequisite MAAB guideline for this check is db_0149: Flowchart patterns for condition actions.

Results and Recommended Actions

Condition	Recommended Action
A system is underspecified.	Validate that the intended design is properly represented in the Stateflow diagram.
Chart has only one exclusive (OR) state.	Make the state a parallel state, or add another exclusive (OR) state.

Condition	Recommended Action
Chart does not have a default state defined.	Define a default state.
Chart has multiple default states defined.	Define only one default state. Make the others nondefault.
State has only one exclusive (OR) substate.	Make the state a parallel state, or add another exclusive (OR) state.
State does not have a default substate defined.	Define a default substate.
State has multiple default substates defined.	Define only one default substate, make the others nondefault.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check optimization parameters for Boolean data types

Check the optimization parameter for Boolean data types.

Description

Optimization for Boolean data types is required

This guideline facilitates

- Workflow
- Code generation

See MAAB guideline jc_0011: Optimization parameters for Boolean data types.

Results and Recommended Actions

Condition	Recommended Action
Configuration setting for **Implement logic signals as boolean data (vs. double)** is incorrect.	Select the **Implement logic signals as boolean data (vs. double)** check box in the Configuration Parameters dialog box **Optimization** pane.

Prerequisite

A prerequisite MAAB guideline for this check is na_0002: Appropriate implementation of fundamental logical and numerical operations.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check model diagnostic settings

Check the model diagnostics configuration parameter settings.

Description

You should enable the following diagnostics:

Algebraic loop
Minimize algebraic loop
Inf or NaN block output
Duplicate data store names
Unconnected block input ports
Unconnected block output ports
Unconnected line
Unspecified bus object at root Outport block
Mux blocks used to create bus signals
Element name mismatch
Invalid function-call connection

This guideline facilitates

- Workflow
- Code generation

Diagnostics not listed in the Results and Recommended Actions section below can be set to any value.

See MAAB guideline jc_0021: Model diagnostic settings.

Results and Recommended Actions

Condition	Recommended Action
Algebraic loop is set to `none`.	Set **Algebraic loop** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box to `error` or `warning`. Otherwise, Simulink might attempt to automatically break the algebraic loops, which can affect execution order of the blocks.
Minimize algebraic loop is set to `none`.	Set **Minimize algebraic loop** on the **Diagnostics > Solver** pane of the Configuration Parameters dialog box to `error` or `warning`. Otherwise, Simulink might attempt to automatically break the algebraic loops for reference models and atomic subsystems, which can affect the execution order for those models or subsystems.
Inf or NaN block output is set to `none`, which can result in numerical exceptions in the generated code.	Set **Inf or NaN block output** on the **Diagnostics > Data Validity > Signals** pane of the Configuration Parameters dialog box to `error` or `warning`.
Duplicate data store names is set to `none`, which can result in nonunique variable naming in the generated code.	Set **Duplicate data store names** on the **Diagnostics > Data Validity > Signals** pane of the Configuration Parameters dialog box to `error` or `warning`.
Unconnected block input ports is set to `none`, which prevents code generation.	Set **Unconnected block input ports** on the **Diagnostics > Data Validity > Signals** pane of the Configuration Parameters dialog box to `error` or `warning`.

Condition	Recommended Action
Unconnected block output ports is set to `none`, which can lead to dead code.	Set **Unconnected block output ports** on the **Diagnostics > Data Validity > Signals** pane of the Configuration Parameters dialog box to `error` or `warning`.
Unconnected line is set to `none`, which prevents code generation.	Set **Unconnected line** on the **Diagnostics > Connectivity > Signals** pane of the Configuration Parameters dialog box to `error` or `warning`.
Unspecified bus object at root Outport block is set to `none`, which can lead to an unspecified interface if the model is referenced from another model.	Set **Unspecified bus object at root Outport block** on the **Diagnostics > Connectivity > Buses** pane of the Configuration Parameters dialog box to `error` or `warning`.
Mux blocks used to create bus signals is set to `none`, which can lead to an unintended bus being created in the model.	Set **Mux blocks used to create bus signals** on the **Diagnostics > Connectivity > Buses** pane of the Configuration Parameters dialog box to `error` or `warning`.
Element name mismatch is set to `none`, which can lead to an incorrect interface in the generated code.	Set **Element name mismatch** on the **Diagnostics > Connectivity > Buses** pane of the Configuration Parameters dialog box to `error` or `warning`.
Invalid function-call connection is set to `none`, which can lead to an error in the operation of the generated code.	Set **Invalid function-call connection** on the **Diagnostics > Connectivity > Function Calls** pane of the Configuration Parameters dialog box to `error` or `warning`, since this condition can lead to an error in the operation of the generated code.

Tip

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check the display attributes of block names

Check the display attributes of block names.

Description

Block names should be displayed when providing descriptive information. Block names should not be displayed if the block function is known from its appearance.

This guideline facilitates

- Readability

See MAAB guideline jc_0061: Display of block names.

Results and Recommended Actions

Condition	Recommended Action
Block name is not descriptive.	These block names should be modified to be more descriptive or not be shown.
Block name is not displayed.	These block names should be shown since they appear to have a descriptive name.
Block name is obvious.	These block names should not be displayed.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check icon display attributes for port blocks

Check the **Icon display** setting for Inport and Outport blocks.

Description

The **Icon display** setting is required.

This guideline facilitates

- Readability

See MAAB guideline jc_0081: Icon display for Port block.

Results and Recommended Actions

Condition	Recommended Action
The **Icon display** setting is incorrect.	Set the **Icon display** to `Port number` for the specified Inport and Outport blocks.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether subsystem block names include invalid characters

Check whether subsystem block names include invalid characters.

Description

The names of all subsystem blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0201: Usable characters for Subsystem names.

Results and Recommended Actions

Condition	Recommended Action
The subsystem name contains illegal characters.	Rename the subsystem. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The subsystem name starts with a number.	Rename the subsystem.
The subsystem name starts with an underscore ("_").	Rename the subsystem.
The subsystem name ends with an underscore ("_").	Rename the subsystem.
The subsystem name contains consecutive underscores.	Rename the subsystem.
The subsystem name has consecutive underscores.	Rename the subsystem.
The subsystem name has blank spaces.	Rename the subsystem.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Tip

Use underscores to separate parts of a subsystem name instead of spaces.

Check whether Inport and Outport block names include invalid characters

Check whether Inport and Outport block names include invalid characters.

Description

The names of all Inport and Outport blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0211: Usable characters for Inport blocks and Outport blocks.

Results and Recommended Actions

Condition	Recommended Action
The block name contains illegal characters.	Rename the block. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The block name starts with a number.	Rename the block.
The block name starts with an underscore ("_").	Rename the block.
The block name ends with an underscore ("_").	Rename the block.
The block name contains consecutive underscores.	Rename the block.
The block name has consecutive underscores.	Rename the block.
The block name has blank spaces.	Rename the block.

Tips

Use underscores to separate parts of a block name instead of spaces.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether signal line names include invalid characters

Check whether signal line names include invalid characters.

Description

The names of all signal lines are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0221: Usable characters for signal line names.

Results and Recommended Actions

Condition	Recommended Action
The signal line name contains illegal characters.	Rename the signal line. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The signal line name starts with a number.	Rename the signal line.
The signal line name starts with an underscore ("_").	Rename the signal line.
The signal line name ends with an underscore ("_").	Rename the signal line.
The signal line name contains consecutive underscores.	Rename the signal line.
The signal line name has consecutive underscores.	Rename the signal line.

Condition	Recommended Action
The signal line name has blank spaces.	Rename the signal line.
The signal line name has control characters.	Rename the signal line.

Tip

Use underscores to separate parts of a signal line name instead of spaces.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check whether block names include invalid characters

Check whether block names include invalid characters.

Description

The names of all blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

This guideline does not apply to subsystem blocks.

See MAAB guideline jc_0231: Usable characters for block names.

Results and Recommended Actions

Condition	Recommended Action
The block name contains illegal characters.	Rename the block. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The block name starts with a number.	Rename the block.
The block name has blank spaces.	Rename the block.
The block name has double byte characters.	Rename the block.

Prerequisite

A prerequisite MAAB guideline for this check is jc_0201: Usable characters for Subsystem names.

Tip

Carriage returns are allowed in block names.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check Trigger and Enable block port names

Check Trigger and Enable block port names.

Description

Block port names should match the name of the signal triggering the subsystem.

This guideline facilitates

- Readability

See MAAB guideline jc_0281: Naming of Trigger Port block and Enable Port block.

Results and Recommended Actions

Condition	Recommended Action
Trigger block does not match the name of the signal to which it is connected.	Match Trigger block names to the connecting signal.
Enable block does not match the name of the signal to which it is connected.	Match Enable block names to the connecting signal.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for Simulink diagrams that have nonstandard appearance attributes

Check model appearance setting attributes.

Description

Model appearance settings are required to conform to the guidelines when the model is released.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline na_0004: Simulink model appearance.

Results and Recommended Actions

Condition	Recommended Action
Diagrams do not have white backgrounds.	Select **Format > Screen Color > Automatic**.
Diagrams do not have zoom factor set to 100%.	Select **View > Normal (100%)**.
The toolbar is not visible.	Select **View > Toolbar**.
The status bar is not visible.	Select **View > Status Bar**.
Block backgrounds are not white.	Blocks should have black foregrounds with white backgrounds. Click the specified block and select **Format > Foreground Color > Black** and **Format > Background Color > White**.
Wide Nonscalar Lines is cleared.	Select **Format > Port/Signal Displays > Wide Nonscalar Lines**.

Condition	Recommended Action
Viewer Indicators is cleared.	Select **Format > Port/Signal Displays > Viewer Indicators**.
Testpoint Indicators is cleared.	Select **Format > Port/Signal Displays > Testpoint Indicators**.
Port Data Types is selected.	Clear **Format > Port/Signal Displays > Port Data Types**.
Storage Class is selected.	Clear **Format > Port/Signal Displays > Storage Class**.
Signal Dimensions is selected.	Clear **Format > Port/Signal Displays > Signal Dimensions**.
Model Browser is selected.	Clear **View > Model Browser Options > Model Browser**.
Sorted Order is selected.	Clear **Format > Block Displays > Sorted Order**.
Model Block Version is selected.	Clear **Format > Block Displays > Model Block Version**.
Model Block I/O Mismatch is selected.	Clear **Format > Block Displays > Model Block I/O Mismatch**.
Execution Context Indicator is selected.	Clear **Format > Block Displays > Execution Context Indicator**.
Sample Time Colors is selected.	Clear **Format > Port/Signal Displays > Sample Time Colors**.
Library Link Display is set to **User** or **All**.	Select **Format > Library Link Display > None**.
Linearization Indicators is cleared.	Select **Format > Port/Signal Displays > Linearization Indicators**.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check visibility of port block names

Check the visibility of port block names.

Description

An organization applying the MAAB guidelines must select one of the following alternatives to enforce:

- The name of port blocks are not hidden.
- The name of port blocks must be hidden.

This guideline facilitates

- Readability

Note This check does not look in masked subsystems.

See MAAB guideline na_0005: Port block name visibility in Simulink models.

Input Parameters

All Port names should be shown (Format/Show Name)
Select this check box if all ports should show the name, including subsystems.

Results and Recommended Actions

Condition	Recommended Action
Blocks do not show their name and the **All Port names should be shown (Format/Show Name)** check box is selected.	Change the format of the specified blocks to show names according to the input requirement.

Condition	Recommended Action
Blocks show their name and the **All Port names should be shown (Format/Show Name)** check box is cleared.	Change the format of the specified blocks to hide names according to the input requirement.
Subsystem blocks do not show their port names.	Set the subsystem parameter **Show port labels** to a value other than `none`.
Subsystem blocks show their port names.	Set the subsystem parameter **Show port labels** to `none`.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for direction of subsystem blocks

Check the orientation of subsystem blocks.

Description

Subsystem inputs must be located on the left side of the block, and outputs must be located on the right side of the block.

This guideline facilitates

- Readability

See MAAB guideline jc_0111: Direction of Subsystem.

Results and Recommended Actions

Condition	Recommended Action
Subsystem blocks are not in the correct orientation.	Change the subsystem blocks to have the correct orientation, with inports on the left and outports on the right.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for proper position of constants used in Relational Operator blocks

Check the position of Constant blocks used in Relational Operator blocks.

Description

When the relational operator is used to compare a signal to a constant value, the constant input should be the second, lower input.

This guideline facilitates

- Readability
- Code generation

See MAAB guideline jc_0131: Use of Relational Operator block.

Results and Recommended Actions

Condition	Recommended Action
Relational Operator blocks have a Constant block on the first, upper input.	Move the Constant block to the second, lower input.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for entry format in state blocks

Check the format of entries in state blocks.

Description

A new line should be started after the `entry`, `during`, and `exit` action statements and after the completion of an assignment statement ";".

This guideline facilitates

- Readability

See MAAB guideline jc_0501: Format of entries in a State block.

Results and Recommended Actions

Condition	Recommended Action
An entry in a state block is not formatted correctly.	Validate that the intended design is properly represented in the Stateflow diagram.
An `entry` action statement is not by itself.	Add a new line.
Multiple `entry` action statements found on one line.	Add a new line between `entry` action statements.
An `during` action statement is not by itself.	Add a new line.
Multiple `during` action statements found on one line.	Add a new line between `during` action statements.
An `exit` action statement is not by itself.	Add a new line.
Multiple `exit` action statements found on one line.	Add a new line between `exit` action statements.

Condition	Recommended Action
Multiple action statements found on one line.	Add a new line between action statements.
Potential misuse of semicolon (;) on a line.	Correct the use of the semicolon where specified.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for use of tunable parameters in Stateflow

Check for use of tunable parameters in Stateflow charts.

Description

Include tunable parameters in a Stateflow chart as inputs from the Simulink model.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0541: Use of tunable parameters in Stateflow.

Results and Recommended Actions

Condition	Recommended Action
Stateflow charts reference Simulink data objects, which should be used as inputs from the Simulink model.	Make the Simulink data objects inputs from the Simulink model to the specified Stateflow chart.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for proper use of Switch blocks

Check for proper use of Switch blocks.

Description

This check verifies that the Switch block's control input (the second input) is a Boolean value and that the block is configured to pass the first input when the control input is nonzero.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline jc_0141: Use of the Switch block.

Results and Recommended Actions

Condition	Recommended Action
The Switch block's control input (second input) is not a Boolean value.	Change the data type of the control input to Boolean.
The Switch block is not configured to pass the first input when the control input is nonzero.	Set the block parameter **Criteria for passing first input** to u2 ~=0.

See Also

- See the description of the Switch block in the Simulink documentation.
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for proper use of signal buses and Mux block usage

Check for proper use of signal busses and Mux block usage.

Description

This check verifies whether a model is using signal buses and Mux blocks properly.

This guideline facilitates

- Readability
- Workflow

See MAAB guidelinena_0010: Grouping data flows into signals.

Results and Recommended Actions

Condition	Recommended Action
The individual scalar input signals for a Mux block do not have common functionality, data types, dimensions, and units.	Modify the scalar input signals such that the specifications match.
The output of a Mux block is not a vector.	Change the output of the Mux block to a vector.
All inputs to a Mux block are not scalars.	Make sure that all input signals to Mux blocks are scalars.
The input for a Bus Selector block is not a bus signal.	Make sure that the input for all Bus Selector blocks is a bus signal.

See Also

- Using Composite Signals in the Simulink documentation.

- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for mismatches between Stateflow ports and associated signal names

Check for mismatches between Stateflow ports and associated signal names.

Description

The name of Stateflow input and output should be the same as the corresponding signal. This guideline is required for:

- Readability
- Workflow

See MAAB guideline db_0123: Stateflow port names.

Results and Recommended Actions

Condition	Recommended Action
Signals have names that differ from those of their corresponding Stateflow ports.	Change the names of either the signals or the Stateflow ports.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for proper scope of From and Goto blocks

Check the scope of From and Goto blocks.

Description

You can use global scope for controlling flow. However, From and Goto blocks must use local scope for signal flows.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline na_0011: Scope of Goto and From blocks.

Results and Recommended Actions

Condition	Recommended Action
From and Goto blocks are not configured with local scope.	• Make sure the ports are connected correctly. • Change the scope of the specified blocks to local.

See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Requirements Consistency Checks

In this section...

Identify requirement links with missing documents

Ensure that requirements link to existing documents.

Description

You used the Requirements Management Interface (RMI) to associate a design requirements document with a part of your model design and the interface cannot find the specified document.

Results and Recommended Actions

Condition	Recommended Action
The requirements document associated with a part of your model design is not accessible at the specified location.	Open the Requirements dialog box and correct the path name of the requirements document or move the document to the specified location.

See Also

"Adding and Viewing Requirement Links" on page 2-4

Identify requirement links that specify invalid locations within documents

Ensure that requirements link to valid locations (e.g., bookmarks, line numbers, anchors) within documents.

Description

You used the Requirements Management Interface (RMI) to associate a location in a design requirements document (a bookmark, line number, or anchor) with a part of your model design and the interface cannot find the specified location in the specified document.

Results and Recommended Actions

Condition	Recommended Action
The location in the requirements document associated with a part of your model design is not accessible.	Open the Requirements dialog box and correct the location reference within the requirements document.

See Also

"Adding and Viewing Requirement Links" on page 2-4

Identify selection-based links having descriptions that do not match their requirements document text

Ensure that descriptions of selection-based links use the same text found in their requirements documents.

Description

You used selection-based linking of the Requirements Management Interface (RMI) to label requirements in the model's **Requirements** menu with text that appears in the corresponding requirements document. This check helps you manage traceability by identifying requirement descriptions in the menu that are not synchronized with text in the documents.

Results and Recommended Actions

Condition	Recommended Action
Selection-based links have descriptions that differ from their corresponding selections in the requirements documents.	If the difference reflects a change in the requirements document, click the link in the Model Advisor results to replace the current description in the selection-based link with the text from the requirements document (the external description). Alternatively, you can right click the object in the model window, select **Edit/Add Links** from the **Requirements** menu, and use the Requirements dialog box that appears to synchronize the text.

See Also

"Selection-Based Linking" on page 2-20

Identify requirement links with inconsistent path types and preferences

Check that requirement paths are of the type selected in the preferences.

Description

You are using the Requirements Management Interface (RMI) and the paths specifying the location of your requirements documents differ from the file reference type set as your preference.

Results and Recommended Actions

Condition	Recommended Action
The paths indicating the location of requirements documents use a file reference type that differs from the preferences specified in the **Selection-based linking** dialog box.	Change the preferred document file reference type or the specified paths by doing one of the following: • Click **Fix** to change the current path to the valid path. • Update the preference in the **Selection-based linking** dialog box. In the model window, select **Tools > Requirements > Link settings** and change the value for the **Document file reference** option.

See Also

"Selection-Based Linking" on page 2-20

A

Examples

Use this list to find examples in the documentation.

Requirements Management Interface

Requirements Management Interface (DOORS Version)

Verification Manager

Model Coverage

Index

A

C

D

E

F

I

L

M

O

P

R

S

T

V