Simulink® Verification and Validation™

User's Guide

R2012a

MATLAB®
&SIMULINK®

MathWorks®

How to Contact MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Simulink® Verification and Validation™ User's Guide

© COPYRIGHT 2004–2012 by The MathWorks, Inc.

Trademarks

Patents

Revision History

Contents

How Is Requirements Link Information Stored?

3

Reviewing Requirements

4

Requirements Links Maintenance

5

IBM Rational DOORS Surrogate Module Synchronization

6

Navigation from Requirements Documents

7

Custom Types of Requirements Documents

8

Requirements Information in Generated Code

9

Model Component Testing

Signal Monitoring with Model Verification Blocks

Using Model Verification Blocks

12

Constructing Simulation Tests Using the Verification Manager

13

Linking Test Cases to Requirements Documents Using the Verification Manager

14

Model Coverage Analysis

Model Coverage Definition

15

Model Objects That Receive Model Coverage

16

Setting Model Coverage Options

17

19

20

Excluding Model Objects From Coverage

Automating Model Coverage Tasks

21

Checking Systems with the Model Advisor

Checking Systems Interactively

22

23

Customizing the Model Advisor

24

<div style="text-align: right">

Authoring Custom Checks

</div>

25

Creating Custom Configurations by Organizing Checks and Folders

26

Creating Procedural-Based Model Advisor Configurations

27

Deploying Custom Configurations

28

Examples

A

Index

Getting Started

- "Product Description" on page 1-2
- "System Requirements" on page 1-3

Product Description

Verify models and generated code

Simulink® Verification and Validation™ automates requirements tracing, modeling standards compliance checking, and test-harness generation.

You can create detailed requirements traceability reports, author your own modeling style checks, and develop check configurations to share with engineering teams. Requirements documentation can be linked to models, test cases, and generated code. You can generate harness models for testing model components and code before the complete system becomes available.

Simulink Verification and Validation provides modeling standards checks for the DO-178B and IEC 61508 industry standards. Additional support is available through DO Qualification Kit and IEC Certification Kit.

Key Features

- Compliance checking for MAAB style guidelines and high-integrity system design guidelines (DO-178B and IEC-61508)

- Model Advisor Configuration Editor, including custom check authoring

- Requirements Management Interface for traceability of model objects, code, and tests to requirements documents

- Automatic test-harness generation for subsystems

- Component testing via simulation, software-in-the-loop (SIL), and processor-in-the-loop (PIL)

- Programmable scripting interface for automating compliance checking, requirements traceability analysis, and component testing

System Requirements

Operating System Requirements

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

- Microsoft® Windows® XP, Windows Vista™, and Windows 7

- UNIX® systems (Requirements linking to HTML and TXT 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 2003 or later

- Microsoft Excel® 2003 or later

- IBM® Rational® DOORS® 6.0 or later

- Adobe® PDF

Requirements Traceability

- Chapter 2, "Links Between Models and Requirements"
- Chapter 3, "How Is Requirements Link Information Stored?"
- Chapter 4, "Reviewing Requirements"
- Chapter 5, "Requirements Links Maintenance"
- Chapter 6, "IBM Rational DOORS Surrogate Module Synchronization"
- Chapter 7, "Navigation from Requirements Documents"
- Chapter 8, "Custom Types of Requirements Documents"
- Chapter 9, "Requirements Information in Generated Code"

2

Links Between Models and Requirements

- "Linking to Requirements Modeled in Simulink" on page 2-40
- "Linking Signal Builder Blocks to Requirements Documents" on page 2-47
- "Linking Signal Builder Blocks to Model Objects" on page 2-49

Overview of the Requirements Management Interface (RMI)

If you want to link Simulink and Stateflow objects to requirements, including external documents, verification objects, or tests, use the RMI to:

- Associate Simulink and Stateflow objects with requirements.
- Create links between Simulink and Stateflow model objects.
- Navigate from a Simulink or Stateflow object to requirements.
- Navigate from an embedded link in a requirements document to the corresponding Simulink or Stateflow object.
- Review requirements links in your model using highlighting and tags that you define.
- Create reports for your Simulink model that show which objects link to which requirements.

What Is a Requirements Link?

When you want to navigate from a Simulink model to a location inside a requirements document, requirements links are inserted into the model.

Requirements links have the following attributes:

- A description of up to 255 characters.

- A requirements document path name, such as a Microsoft Word file or a module in an IBM Rational DOORS database. (The RMI supports several built-in document formats. You can also register custom types of requirements documents. See "Supported Requirements Document Types" on page 2-6.)

- A designated location inside the requirements document, such as:

  - Bookmark
  - Anchor
  - ID
  - Page number
  - Line number
  - Cell range
  - Link target
  - Tags that you define

Requirements Link Storage

When you create links from a model to external requirements, by default, the Requirements Management Interface (RMI) stores the information about the links internally in the model (.mdl) file. When links are stored with the model, each time you change requirements links, the time stamp and version number of the model changes.

If you do not want to modify your model when creating or modifying requirements links, use *external storage* for requirements links. External storage is a mechanism for saving information about linked requirements in a file that is separate from the model .mdl file. The RMI saves information about the requirements links in a file that, by default, is named *model_name*.req and is saved in the same folder as the model.

When you use external storage for requirements links, no changes are made to the .mdl file when you modify the requirements links.

Supported Requirements Document Types

The following table lists the supported requirements document types. For each document type, it lists the options for requirements locations within the document.

Requirements Document Type	Location Options
Microsoft Word document	• `Named item` — A bookmark name. The RMI links to the location of that bookmark in the document. The most reliable location identifier because the link is maintained when the target content is modified or moved. • `Search text` — A search string. The RMI links to the first occurrence of that string in the document. This search is not case sensitive. • `Page/item number` — A page number. The RMI links to the top of the specified page.

Microsoft Excel workbook	• `Named item` — A named range of cells. The RMI links to that named item in the workbook. The most reliable location identifier because the link is maintained when the target content is modified or moved.
	
	• `Search text` — A search string. The RMI links to the first occurrence of that string in the workbook. This search is not case sensitive.
	
	• `Sheet range` — A cell location in a workbook:
	  ▪ Cell number (`A1`, `C13`)
	  ▪ Range of cells (`C5:D7`)
	  ▪ Range of cells on another worksheet (`Sheet1!A1:B4`)
	The RMI links to that cell or cells.

IBM Rational DOORS database	`Page/item number` — The unique numeric ID of the target DOORS object. The RMI links to that object.

Requirements Document Type	Location Options
MuPAD® notebook	Named item — The name of a link target in a MuPAD notebook.
Simulink DocBlock block (RTF format only)	Create links to the RTF file associated with the DocBlock block to a Microsoft Word file: • Search text — A search string. The RMI links to the first occurrence of that string in the document. This search is not case sensitive. • Named item — A bookmark name. The RMI links to the location of that bookmark in the document. • Page/item number — A page number. The RMI links to the top of that page.
Text document	• Search text — A search string. The RMI links to the first occurrence of that string within the document. This search is not case sensitive. • Line number — A line number. The RMI links to the beginning of that line.
HTML file	You can link only to a named anchor. For example, in your HTML requirements document, if you define the anchor `<A NAME=valve_timing> ...contents... </A>` in the **Location** field, enter valve_timing or, from the document index, choose the anchor name. Select the **Document Index** tab in the **Requirements > Edit/Add Links** dialog box to see available anchors in an HTML file.

Requirements Document Type	Location Options
PDF file	• `Named item` — A bookmark name. The RMI links to the location of that bookmark in the document. Not fully supported, depends on the platform and bookmark type. • `Page/item number` — A page number. The RMI links to the top of that page. **Note** The RMI cannot create a document index of bookmarks in PDF files.
Web browser URL	The RMI can link to a URL location. In the **Document** field, type the URL string. When you click the link, the document opens in a Web browser: • `Named item` — An anchor name. The RMI links to that location on the Web page at that URL.

If you register custom types of requirements documents, the RMI supports those types of documents. For more information, see Chapter 8, "Custom Types of Requirements Documents".

Supported Model Objects for Requirements Linking

You can associate requirements links between the following types of Simulink model objects:

- Simulink block diagrams

- Simulink blocks, including library-linked blocks and subsystems

- Signal Builder signal groups

- Stateflow charts, subcharts, states, transitions, and boxes

- Stateflow functions

What Is Selection-Based Linking?

You can use *selection-based linking* to create links from a model object to another model object or to an object in a requirements document. Selection-based linking is the easiest way to create requirements links from a model to an external document.

Workflows for Creating Links Using Selection-Based Linking

To create a link from the model to the document, using any model and a requirements document:

1 In a requirements document, select text or objects to link to.

2 Right-click the model object. Select **Requirements** and then the option that corresponds to one of the types for which selection-based linking is available:

- **Add link to Word selection**
- **Add link to active Excel cell**
- **Add link to current DOORS object**
- **Select for linking with Simulink**

Linking Model Objects

To create a requirement link from one model object to another model object:

1 Right-click the link destination model object and select **Requirements > Select for linking with Simulink**.

2 Right-click the link source model object and select **Requirements > Add link to selected object**.

3 Right-click the link source model object again and select **Requirements**. The new link appears at the top of the **Requirements** submenu.

Linking Multiple Model Objects to a Requirements Document

You can use the same technique to link multiple Simulink and Stateflow objects to requirements in any type of requirements document that the RMI supports. The workflow for linking to multiple model objects is:

1 In the requirements document, select the requirement.

2 In the model window, select all the objects to link to that requirement.

3 Right-click one of the selected objects and select the selection-based linking option that corresponds to your requirements document.

Configuring the Requirements Management Interface (RMI)

To use the features of the Requirements Management Interface (RMI), you must communicate with external software products such as Microsoft Office and IBM Rational DOORS.

Initial configuration steps are required to setup the RMI if you need to:

- Use the RMI with DOORS applications (PC only).
- Use ActiveX® controls for navigation from Microsoft Office documents to Simulink models. You might need to register ActiveX controls when you work with existing requirements documents (PC only).

You can setup the initial configuration for both cases by running this command:

```
rmi setup
```

If you do not have DOORS installed on your system, the `rmi setup` command does not install the DOORS API.

If the `rmi setup` command fails to detect a DOORS installation on your system, and you know that the DOORS software is installed, enter the following command:

```
rmi setup doors
```

This command prompts you to enter the path to your DOORS installation, and then installs the required files.

The Requirements Dialog Box

In this section...
"Creating Requirements Using the Requirements Dialog Box" on page 2-14
"Requirements Tab" on page 2-15
"Document Index Tab" on page 2-16

Creating Requirements Using the Requirements Dialog Box

You can manage the requirements links associated with a given model object in the centralized location of the Requirements dialog box. In this dialog box, you can:

- Create requirements links from one or more model objects.

- Customize information about requirements links, including specifying user tags to filter requirements highlighting and reporting.

- Delete existing requirements links.

- Modify the stored order of requirements to control the order of labels in context menus for linked objects.

To open the Requirements dialog box, right-click a block in a Simulink model and select **Requirements > Edit/Add Links**.
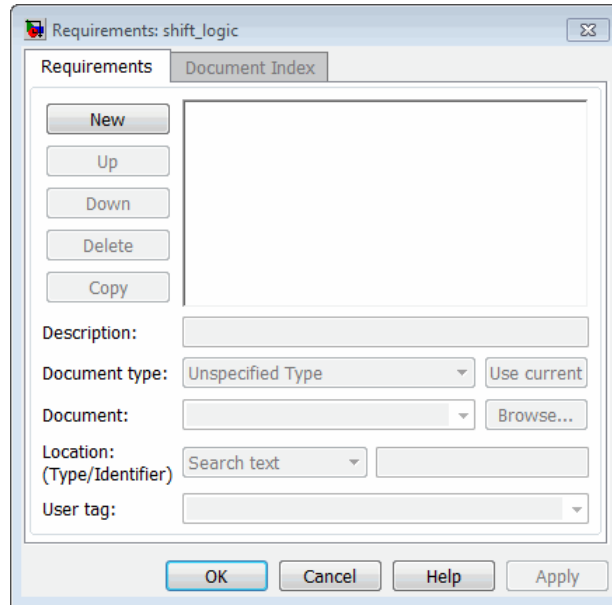
Requirements Tab

On the **Requirements** tab, you specify detailed information about the link, including:

- Description of the requirement (up to 255 words). If you create a link using the document index, *unless* a description already exists, the name of the index location becomes the description for the link .

- Path name to the requirements document.

- Document type (Microsoft Word, Microsoft Excel, IBM Rational DOORS, MuPAD, HTML, text file, etc.).

- Location of the requirement (search text, named location, or page or item number).

- User-specified tag or keyword.

Document Index Tab

The **Document Index** tab is available only if you have specified a file in the **Document** field on the **Requirements** tab that supports indexing. On the **Document Index** tab, the RMI generates a list of locations in the specified requirements document for the following types of requirements documents:

- Microsoft Word
- IBM Rational DOORS
- HTML files
- MuPAD

Note The RMI cannot create document indexes for PDF files.

From the document index, select the desired requirement from the document index and click **OK**. *Unless* a description already exists, the name of the index location becomes the description for the link.

If you make any changes to your requirements document, to load any newly created locations into the document index, you must click **Refresh**. During any MATLAB session, the RMI does not reload the document index unless you click the **Refresh** button.

The Requirement Settings Dialog Box

You can manage your RMI preferences in the Requirements Settings dialog box. The settings are not associated with the model. To open the Requirements Settings dialog box, from any model window, select **Tools > Requirements > Settings**. In this dialog box, you can select the:

- **Report** tab to customize the requirements report without using the Report Generator. For setting information, see "Customizing a Requirements Report Using the RMI Settings" on page 4-17.

- **Selection Linking** tab to set the options for linking to the active selection in a supported document. For setting information, see "Selection Linking Tab" on page 2-17.

- **Filter** tab to set the options for filtering requirements in a model. For filtering information, see "Configuring Requirements Filtering" on page 4-30.

- **Storage** tab to set the default way in which the RMI stores requirements links in the model. For storage information, see "Specifying Requirements Links Storage" on page 3-4.

Selection Linking Tab

In the Requirements Settings dialog box, on the **Selection Linking** tab, are the following options for linking to the active selection in a supported document. To open the Requirements Settings dialog box, select **Tools > Requirements > Settings**.

Options	Description
For linking to the active selection within an external document:	
Enabled applications	Enable selection-based linking shortcuts to Microsoft Word, Microsoft Excel, or DOORS applications.
Document file reference	Select type of file reference. For information on what settings to use, see "Resolving the Document Path" on page 5-15.

Options	Description
Apply this user tag to new links	Enter text to attach to the links you create. For more information about user tags, see "Filtering Requirements with User Tags" on page 4-25
When creating selection-based links:	
Modify destination for bi-directional linking	Creates links both to and from selected link destination.
Store absolute path to model file	Select type of file reference. For information, see "Resolving the Document Path" on page 5-15.
Use custom bitmap for navigation controls in documents	Select and browse for your bitmap. You can use your own bitmat file to control the appearance of navigation links in your document.
Use ActiveX buttons in Word and Excel (backward compatibility)	Select to use legacy ActiveX controls to create links in Microsoft Word and Microsoft Excel applications. By default, if not selected, you create URL-based links.

Linking Model Objects

You can create a requirements link from one model object to another model object:

1 Right-click the link destination model object and select **Requirements > Select for linking with Simulink**.

2 Right-click the link source model object and select **Requirements > Add link to selected object**.

3 Right-click the link source model object again and select **Requirements**. The new link appears at the top of the **Requirements** submenu.

You can create links between objects in related models. This example shows how to link model objects in slvnvdemo_powerwindow_controller and slvnvdemo_powerwindow.

1 Open the slvnvdemo_powerwindow_controller and slvnvdemo_powerwindow models.

2 In the slvnvdemo_powerwindow model window, double-click the power_window_control_system subsystem. The power_window_control_system subsystem opens.

3 In the slvnvdemo_powerwindow/power_window_control_system subsystem window, right-click the control subsystem. Select **Requirements > Select for linking with Simulink**.

4 In the slvnvdemo_powerwindow_controller model window, right-click the control subsystem. Select **Requirements > Add link to selected object**.

5 Right-click the slvnvdemo_powerwindow_controller/control subsystem and select **Requirements**. The new RMI link appears at the top of the **Requirements** submenu.

6 To verify that the links were created, in the slvnvdemo_powerwindow_controller model window, select **Tools > Requirements > Highlight model**.

The blocks with requirements links are highlighted.

7 Close the `slvnvdemo_powerwindow_controller` and `slvnvdemo_powerwindow` models.

Linking Multiple Model Objects to a Requirements Document

You can link multiple Simulink and Stateflow objects to a requirement. The workflow is:

1 In the requirements document, select the requirement.

2 In the model window, select the objects to link to that requirement.

3 Right-click one of the selected objects and select one of the selection-based linking options:

- **Add link to Word selection**
- **Add link to active Excel cell**
- **Add link to current DOORS object**

Linking Multiple Model Objects to a Requirement Document Using a Simulink DocBlock

You can minimize the number of links to the external requirements document by using a DocBlock in the model. You can insert aDocBlock at the top level of the model and link your external requirements document to it. Then you can link the DocBlock to all objects in the model that are relevant to the requirement in the external document.

This example shows how to link multiple model objects to a requirement document using a DocBlock.

1 Open the example model:

```
slvnvdemo_fuelsys_officereq
```

2 Open a requirements document associated with that model:

```
rmi('view','slvnvdemo_fuelsys_officereq',1);
```

3 Insert a Simulink DocBlock into the model.

4 Right-click DocBlock and select **Mask Parameters**. The Block Parameters: DocBlock dialog box opens.

5 In the Block Parameters: DocBlock dialog box **Document type** drop-down list, select RTF.

6 Click **Apply** or **OK** to create the link.

7 In slvnvdemo_FuelSys_DesignDescription.docx, find and select the section titled **2.2 Determination of pumping efficiency**.

8 In the slvnvdemo_fuelsys_officereq model window, right-click the DocBlock and select **Requirements > Add link to Word selection**.

The RMI inserts a bookmark at that location in the requirements document.

9 In the slvnvdemo_fuelsys_officereq model window, right-click the DocBlock and select **Requirements > Select for linking with Simulink**.

10 In the slvnvdemo_fuelsys_officereq/fuel rate controller subsystem window, select the control logic chart and the Airflow calculation subsystems. Use the shift key to select both.

11 Right-click the Airflow calculation subsystem and select **Requirements > Add link to selected object**.

12 In the slvnvdemo_fuelsys_officereq model window, select DocBlock, right-click, and select **Requirements**.

The link to the new requirements are on the top menu option.

13 To verify that the links were created, select **Tools > Requirements > Highlight model**.

The DocBlock, control logic chart, and Airflow calculation subsystem are highlighted.

14 Close the slvnvdemo_fuelsys_officereq model and the slvnvdemo_FuelSys_DesignDescription.docx requirements document.

Linking to Requirements in Microsoft Word Documents

Creating Bookmarks in a Microsoft Word Requirements Document

You can create bookmarks in your Microsoft Word requirements documents to identify the requirements that you want to link to. When you create the links, you specify that the RMI must link to an existing bookmark, rather than create a new bookmark.

This approach offers several advantages:

- You can give the bookmarks meaningful names that represent the content of the requirement.

- When the RMI creates the links, it does not modify your requirements document.

If you have a requirements document that contains bookmarks for requirements, to link from a Simulink model to bookmarks that represent requirements, follow these steps:

1 Open your model.

2 Open your Microsoft Word requirements document that contains bookmarks that identify requirements.

3 Right-click a block in the model that you want to link to a requirement and select **Requirements > Edit/Add Links**

The Requirements dialog box opens.

4 Click **New**.

5 Click **Browse** and navigate to the Microsoft Word requirements document that has bookmarks.

6 Open the document. The RMI populates the **Document** and **Document type** fields.

7 Click the **Document Index** tab of the Requirements dialog box.

The **Document Index** tab lists all bookmarks in the requirements document, as well as all section headings (text that you have styled as **Heading 1**, **Heading 2**, and so on).

The following graphic is an example of a document index that lists the bookmarks in a requirements document. The document index lists the bookmarks in alphabetical order, not in order of location within the document.
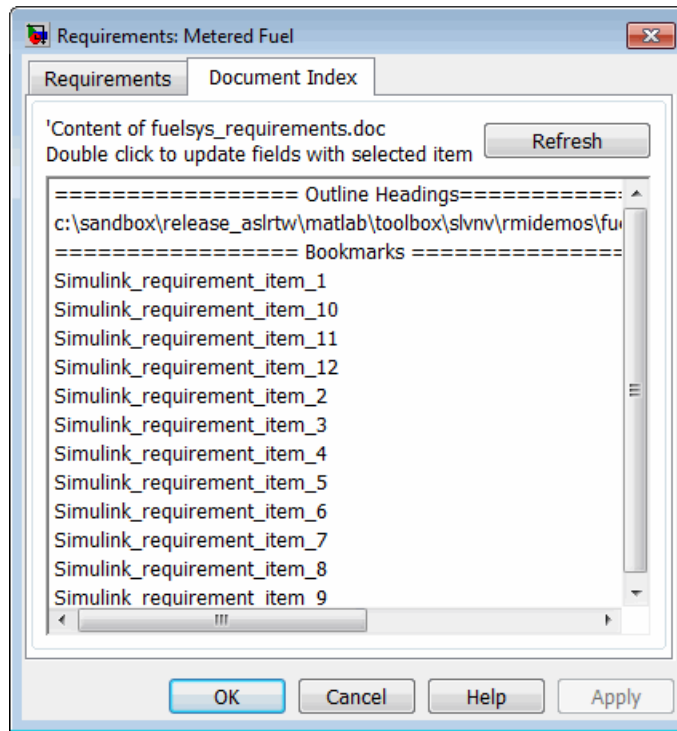
8 Select the bookmark that you want to link the block to and click **OK**.

The RMI creates a link from the block to the location of the bookmark in the requirements document without modifying the document itself.

Opening the Demo Model and Associated Requirements Document

This example describes how to create links from objects in a Simulink model to selected requirements text in a Microsoft Word document.

Navigate from the model to the requirements document:

1 Open the demo model:

```
slvnvdemo_fuelsys_officereq
```

2 Open a requirements document associated with that model:

```
rmi('view','slvnvdemo_fuelsys_officereq',1);
```

Keep the demo model and the requirements document open.

Creating a Link from a Model Object to a Microsoft Word Requirements Document

Create a link from the Airflow calculation subsystem in the slvnvdemo_fuelsys_officereq model to selected text in the requirements document:

1 In slvnvdemo_FuelSys_DesignDescription.docx, find the section titled **2.2 Determination of pumping efficiency**.

2 Select the header text.

3 Open the demo model:

```
slvnvdemo_fuelsys_officereq
```

4 Select **Tools > Requirements > Settings**. The Requirements Settings dialog box opens.

5 On the **Selection Linking** tab of the Requirements Settings dialog box:

- Set the **Document file reference** option to path relative to model folder.

- Enable **Modify destination for bi-directional linking**.

  When you select this option, every time you create a selection-based link from a model object to a requirement, the RMI inserts navigation objects at the designated location.

  For more information about the settings, see "The Requirement Settings Dialog Box" on page 2-17.

6 Double-click the fuel rate controller subsystem to open it.

7 Open the Airflow calculation subsystem.

8 Right-click the Pumping Constant block and select **Requirements > Add link to Word selection**.

The RMI inserts a bookmark at that location in the requirements document and assigns it a generic name, in this case, `Simulink_requirement_item_7`.

9 To verify that the link was created, select **Tools > Requirements > Highlight model**.

The Pumping Constant block, and other blocks with requirements links, are highlighted.

10 To navigate to the link, right-click the Pumping Constant block and select **Requirements > 1. "Determination of pumping efficiency"**.

The section **2.2 Determination of pumping efficiency** is displayed, selected in the requirements document.

Keep the demo model and the requirements document open.

Viewing Link Details

To view the details of the link that you just created, right-click the Pumping Constant block and select **Requirements > Edit/Add Links**.

The Requirements dialog box opens.
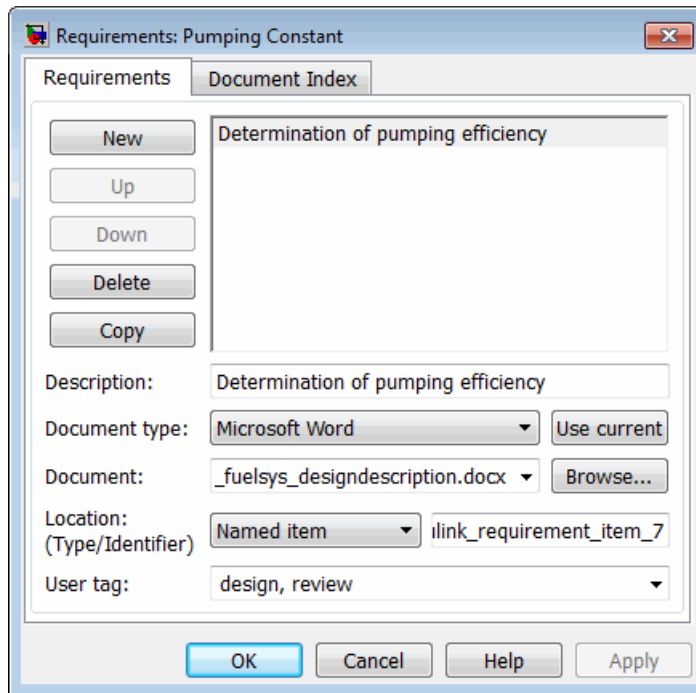
This dialog box contains the following information for the link you just created:

- Description of the link, which for selection-based links, matches the text of the selected requirements document, in this case `Determination of pumping efficiency`.

- Name of the requirements document, in this case `slvnvdemo_FuelSys_DesignDescription.docx`.

- Document type, in this case, `Microsoft Word`.

- The type and identifier of the location in the requirements document. With selection-based linking for Microsoft Word requirements documents, the RMI creates a bookmark in the requirements document. For this link, the RMI created a bookmark named `Simulink_requirement_item_7`.

  If you do not want the RMI to modify the Microsoft Word requirements document when it creates links, create bookmarks in your Microsoft

Word file, as described in "Creating Bookmarks in a Microsoft Word Requirements Document" on page 2-23.

- User tag, a user-defined keyword. This link does not have a user tag.

Note For more information about user tags, see "Filtering Requirements with User Tags" on page 4-25

Linking to Requirements in IBM Rational DOORS Databases

This example shows how to create links from objects in a Simulink model to requirements in an IBM Rational DOORS database.

1 Open a DOORS formal module.

2 Click to select one of the requirement objects.

3 Open the demo model:

   sldemo_fuelsys

4 Open the fuel_rate_control subsystem.

5 Right-click the airflow_calc subsystem and select **Requirements > Add link to current DOORS object**.

6 To confirm the requirement link, right-click the airflow_calc subsystem and select **Requirements**. In the submenu, the top item is the heading text for the DOORS requirement object. If you navigate to a DOORS requirement, the DOORS module opens in read only mode. If you want to modify the DOORS module, open the module using DOORS software.
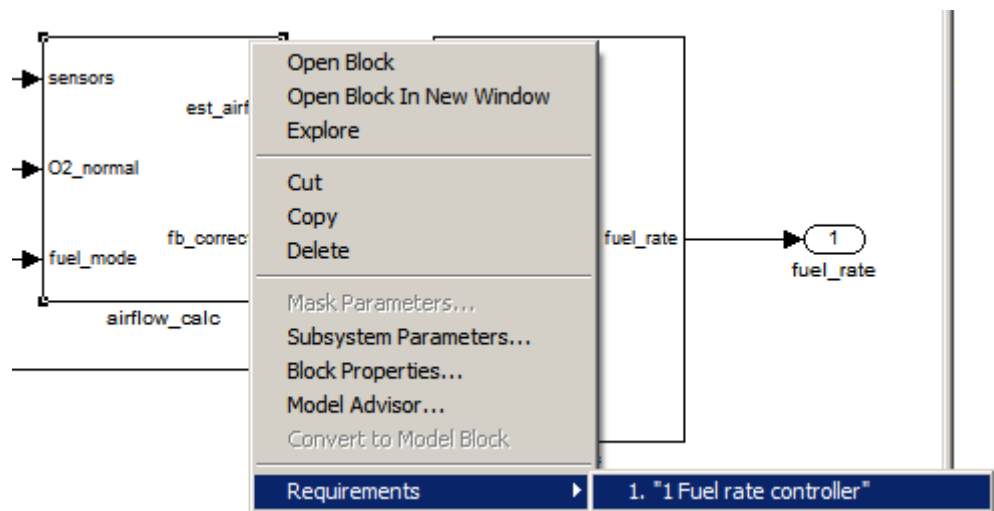
Note To view a demo of using the RMI with an IBM Rational DOORS database, run the Managing Requirements for Fault-Tolerant Fuel Control System (IBM Rational DOORS) demo.

Linking to Requirements in Microsoft Excel Workbooks

In this section...

"Navigating from a Model Object to Requirements in a Microsoft® Excel® Workbook" on page 2-32

"Creating Requirements Links to the Workbook" on page 2-32

"Linking Multiple Model Objects to a Microsoft® Excel® Workbook" on page 2-33

"Changing Requirements Links" on page 2-34

Navigating from a Model Object to Requirements in a Microsoft Excel Workbook

1 Open the demo model:

slvnvdemo_fuelsys_officereq

2 Select **Tools > Requirements > Highlight model** to highlight the model objects with requirements.

3 Right-click the Test inputs Signal Builder block and select **Requirements > 1. "Normal mode of operation"**.

The slvnvdemo_FuelSys_TestScenarios.xlsx file opens, with the associated cell highlighted.

Keep the demo model and Microsoft Excel requirements document open.

For information about creating requirements links in Signal Builder blocks, see "Linking Signal Builder Blocks to Requirements Documents" on page 2-47.

Creating Requirements Links to the Workbook

1 At the top level of the slvnvdemo_fuelsys_officereq model, right-click the speed sensor block and select **Requirements > Edit/Add Links**.

The Requirements dialog box opens.

2 To create a requirements link, click **New**.

3 In the **Description** field, enter:

```
Speed sensor failure
```

You will link the speed sensor block to the **Speed Sensor Failure** information in the Microsoft Excel requirements document.

4 When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

For information about which setting to use for your working environment, see "Resolving the Document Path" on page 5-15.

5 At the **Document** field, click **Browse** to locate and open the slvnvdemo_FuelSys_TestScenarios.xlsx file.

The **Document Type** field information changes to Microsoft Excel.

6 In the workbook, the **Speed sensor failure** information is in cells B22:E22. For the **Location (Type/Identifier)** field, select Sheet range and in the second field, enter B22:E22. (The cell range letters are not case sensitive.)

7 Click **Apply** or **OK** to create the link.

8 To confirm that you created the link, right-click the speed sensor block and select **Requirements > 1. "Speed sensor failure"**.

The workbook opens, with cells B22:E22 highlighted.

Keep the demo model and Microsoft Excel requirements document open.

Linking Multiple Model Objects to a Microsoft Excel Workbook

You can use the same technique to link multiple Simulink and Stateflow objects to a requirement in a Microsoft Excel workbook. The workflow is:

1 In the model window, select the objects to link to a requirement.

2 Right-click one of the selected objects and select **Requirements > Edit/Add Links**.

3 When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

   For information about which setting to use for your working environment, see "Resolving the Document Path" on page 5-15.

4 Use the Requirements dialog to specify information about the Microsoft Excel requirements document, the requirement, and the link.

5 Click **Apply** or **OK** to create the link.

Changing Requirements Links

1 In the `slvnvdemo_fuelsys_officereq` model, right-click the MAP sensor block and select **Requirements > Edit/Add Links**.

   The Requirements dialog box opens displaying the information about the requirements link.
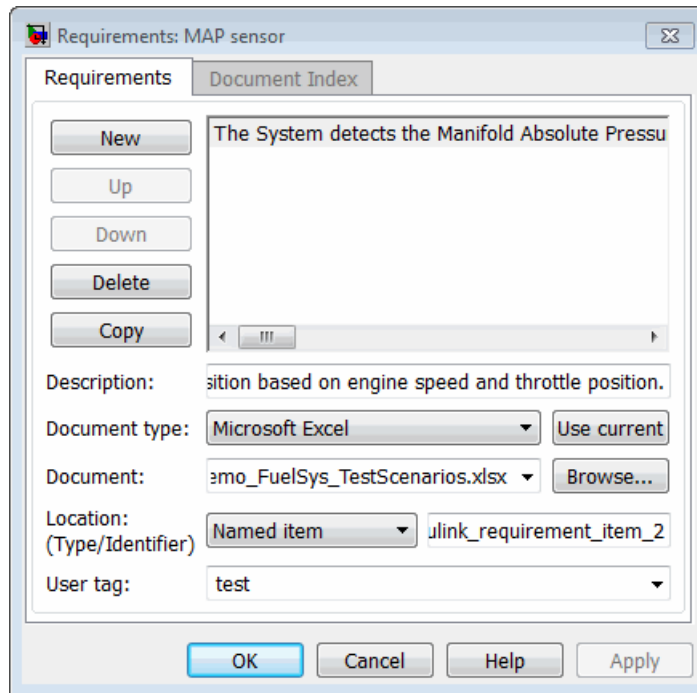
2 In the **Description** field, enter:

```
MAP sensor test scenario
```

The **User tag** field contains the tag `test`. User tags filter requirements for highlighting and reporting.

Note For more information about user tags, see "Filtering Requirements with User Tags" on page 4-25.

3 Click **Apply** or **OK** to save the change.

Keep the demo model open.

Linking to Requirements in MuPAD Notebooks

This example shows how to create a link from a Simulink model to a MuPAD notebook. You use a model that simulates a nonlinear second-order system with the van der Pol equation.

Before beginning this example, create a MuPAD notebook with one or more link targets. This example uses a MuPAD notebook that includes information about solving the van der Pol equation symbolically and numerically.

Note You must have the Symbolic Math Toolbox™ installed on your system to open a MuPAD notebook. For information about creating a MuPAD notebook, see "Creating, Opening, and Saving MuPAD Notebooks".

---

1 Open a demo model for the van der Pol equation:

   `vdp`

2 Right-click a blank area of the model and select **Requirements > Edit/Add Links**.

   You are adding the requirement to the model itself, not to a specific block in the model.

3 Click **New**.

4 In the **Document type** drop-down list, select `MuPAD Notebook`.

5 When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

   For information about which setting to use for your working environment, see "Resolving the Document Path" on page 5-15.

6 Click **Browse** to navigate to the notebook that you want to use.

   Use a notebook that has link targets in it.

7 Make sure that the MuPAD notebook is in a saved state. Any link targets created since the last save do not appear in the RMI document index.

8 To list the link targets, in the Requirements dialog box, click the **Document Index** tab.

This figure shows two link targets.
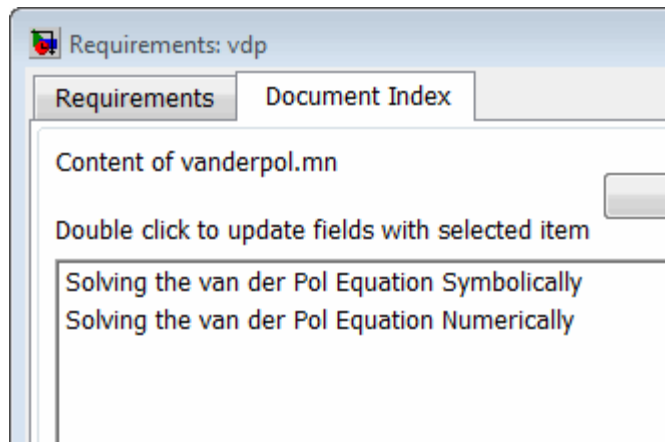
Note These link targets are in a MuPAD notebook that was created for this example. The **Document Index** tab displays only link targets that you have created in your MuPAD notebook.

9 Select a link target name and click **Apply**.

The **Requirements** tab reopens, displaying the details of the newly created link. Unless you have previously entered a description, the link target name appears in the **Description** field.

10 To confirm that you created the link, right-click a blank area of the model and select **Requirement**. The new link is at the top of the submenu.

Creating Requirements Reports

To create the default requirements report for a Simulink model:

1 Open the example model:

```
slvnvdemo_fuelsys_officereq
```

2 Make sure that your current working folder is writable.

3 In the Model Editor, select **Tools > Requirements > Generate report**.

If your model is large and has many requirements links, it takes a few minutes to create the report.

A Web browser window opens with the contents of the report. The following graphic shows the **Table of Contents** for the slvnvdemo_fuelsys_officereq model.

Requirements Report for slvnvdemo_fuelsys_officereq

username

17-Jun-2010 10:57:04

Table of Contents

1. Model Information for "slvnvdemo_fuelsys_officereq"
2. Document Summary for "slvnvdemo_fuelsys_officereq"
3. System - slvnvdemo_fuelsys_officereq
4. System - engine gas dynamics
5. System - fuel rate controller
6. System - Mixing & Combustion
7. System - Airflow calculation
8. System - Sensor correction and Fault Redundancy
9. System - MAP Estimate
10. Chart - control logic

A typical requirements report includes:

- Table of contents
- List of tables
- Per-subsystem sections that include:
  - Tables that list objects with requirements and include links to associated requirements documents
  - Graphic images of objects with requirements
  - Lists of objects with no requirements

For detailed information about requirements reports, see "Creating and Customizing a Requirements Report" on page 4-7.

Linking to Requirements Modeled in Simulink

You can use Simulink to model your design requirements. For example, you can use verification blocks to specify desired system properties and model the design requirements. The Requirements Management Interface (RMI) allows you to create navigation links between the requirements modeled in Simulink, the associated Simulink objects, and related test cases. This example shows how to use the RMI to create and view links to requirements modeled in Simulink.

Open Example Model

Open the Power Window Controller model by typing the command:

```
open_system('slvnvdemo_powerwindowController');
```
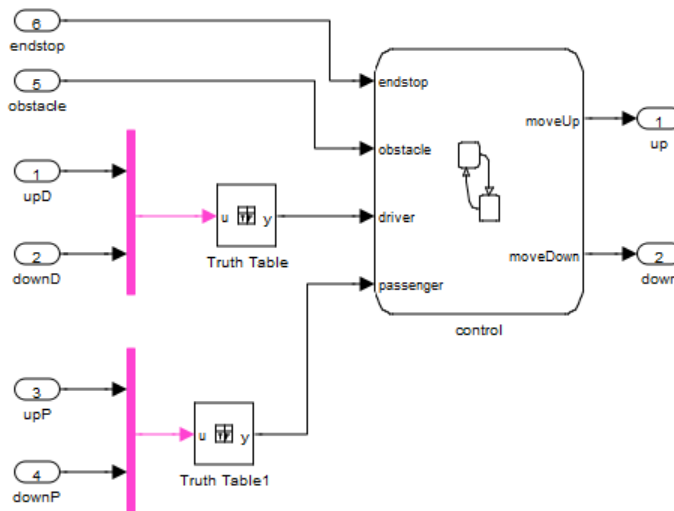

Copyright 1990-2010 The MathWorks, Inc.

Verification Subsystems for Power Window Controller Model

Open the verification model, 'Power Window Controller Temporal Property Specification'. This model specifies properties and requirements of the `slvnvdemo_powerwindowController` model.

Consider the following design requirements for the controller:

1 **Requirement 1** (Obstacle Response) - Whenever an obstacle is detected, the controller shall give the down command for one second. This requirement is modeled in Verification Subsystem2.

2 **Requirement 2** (Autodown feature) - If the driver presses the down button for less than 1 second, the controller keeps issuing the down command until the end has been reached, or the driver presses the up button. This requirement is modeled in Verification Subsystem3

See Design Verifier Temporal Properties demo for more details.

```
open_system('slvnvdemo_powerwindow_vs');
```

Power Window Controller Temporal Property Specification
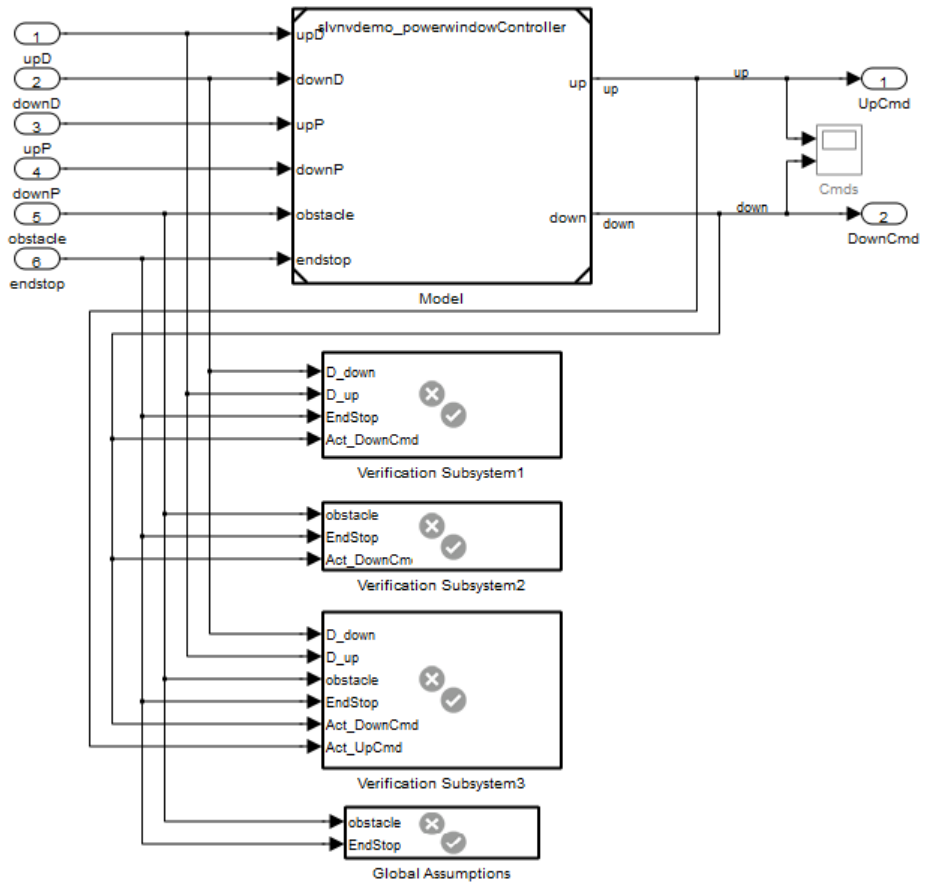
Copyright 1990-2010 The MathWorks, Inc.

Creating RMI Link to a Simulink Object

Create an RMI link from Verification Subsystem2 to the emergencyDown state in the `slvnvdemo_powerwindowController` model.

1 Open slvnvdemo_powerwindowController model

2 Right-click on emergencyDown state and select **Requirements > Select for linking with Simulink**.

3 Right-click on Verification Subsystem2 and select **Requirements > Add link to selected object**.

4 Right-click the Verification Subsystem2. The new RMI link appears at the top of **Requirements** submenu.

5 Close slvnvdemo_powerwindowController model

6 Right-click on Verification Subsystem2. Navigate the new link at the top of the Requirements submenu. Model opens and `emergencyDown` state is highlighted.

Linking Simultaneously to Multiple Simulink Objects

You can link to a multiple selection of Simulink objects. Use the `Shift` key to select all the following objects as in figure below

- iniDriverDown
- autoDriverDown
- after(5,tick) transition out of iniDriverDown
- [driver[1]] transition to autoDriverDown

2-43
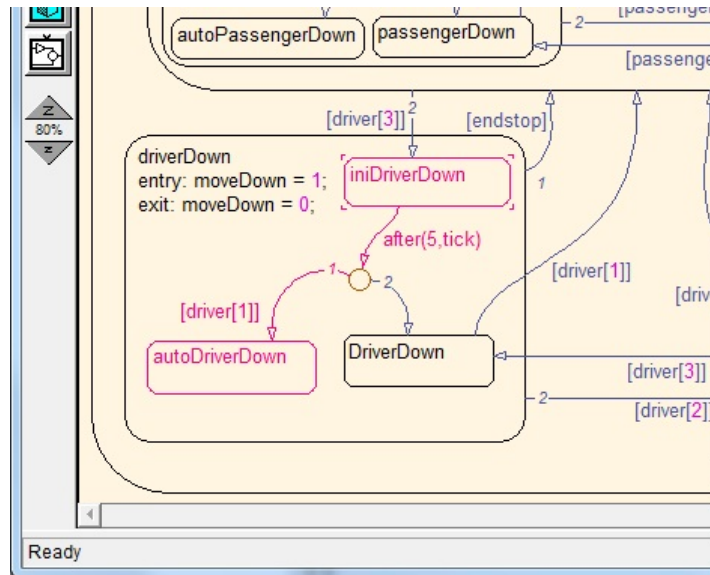
1 Right-click on this group of objects, select **Requirements > Select for linking with Simulink**. Be careful to not loose the selections when you right-click.

2 Right-click on Verification Subsystem3 and select **Requirements > Link to 4 selected objects**.

Linking to a Group of Simulink Objects

1 Right-click on NAND block in `Global Assumptions` and select **Requirements > Select for linking with Simulink**.

2 Drag the mouse across endstop and obstacle inputs in `slvnvdemo_powerwindowController` to select both inputs.

3 Right-click on this group of objects and select **Add link to selected object**. Be careful not to loose the selection.

4 Click on the background of `slvnvdemo_powerwindowController` to loose the group selection.

5 Right-click each input and select **Requirements** to display new links. Click the new link, confirm that NAND is highlighted.

Creating Links for Navigation in Both Directions.

To create links for navigation in both directions:

1 Open **Requirements Settings** dialog box.

2 Select the Selection Linking tab.

3 Enable the **Modify target for navigation to source object**.

Now, when you create a link from one Simulink object to another, a corresponding "return" link is also created.
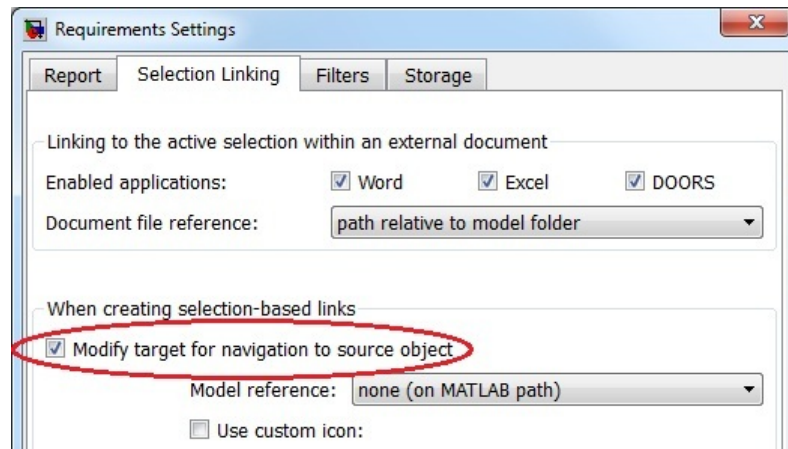
Highlighting and Reporting RMI Links Between Simulink Objects

Create RMI links to Simulink objects in the same way as links to external documents:

1 In the slvnvdemo_powerwindow_vs model window, select **Tools > Requirements > Highlight Model** to highlight all RMI links in the model, including links to Simulink objects.

2 In the slvnvdemo_powerwindow_vs model window, select **Tools > Requirements > Generate Report**.

3 In the generated report, click a hyperlink in any requirements table. This navigates to the corresponding object in Simulink diagram.

Copyright 1990-2010 The MathWorks, Inc.

Table 3.1. Blocks in "slvnvdemo_powerwindow_vs" that have requirements

Name	Requirements	
Verification Subsystem2	Link#1 label:	"slvnvdemo_powerwindowController/.../control/emergencyDown (State)"
	Target:	emergencyDown

Cleanup

Close all open models. Do not save changes.

```
close_system('slvnvdemo_powerwindowController', 0);
close_system('slvnvdemo_powerwindow_vs', 0);
```

Linking Signal Builder Blocks to Requirements Documents

You can create links from a signal group in a Signal Builder block to a requirements document:

1 Open the model:

   sf_car

2 In the sf_car model window, double-click the User Inputs block.

   The Signal Builder dialog box opens, displaying four groups of signals. The Passing Maneuver signal group is the current tab. The RMI associates any requirements links that you add to the current signal group.

3 At the far-right end of the toolbar, click the **Show verification settings** button ![icon]. (You might need to expand the Signal Builder dialog box for this button to become visible.)

   A **Requirements** pane opens on the right-hand side of the Signal Builder dialog box.

Requirements:

No requirements in this group
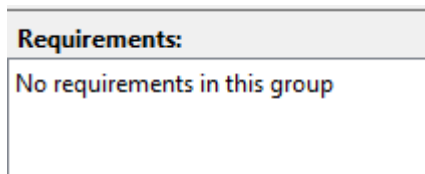
4 Place your cursor in the window, right-click, and select **Edit/Add Links**.

   The Requirements dialog box opens.

5 Click **New**. In the **Description** field, enter User input requirements.

6 When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

   For information about which setting to use for your working environment, see "Resolving the Document Path" on page 5-15.

7 Browse to a requirements document and click **Open**.

8 In the **Location** drop-down list, select **Search text** to link to specified text in the document.

9 Next to the **Location** drop-down list, enter `User Input Requirements`.

10 Click **Apply** to create the link.

11 To verify that the RMI created the link, in the Model Editor, select the User Inputs block, right-click, and select **Requirements**.

The link to the new requirement is the top menu option.

12 Save the `sf_car_linking` model.

Note Links that you create in this way associate requirements information with individual signal groups, not with the entire Signal Builder block.

In this example, switch to a different tab to link a requirement to another signal group.

Linking Signal Builder Blocks to Model Objects

This example shows how to create links from a signal group in a Signal Builder block to a model object:

1 Open the `sf_car` model.

2 Open the `sf_car/shift_logic` chart.

3 Right-click `upshifting` and select **Requirements > Select for linking with Simulink**.

4 In the `sf_car` model window, double-click the User Inputs block.

The Signal Builder dialog box opens, displaying four groups of signals. The Passing Maneuver signal group is the current tab. The RMI associates any requirements links that you add to the current signal group.

5 In the Signal Builder dialog box, click the **Gradual Acceleration** tab.

6 At the far-right end of the toolbar, click the **Show verification settings** button ![icon]. (You might need to expand the Signal Builder dialog box for this button to become visible.)

A **Requirements** pane opens on the right-hand side of the Signal Builder dialog box.
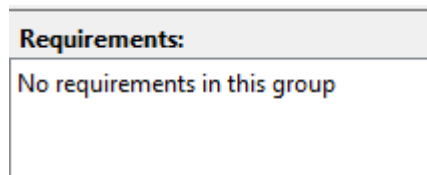
7 Place your cursor in the window, right-click, and select **Edit/Add Links**.

The Requirements dialog box opens.

8 Click **New**. In the **Description** field, enter `Upshifting`.

9 In the **Document type** field, select Simulink. Click **Use current**. The software fills in the field with the **Location: (Type/Identifer)** information for upshifting.
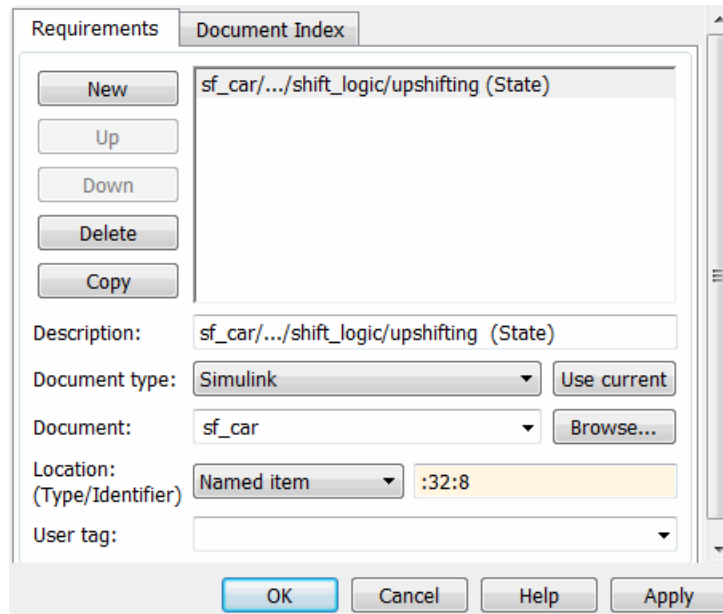
10 Click **Apply** to create the link.

11 In the model window, select the User Inputs block, right-click, and select **Requirements**.

The link to the new requirement is the top menu option.

12 To verify that the links were created, in the sf_car model window, select **Tools > Requirements > Highlight model**.

The blocks with requirements links are highlighted.

13 Close the sf_car model.

Note Links that you create in this way associate requirements information with individual signal groups, not with the entire Signal Builder block.

3

How Is Requirements Link Information Stored?

How Does External Storage Work?

The first time you create links to requirements in a Simulink model, the RMI uses your designated storage preference. When you reopen the model, the RMI loads the internally stored links, or the links from the external file, as long as the file exists with the same name and location as when you last saved the links.

The RMI allows you to save your links file as a different name or in a different folder. However, when you start with the links file in a nondefault location, you must manually load those links into the model. After you load those links, the RMI associates that model with that file and loads the links automatically.

As you work with your model, the RMI stores links using the same storage as the existing links. For example, if you open a model that has internally stored requirements links, any new links you create are also stored internally. This is true even if your preference is set to external storage.

All requirements links must be stored either with the model or in an external file. You cannot mix internal and external storage within a given model.

To watch a demo of the external storage capability using a Simulink model, at the MATLAB command line, enter:

```
slvnvdemo_powerwindow_external
```

Guidelines for External Storage of Requirements Links

If you decide to store requirements links in an external file, keep the following guidelines in mind:

- **When sharing models, use the default name and location.**

  By default, external requirements are stored in a file named *model_name*`.req` in the same folder as the model. If you give your model to others to review the requirements traceability, give the reviewer both the `.mdl` and `.req` files. That way, when you load the model, the RMI automatically loads the links file.

- **Do not rename the model outside of Simulink**.

  If you need to resave the model with a new name or in a different location, in the model window, use **File > Save As**. Selecting this option causes the RMI to resave the corresponding `.req` file using the model name and in the same location as the model.

- **Be aware of unsaved requirements changes.**

  When you change a Simulink model, an asterisk appears next to the model name in the title bar, indicating that the model has unsaved changes. If you are creating new requirements links and storing them externally, this asterisk does not appear because the model file itself has not changed. You can explicitly save the links, or, when you close the model, the RMI prompts you to save the requirements links. When you save the model, the RMI saves the links in the external file.

Specifying Requirements Links Storage

By default, the Requirements Management Interface (RMI) stores requirements links with the model file. In the Requirements Settings dialog box, on the **Storage** tab, you can enable external storage or reenable internal storage. This setting goes into effect immediately and applies to all new models and to existing models with no links to requirements.

If you open a model that already has requirements links, the RMI uses the storage mechanism you used previously with that model, regardless of what your default storage setting is.

To specify the requirements storage setting:

1 In the model window, select **Tools > Requirements > Settings**.

2 In the Requirements Settings dialog box, select the **Storage** tab.

3 Under **Default storage location for requirements links data**:

- To enable external storage, select **Store externally (in a separate *.req file)**.

- To enable internal storage, select **Store internally (embedded in model file)**.

These settings go into effect immediately.

Saving Requirements Links in External Storage

The Requirements Management Interface (RMI) stores externally stored requirements links in a file whose name is based on the model file. Because of this, before you create requirements links to be stored in an external file, you must save the model with a value file name.

You add, modify, and delete requirements links in external storage the same way you do when the requirements links are stored in the model file. The main difference is when you change externally stored links, the model file does not change. The asterisk in the title bar of the model window that indicates a model has unsaved changes does not appear when you change requirements links. However, when you close the model, the RMI asks if you want to save the requirements links modifications.

There are several ways to save requirements links that are stored in an external file, as listed in the following table.

Select...	To...
Tools > Requirements > Save links as	Save the requirements links in an external file using a file name that you specify. The model itself is not saved.
Tools > Requirements > Save links	Save the requirements links in an external file using the default file name, *model_name*.req, or to the previously specified file. The model itself is not saved.
File > Save	Save the current requirements links to an external file named *model_name*.req, or to the previously specified file. Any changes you have made to the model are also saved.
File > Save As	Rename and save the model and the external requirements links. The external file is saved as *new_model_name*.req.

Loading Requirements Links from External Storage

When you open a Simulink model that does not have internally stored requirements links, the RMI tries to load requirements links from an `.req` file, either the default file, or a previously specified file. If that file does not exist, the RMI assumes that this model has no requirements links.

To explicitly load requirements links from an external file:

1 Select **Tools > Requirements > Load links**.

The Select a file to load RMI data dialog box opens, with the default file name or the previously used file name loaded into the **File name** field.

2 Select the file from which to load the requirements links.

3 Click **Open** to load the links from the selected file.

Caution   If your model has unsaved changes to requirements links and you try to load another file, a warning appears.

Moving Internally Stored Requirements Links to External Storage

If you have a model with requirements links that are stored with the model, you can move those links to an external file. When you move internally stored links to a file, the RMI deleted the internally stored links from the model file and saves the model. From this point on, the data exists only in the external file.

1 Open the model that contains internally stored requirements links.

2 Select **Tools > Requirements > Move to file**.

The Select a file to store RMI data dialog box prompts you to save the file with the default name *model_name*.req.

3 Accept the default name, or enter a different file name if required.

4 Click **Save**.

Note Use the default name for externally stored requirements. For more information about this recommendation, see "Guidelines for External Storage of Requirements Links" on page 3-3.

Moving Externally Stored Requirements Links to the Model File

If you have a model with requirements links that are stored in an external file, you can move those links to the model file.

1 Open the model with no internally stored requirements links.

2 Make sure the right set of requirements links are loaded from the external file.

3 Select **Tools > Requirements > Copy to model**.

An asterisk appears next to the model name in the title bar of the model window indicating that your model now has unsaved changes.

4 Save the model with the requirements links.

From this point on, the RMI stores all requirements links internally, in the model file. When you add, modify, or delete links, the changes are stored with the model, even if the **Default storage location for requirements links data** option is set to **Store externally (in a separate .req file)**.

Reviewing Requirements

Highlighting Requirements in a Model

You can highlight a model to see which objects in the model have links to requirements in external documents. You highlight a model from the Model Editor or from the Model Explorer.

In this section...
"Highlighting a Model Using the Model Editor" on page 4-2
"Highlighting a Model Using the Model Explorer" on page 4-3

Highlighting a Model Using the Model Editor

If you are working in the Model Editor and want to see which model objects in the slvnvdemo_fuelsys_officereq model have requirements, follow these steps:

1 Open the demo model:

slvnvdemo_fuelsys_officereq

2 Select **Tools > Requirements > Highlight model**.

Two types of highlighting indicate model objects with requirements:

- Yellow highlighting indicates objects that have requirements links for the object itself.
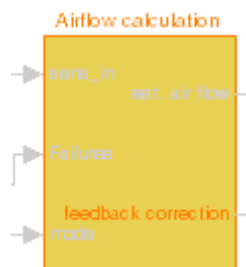
- Orange outline indicates objects, such as subsystems, whose child objects have requirements links.
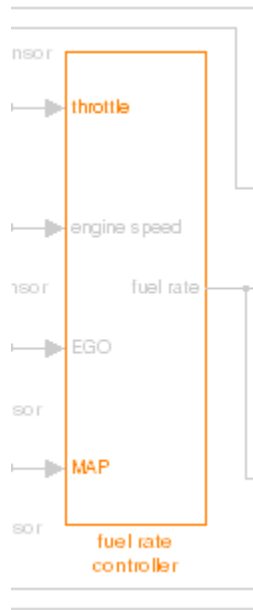
Objects that do not have requirements are colored gray.

3 To remove the highlighting from the model, select
Tools > Requirements > Unhighlight model. Alternatively, you can
right-click anywhere in the model, and select **Remove Highlighting**.

While a model is highlighted, you can still manage the model and its contents.

Highlighting a Model Using the Model Explorer

If you are working in Model Explorer and want to see which model objects
have requirements, follow these steps:

1 Open the demo model:

   slvnvdemo_fuelsys_officereq

2 Select **View > Model Explorer**.

3 To highlight all model objects with requirements, click the **Highlight items with requirements on model** icon ().

   The Model Editor window opens, and all objects in the model with requirements are highlighted.

Note If you are running a 64-bit version of MATLAB, when you navigate to a requirement in a PDF file, the file opens at the top of the page, not at the bookmark location.

Navigating to Requirements from a Model

Navigating from the Model Object

You can navigate directly from a model object to that object's associated requirement. When you take these steps, the external requirements document opens in the application, with the requirements text highlighted.

1 Open the demo model:

   slvnvdemo_fuelsys_officereq

2 Open the fuel rate controller subsystem.

3 To open the linked requirement, right-click the Airflow calculation subsystem and select **Requirements > 1. "Mass airflow estimation"**.

   The Microsoft Word document slvnvdemo_FuelSys_DesignDescription.docx, opens with the section **2.1 Mass airflow estimation** selected.

Note If you are running a 64-bit version of MATLAB, when you navigate to a requirement in a PDF file, the file opens at the top of the page, not at the bookmark location.

Navigating from a System Requirements Block

Sometimes you want to see all the requirements links at a given level of the model hierarchy. In such cases, you can insert a System Requirements block to collect all requirements links in the model or subsystem. The System Requirements block does not list requirements links for any blocks for that model or subsystem.

For example, you can insert the System Requirements block at the top level of the slvnvdemo_fuelsys_officereq model, and navigate to the requirements using the links inside the block.

1 Open the demo model:

slvnvdemo_fuelsys_officereq

2 Select **Tools > Requirements > Highlight model**.

3 In the slvnvdemo_fuelsys_officereq model, open the fuel rate controller subsystem.

The Airflow calculation subsystem has a requirements link.

4 Open the Airflow calculation subsystem.

5 Select **View > Library Browser**

6 On the **Libraries** pane, click the **Simulink Verification and Validation** library.

This library contains only one block—the System Requirements block.

7 Drag a System Requirements block into the Airflow calculation subsystem.

The RMI software collects and displays any requirements links for that subsystem.

8 Double-click **1. "Mass airflow subsystem"**.

The Microsoft Word document, slvnvdemo_FuelSys_DesignDescription.docx, opens, with the section **2.1 Mass airflow estimation** selected.

Creating and Customizing a Requirements Report

In this section...
"Creating a Default Requirements Report" on page 4-7
"Reporting on Requirements in Model Blocks" on page 4-15
"Customizing the Requirements Report" on page 4-17
"Generating Requirements Reports Using Simulink" on page 4-22

Creating a Default Requirements Report

If you have a model that contains links to external requirements documents, you can create an HTML report that contains summarized and detailed information about those links. In addition, the report contains links that allow you to navigate to both the model and to the requirements documents.

You can generate a default report with information about all the requirements associated with a model and its objects.

Note If the model for which you are creating a report contains Model blocks, see "Reporting on Requirements in Model Blocks" on page 4-15.

Before you generate the report, add a requirement to a Stateflow chart to see information that the requirements report contains about Stateflow charts:

1 Open the demo model:

   slvnvdemo_fuelsys_officereq

2 Open the fuel rate controller subsystem.

3 Open the Microsoft Word requirements document:

   matlabroot/toolbox/slvnv/rmidemos/fuelsys_req_docs/slvnvdemo_FuelSys_RequirementsSpecification.docx

4 Create a link from the control logic Stateflow chart to a location in this document.

5 Keep the demo model open, but close the requirements document.

To generate a default requirements report for the
`slvnvdemo_fuelsys_officereq` model:

1 Select **Tools > Requirements > Generate Report**.

The Requirements Management Interface (RMI) searches through all the
blocks and subsystems in the model for associated requirements. The RMI
generates and displays a complete report in HTML format.

The report is saved with the default name, *model_name*`_requirements.html`.
If you generate a subsequent report on the same model, the new report file
overwrites any earlier report file.

The report contains the following content:

- "Table of Contents" on page 4-8
- "List of Tables" on page 4-9
- "Model Information" on page 4-10
- "Documents Summary" on page 4-10
- "System" on page 4-11
- "Chart" on page 4-14

Table of Contents

The **Table of Contents** lists the major sections of the report. There is one
System section for the top-level model and one **System** section for each
subsystem, Model block, or Stateflow chart.

Click a link to view information about a specific section of the model.

Requirements Report for slvnvdemo_fuelsys_officereq

username

17-Jun-2010 10:57:04

Table of Contents

List of Tables

The **List of Tables** includes links to each table in the report.

List of Tables

Model Information

The **Model Information** contains general information about the model, such as when the model was created and when the model was last modified.

Chapter 1. Model Information for "slvnvdemo_fuelsys_officereq"

Table 1.1. slvnvdemo_fuelsys_officereq Version Information

ModelVersion	1.159	ConfigurationManager	none
Created	Tue Jun 02 16:11:43 1998	Creator	The MathWorks Inc.
LastModifiedDate	Sat Jun 12 02:31:44 2010	LastModifiedBy	

Documents Summary

The **Documents Summary** section lists all the requirements documents to which objects in the slvnvdemo_fuelsys_officereq model link, along with some additional information about each document.

Chapter 2. Document Summary for "slvnvdemo_fuelsys_officereq"

Table 2.1. Requirements documents linked in model

ID	Document paths stored in the model	Last modified	# links
DOC1	ERROR: unable to locate slvnvdemo_FuelSys_RequirementsSpecification.docx	unknown	1
DOC2	fuelsys_req_docs\slvnvdemo_FuelSys_DesignDescription.docx	29-Oct-2009 10:56:01	8
DOC3	fuelsys_req_docs\slvnvdemo_FuelSys_RequirementsSpecification.docx	29-Oct-2009 10:56:02	6
DOC4	fuelsys_req_docs\slvnvdemo_FuelSys_TestScenarios.xlsx	29-Oct-2009 10:56:03	2

- **ID** — The ID. In this example, **DOC1**, **DOC2**, **DOC3**, and **DOC4** are short names for the requirements documents linked from this model.

  Before you generate a report, in the Settings dialog box, on the **Reports** tab, if you select **User document IDs in requirements tables**, links with these short names are included throughout the report when referring to a requirements document. When you click a short name link in a report, the requirements document associated with that document ID opens.

  When your requirements documents have long path names that can clutter the report, select the **User document IDs in requirements tables** option. This option is disabled by default, as you can see in the examples in this section.

- **Document paths stored in the model** — Click this link to open the requirements document in its native application.

- **Last modified** — The date the requirements document was last modified.

- **# links** — The total number of links to a requirements document.

System

Each **System** section includes:

- An image of the model or model object. The objects with requirements are highlighted.

Chapter 3. System - slvnvdemo_fuelsys_officereq

Fault-Tolerant Fuel Control System
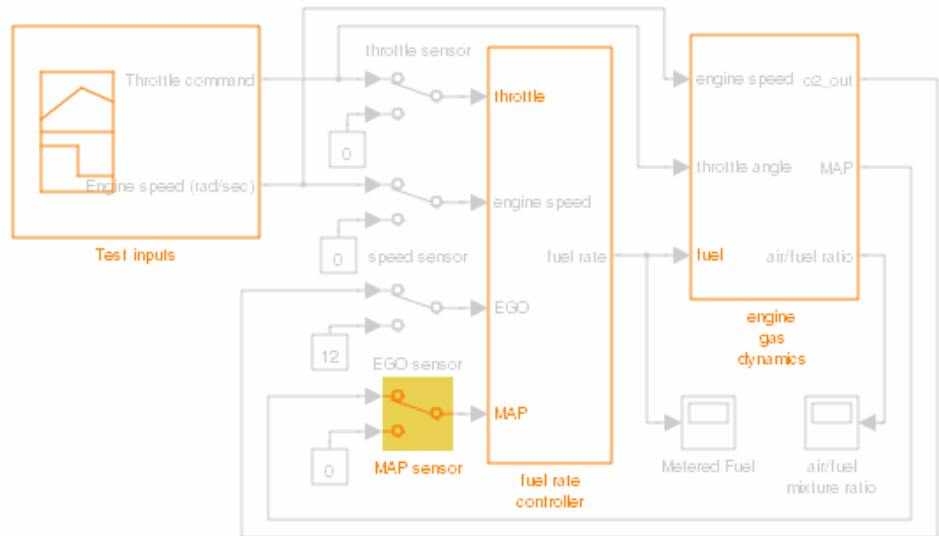
Copyright 1997-2009 The MathWorks, Inc.

- A list of requirements associated with the model or model object. In this example, click the target document name to open the requirements document associated with the `slvnvdemo_fuelsys_officereq` model.

Table 3.1. slvnvdemo_fuelsys_officereq Requirements

Link#	Description		Target (document name and location ID)
1	Label:	Design Description Microsoft Word Document	slvnvdemo_FuelSys_DesignDescription.docx

- A list of blocks in the top-level model that have requirements. In this example, only the MAP sensor block has a requirement at the top level.

Click the link next to **Target:** to open the requirements document associated with the MAP sensor block.

Table 3.2. Blocks in "slvnvdemo_fuelsys_officereq" that have requirements

Name	Requirements	
MAP sensor	Link#1 label:	"The System detects the Manifold Absolute Pressure sensor short to ground or open circuit by monitoring when the signal is below a calibratable threshold. The failure is detected within 100mSec of the occurance and the MAP sensor reading is reverted to an estimated throttle position based on engine speed and throttle position."
	Target:	fuelsys_req_docs\slvnvdemo_FuelSys_TestScenarios.xlsx, at 'Simulink_requirement_item_2'

The preceding table does not include these blocks in the top-level model because:

- The fuel rate controller and engine gas dynamics subsystems are in dedicated chapters of the report.

- The report lists Signal Builder blocks separately, in this example, in Table 3.3.

• A list of requirements associated with each signal group in any Signal Builder block, and a graphic of that signal group. In this example, the Test inputs Signal Builder block in the top-level model has one signal group that has a requirement link. Click the link under **Target (document name and location ID)** to open the requirements document associated with this signal group in the Test inputs block.

Table 3.3. Test inputs : Normal operation signal requirements

Link#	Description		Target (document name and location ID)
1	Label:	Normal mode of operation	slvnvdemo_FuelSys_TestScenarios.xlsx, at 'Simulink_requirement_item_3'

Chart

Each **Chart** section reports on requirements in Stateflow charts, and includes:

- A graphic of the Stateflow chart that identifies each state.
- A list of elements that have requirements.

  To navigate to the requirements document associated with a chart element, click the link next to **Target**.

Table 10.1. Stateflow objects with requirements

Name	Requirements	
warmup	Link#1 label:	"During a calibratible warm up period the oxygen sensor correction shall be disabled."
	Target:	fuelsys_req_docs\slvnvdemo_FuelSys_RequirementsSpecification.docx, at 'Simulink_requirement_item_3'
[speed==0 & press < zero_thresh]/...	Link#1 label:	"Speed sensor failure detection"
	Target:	fuelsys_req_docs\slvnvdemo_FuelSys_DesignDescription.docx, at 'Simulink_requirement_item_6'
Rich_Mixture	Link#1 label:	"Enriched mixture usage"
	Target:	fuelsys_req_docs\slvnvdemo_FuelSys_DesignDescription.docx, at 'Simulink_requirement_item_4'
Warmup	Link#1 label:	"During a calibratible warm up period the oxygen sensor correction shall be disabled."
	Target:	fuelsys_req_docs\slvnvdemo_FuelSys_RequirementsSpecification.docx, at 'Simulink_requirement_item_3'

Reporting on Requirements in Model Blocks

If your model contains Model blocks that reference external models, the default report does not include information about requirements in the referenced models. To generate a report that includes requirements information for referenced models, you must have a license for the Simulink Report Generator™ software. The report includes the same information and graphics for referenced models as it does for the top-level model.

If you have a Simulink Report Generator license, before generating a requirements report, take the following steps:

1 Open the model for which you want to create a requirements report.

2 To open the template for the default requirements report, at the MATLAB command prompt, enter:

```
setedit requirements
```

3 In the Simulink Report Generator software window, in the far-left pane, click the `Model Loop` component.
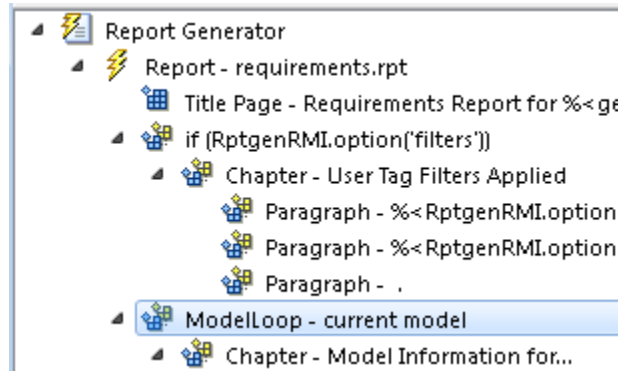
4 On the far-right pane, locate the **Model reference** field. If you cannot see the drop-down arrow for that field, expand the pane.
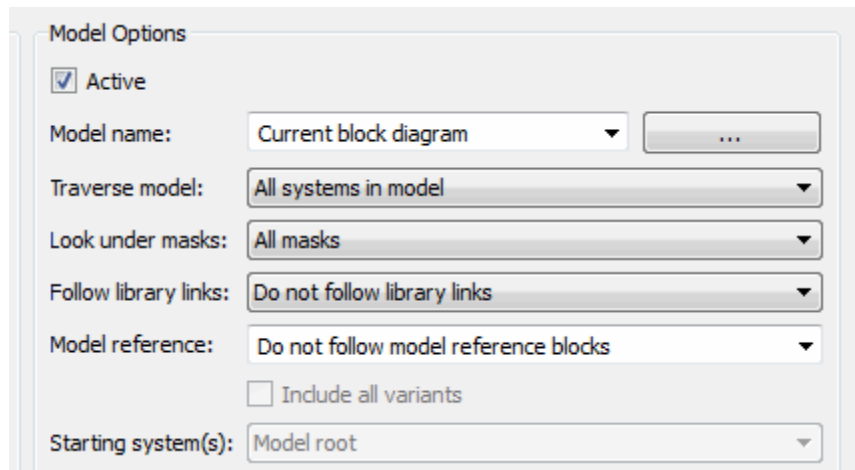
5 In the **Model reference** field drop-down list, select `Follow all model reference blocks`.

6 To generate a requirements report for the open model that includes information about referenced models, click the **Report** icon ⚡.

Customizing the Requirements Report

The Requirements Management Interface (RMI) uses the Simulink Report Generator software to generate the requirements report. You can customize the requirements report using the RMI or the Simulink Report Generator software:

- "Customizing a Requirements Report Using the RMI Settings" on page 4-17
- "Customizing the Report Using the Simulink® Report Generator™ Software" on page 4-21

Customizing a Requirements Report Using the RMI Settings

There are several options for customizing a requirements report using the Requirements Settings dialog box.
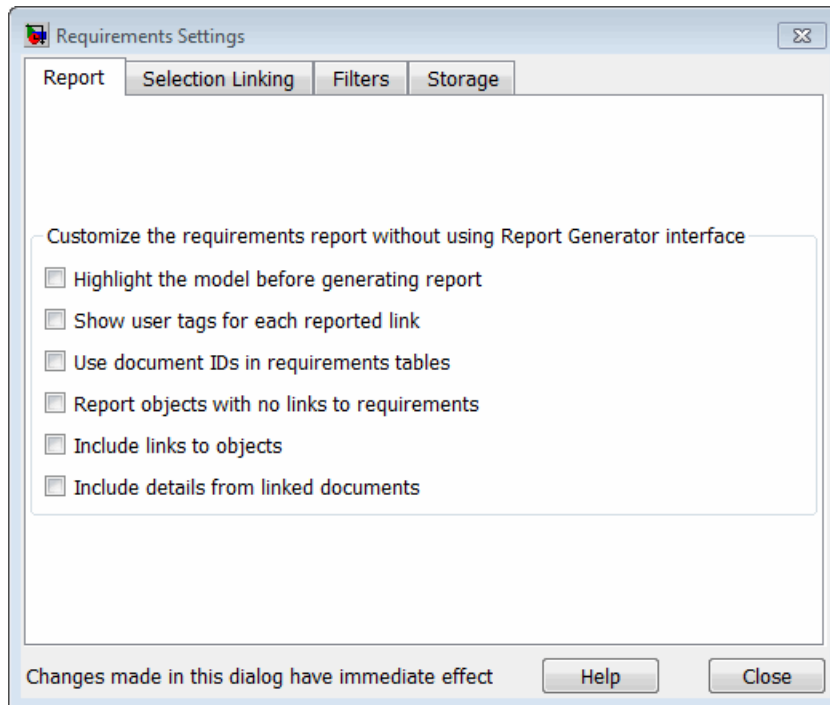
On the **Report** tab, select options that specify the contents that you want in the report.

Requirements Settings Report Option	Description
Highlight the model before generating report	Enables highlighting of Simulink objects with requirements in the report graphics.
Show user tags for each reported link	Lists the user tags, if any, for each reported link.
Use document IDs in requirements tables	Uses a document ID, if available, instead of a path name in the tables of the requirements report. This capability prevents long path names to requirements documents from cluttering the report tables.
Report objects with no links to requirements	Includes lists of model objects that have no requirements.
Include links to objects	Includes hypertext links from the report to model objects.
Include details from linked documents	Includes additional content from linked requirements. The following requirements documents are supported: • Microsoft Word • Microsoft Excel • IBM Rational DOORS

To see how these options affect the content of the report:

1 Open the slvnvdemo_fuelsys_officereq model:

  slvnvdemo_fuelsys_officereq

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

3 In the Requirements Settings dialog box, click the **Report** tab.

4 For this example, select **Highlight the model before generating the report**.

When you select this option, before generating the report, the graphics of the model that are included in the report are highlighted so that you can easily see which objects have requirements.

5 To close the Requirements Settings dialog box, click **Close**.

6 Generate a requirements report by selecting **Tools > Requirements > Generate Report**.

The requirements report opens in a browser window so that you can review the content of the report.

7 If you do not want to overwrite the current report when you regenerate the requirements report, rename the HTML file, for example, `slvnvdemo_fuelsys_officereq_requirements_old.html`.

The default report file name is `model_name_requirements.html`.

8 In the model window, select **Tools > Requirements > Settings**.

9 On the **Report** tab, select:

- **Show user tags for each reported link** — The report lists the user tags (if any) associated with each requirement.

- **Include details from linked documents** — The report includes additional details for requirements in the following types of requirements documents.

Requirements Document Format	Includes in the Report
Microsoft Word	Full text of the paragraph or subsection of the requirement, including tables.
Microsoft Excel	If the target requirement is a group of cells, the report includes all those cells as a table. If the target requirement is one cell, the report includes that cell and all the cells in that row to the right of the target cell.
IBM Rational DOORS	By default, the report includes: – **DOORS Object Heading** – **DOORS Object Text** – All other attributes except **Created Thru**, attributes with empty string values, and system attributes that are false. Use the `RptgenRMI.doorsAttribs` function to include or exclude specific attributes or groups of attributes.

10 Close the Requirements Settings dialog box.

11 Generate a new requirements report by selecting **Tools > Requirements > Generate Report**.

12 Compare this new report to the report that you renamed in step 7 on page 4-19:

- User tags associated with requirements links are included.

- Details from the requirement content are included as specified in step 9 on page 4-19.

13 When you are done reviewing the report, close the report and the model.

To see a demo about including details in the requirements report, enter the following command at the MATLAB prompt:

slvnvdemo_powerwindow_report

Customizing the Report Using the Simulink Report Generator Software

If you have a license for the Simulink Report Generator software, you can further modify the default requirements report.

At the MATLAB command prompt, enter the following command:

setedit requirements

The Report Explorer GUI opens the requirements report template that the RMI uses when generating a requirements report. The report template contains Simulink Report Generator components that define the structure of the requirements report.

If you click a component in the middle pane, the options that you can specify for that component appear in the right-hand pane. For detailed information about using a particular component to customize your report, click **Help** at the bottom of the right-hand pane.

In addition to the standard report components, Simulink Report Generator provides components specific to the RMI in the Requirements Management Interface category.

Simulink Report Generator Component	Report Information
Missing Requirements Block Loop	Applies all child components to blocks that have no requirements
Missing Requirements System Loop	Applies all child components to systems that have no requirements
Requirements Block Loop	Applies all child components to blocks that have requirements
Requirements Documents Table	Inserts a table that lists requirements documents
Requirements Signal Loop	Applies all child components to signal groups with requirements

Simulink Report Generator Component	Report Information
Requirements Summary Table	Inserts a property table that lists requirements information for blocks with associated requirements
Requirements System Loop	Applies all child components to systems with requirements
Requirements Table	Inserts a table that lists system and subsystem requirements

To customize the requirements report, you can:

- Add or delete components.
- Move components up or down in the report hierarchy.
- Customize components to specify how the report presents certain information.

For more information, see *Simulink Report Generator User's Guide*.

Generating Requirements Reports Using Simulink

When you have a model open in Simulink, the Model Editor provides two options for creating requirements reports:

- "System Design Description Report" on page 4-22
- "Design Requirements Report" on page 4-23

System Design Description Report

The System Design Description report describes a system design represented by the current Simulink model.

You can use the System Design Description report to:

- Review a system design without having the model open.
- Generate summary and detailed descriptions of the design.

- Assess compliance with design requirements.

- Archive the system design in a format independent of the modeling environment.

- Build a customized version of the report using the Simulink Report Generator software.

To generate a System Design Description report that includes requirements information:

1 Open the model for which you want to create a report.

2 Select **File > Reports > System Design Description**.

3 In the Design Description dialog box, select **Requirements traceability**.

4 Select any other options that you want for this report.

5 Click **Generate**.

As the software is generating the report, the status appears in the MATLAB command window.

The report name is the model name, followed by a numeral, followed by the extension that reflects the document type (`.pdf`, `.html`, etc.).

If your model has linked requirements, the report includes a chapter, **Requirements Traceability** , that includes:

- Lists of model objects that have requirements with hyperlinks to display the objects

- Images of each subsystem, highlighting model objects with requirements

Design Requirements Report

In the Model Editor, the menu option **File > Reports > Design Requirements** creates a requirements report, as described in "Creating a Default Requirements Report" on page 4-7. This menu option is equivalent to **Tools > Requirements > Generate report**.

To specify options for the report, select **Tools > Requirements > Settings**. Before generating the report, on the **Report** tab, set the options that you want. For detailed information about these options, see "Customizing the Requirements Report" on page 4-17.

Filtering Requirements with User Tags

In this section...
"Understanding Requirements Filtering" on page 4-25
"Applying a User Tag to a Requirement" on page 4-25
"Filtering, Highlighting, and Reporting with User Tags" on page 4-27
"Applying User Tags During Selection-Based Linking" on page 4-28
"Configuring Requirements Filtering" on page 4-30

Understanding Requirements Filtering

User tags are user-defined keywords that you associate with specific requirements. With user tags, you can highlight a model or generate a requirements report for a model in the following ways:

- Highlight or report only those requirements that have a specific user tag.

- Highlight or report only those requirements that have one of several user tags.

- Do not highlight and report requirements that have a specific user tag.

Applying a User Tag to a Requirement

To apply one or more user tags to a newly created requirement:

1 Open the demo model:

   slvnvdemo_fuelsys_officereq

2 Open the fuel rate controller subsystem.

3 To open the requirements document, right-click the Airflow calculation subsystem and select **Requirements > Edit/Add Links**.

   The Requirements dialog box opens with the details about the requirement that you created.
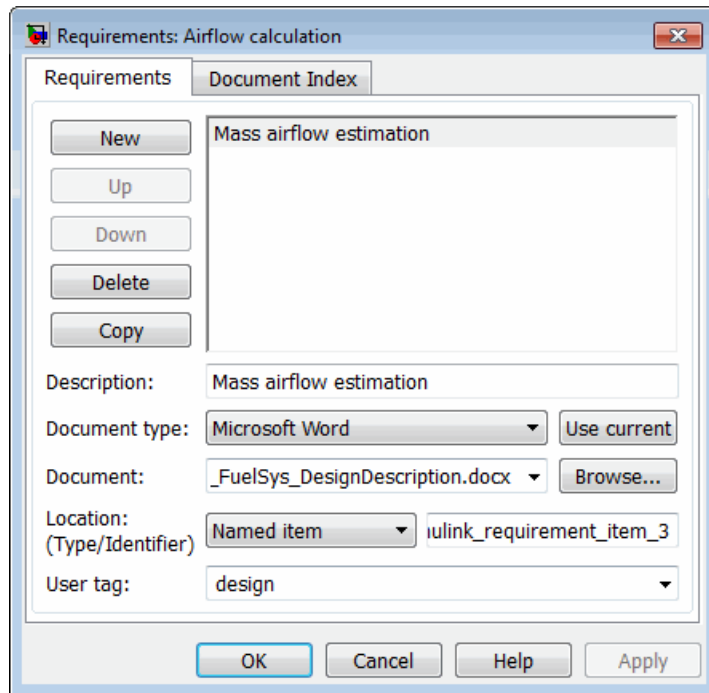
4 In the **User tag** field, enter one or more keywords, separated by commas, that the RMI can use to filter requirements. In this example, after `design`, enter a comma, followed by the user tag `test` to specify a second user tag for this requirement.

User tags:

- Are not case sensitive.

- Can consist of multiple words. For example, if you enter `design requirement`, the entire phrase constitutes the user tag. Separate user tags with commas.

5 Click **Apply** or **OK** to save the changes.

Filtering, Highlighting, and Reporting with User Tags

The `slvnvdemo_fuelsys_officereq` model includes several requirements with the user tag `design`. This section describes how to highlight only those model objects that have the user tag, `test`.

1 In the Model Editor, remove any highlighting from the `slvnvdemo_fuelsys_officereq` model by selecting **Tools > Requirements > Unhighlight model**.

2 Select **Tools > Requirements > Settings**.

3 In the Requirements Settings dialog box, click the **Filters** tab.
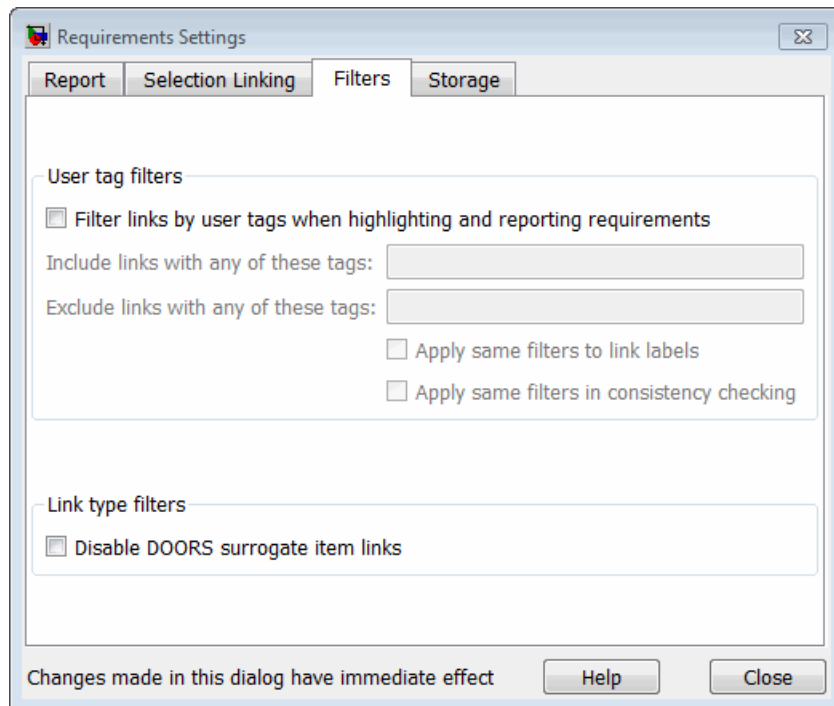
4 To enable filtering with user tags, click the **Filter links by user tags when highlighting and reporting requirements** option.

5 To include only those requirements that have the user tag, `test`, enter `test` in the **Include links with any of these tags** field.

6 Click **Close**.

7 In the Model Editor, select **Tools > Requirements > Highlight model**.

The RMI highlights only those model objects whose requirements have the user tag `test`, for example, the MAP sensor.

8 Reopen the Requirements Settings dialog box to the **Filters** tab.

9 In the **Include links with any of these tags** field, delete `test`. In the **Exclude links with any of these tags** field, add `test`.

In the model, the highlighting changes to exclude objects whose requirements have the `test` user tag. The MAP sensor and Test inputs blocks are no longer highlighted.

10 In the Model Editor, select **Tools > Requirements > Generate Report**.

The report does not include information about objects whose requirements have the `test` user tag.

Applying User Tags During Selection-Based Linking

When creating a succession of requirements links, you can apply the same user tags to all links automatically. This capability, also known as *selection-based linking*, is available only when you are creating links to selected objects in the requirements documents.

When creating selection-based links, specify one or more user tags to apply to requirements:

1 In the Model Editor, select **Tools > Requirements > Settings**.

2 Select the **Selection Linking** tab.
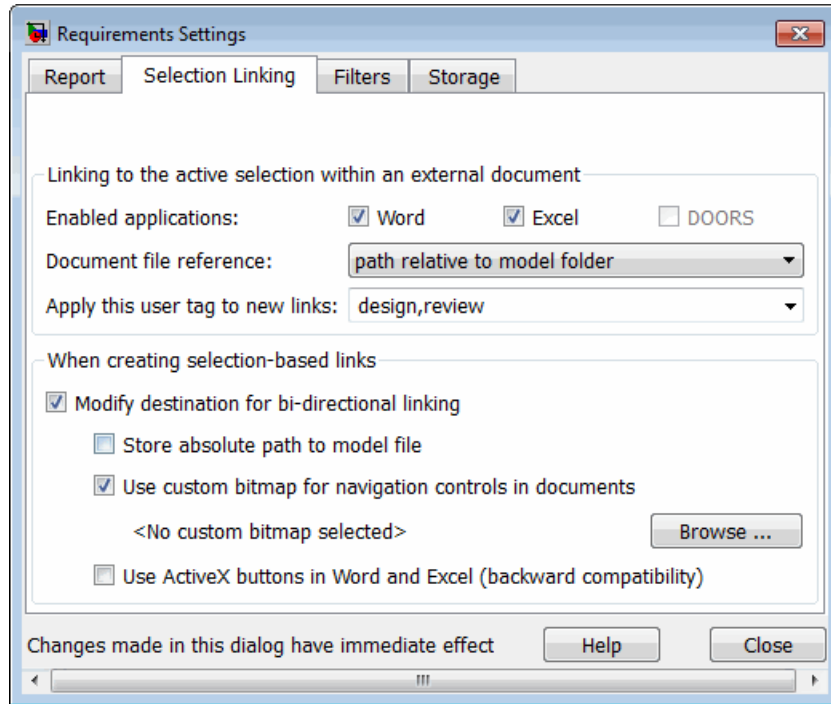
3 In the **Apply this user tag to new links** field, enter one or more user tags, separated by commas.

The RMI applies these user tags to all selection-based requirements links that you create.

4 Click **Close** to close the Requirements Settings dialog box.

5 In a requirements document, select the specific requirement text.

6 Right-click a model object and select **Requirements**.

The selection-based linking options specify which user tags the RMI applies to the link that you create. In the following example, you can apply the user tags `design` and `review` to the link that you create to your selected text.
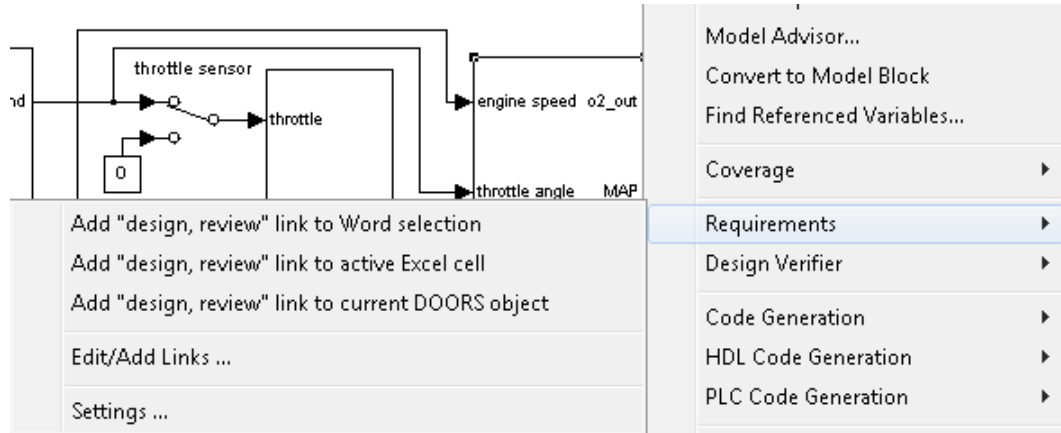
Configuring Requirements Filtering

In the Requirements Settings dialog box, on the **Filters** tab, are the following options for filtering requirements in a model.

Option	Description
Filter links by user tags when highlighting and reporting requirements	Enables filtering for highlighting and reporting, based on specified user tags.
Include links with any of these tags	Includes information about all requirements that have any of the specified user tags. Separate multiple user tags with commas.
Exclude links with any of these tags	Excludes information about all requirements that have any of the specified user tags. Separate multiple user tags with commas or spaces.
Apply same filters in context menus	Disables link labels in context menus if any of the specified filters are satisfied, for example, if a requirement has a designated user tag.

Option	Description
Apply same filters in consistency checking	Includes or excludes requirements with specified user tags when running a consistency check between a model and its associated requirements documents.
Under **Link type filters**, **Disable DOORS surrogate item links in context menus**	Disables links to IBM Rational DOORS surrogate items from the context menus when you right-click a model object. This option does not depend on current user tag filters.

5

Requirements Links Maintenance

Checking Requirements Links

Requirements links in a model can become outdated when requirements change over time. Similarly, links in requirements documents may become invalid when your Simulink model changes, for example, when the model, or objects in the model, are renamed, moved, or deleted. The Simulink Verification and Validation software provides tools that allow you to detect and resolve these problems in the model or in the requirements document.

In this section...

"Checking and Fixing Requirements Links in a Simulink Model" on page 5-2

"Checking and Fixing ActiveX-Based Links in Requirements Documents" on page 5-9

Checking and Fixing Requirements Links in a Simulink Model

- "Checking Requirements Links" on page 5-2
- "Fixing Inconsistent Links" on page 5-5

Checking Requirements Links

To make sure that every requirements link in your Simulink model has a valid target in a requirements document, run the Model Advisor Requirements consistency checks:

1 Open the demo model:

```
slvnvdemo_fuelsys_officereq
```

2 Open the Model Advisor to run a consistency check by selecting **Tools > Requirements > Check Consistency**.

In the **Requirement consistency checking** category, all the checks are selected. For this tutorial, keep all the checks selected.

☑ 📁 Requirement consistency checking
   ☑ 🗒 Identify requirement links with missing documents
   ☑ 🗒 Identify requirement links that specify invalid locations within documents
   ☑ 🗒 Identify selection-based links having description fields that do not match their requirements document text
   ☑ 🗒 Identify requirement links with path type inconsistent with preferences
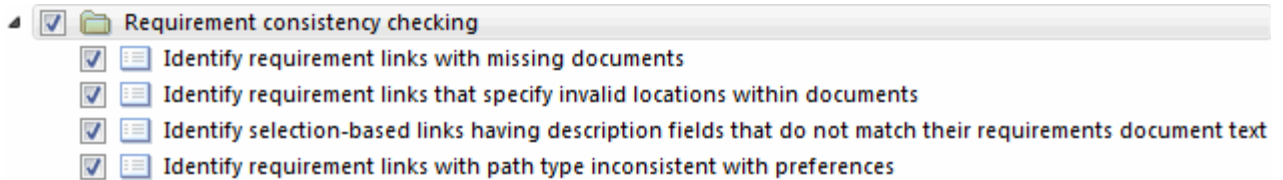
These checks identify the following problems with your model requirements.

Consistency Check	Problem Identified
Identify requirement links with missing documents	The Model Advisor cannot find the requirements document. This might indicate a problem with the path to the requirements document.
Identify requirement links that specify invalid locations within documents	The Model Advisor cannot find the designated location for the requirement. This check is implemented for: • Microsoft Word documents • Microsoft Excel documents • IBM Rational DOORS documents • Simulink objects
Identify selection-based links having description fields that do not match their requirements document text	The **Description** field for the link does not match the requirements document text. When you create selection-based links, the Requirements Management Interface (RMI) saves the selected text in the link **Description** field. This check is implemented for: • Microsoft Word documents • Microsoft Excel documents • IBM Rational DOORS documents • Simulink objects
Identify requirement links with path type inconsistent with preferences	The path to the requirements document does not match the **Document file reference** field in the Requirements Settings dialog box

Consistency Check	Problem Identified
	Selection Linking tab. This might indicate a problem with the path to the requirements document. On Linux® systems, this check is named **Identify requirement links with absolute path type**. The check reports a warning for each requirements links that uses an absolute path. **Note** For information about how the RMI resolves the path to the requirements document, see "Resolving the Document Path" on page 5-15.

The Model Advisor checks to see if any applications that have link targets are running:

- If your model has links to Microsoft Word or Microsoft Excel documents, the consistency check requires that you close all instances of those applications. If you have one of these applications open, it displays a warning and does not continue the checks. The consistency checks must verify up-to-date stored copies of the requirements documents.

- If your model has links to DOORS requirements, you must be logged in to the DOORS software. Your DOORS database must include the module that contains the target requirements.

3 For this tutorial, make sure that you close both Microsoft Word and Microsoft Excel.

4 Click **Run Selected Checks**.

After the check is complete:

- The green circles with the check mark indicate that two checks passed.

- The yellow triangles with the exclamation point indicate that two checks generated warnings.
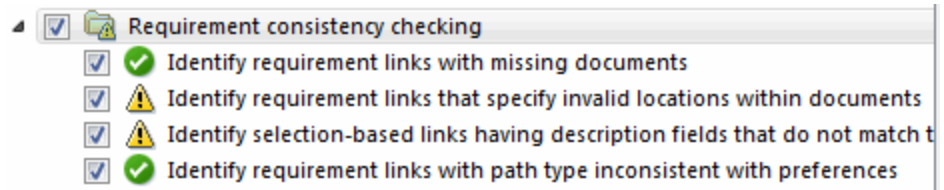
The right-hand pane shows that two checks passed and two checks had warnings. The pane includes a link to the HTML report.
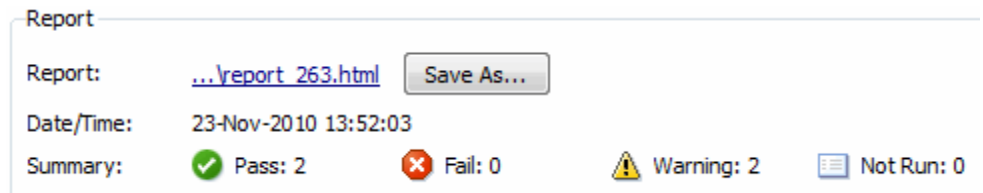
Keep the Model Advisor open. The next section describes how to interpret and fix the inconsistent links.

Note To view a demo that uses the Model Advisor to check requirements links in an IBM Rational DOORS database, run the Managing Requirements for Fault-Tolerant Fuel Control System (IBM Rational DOORS) demo.

Fixing Inconsistent Links

In "Checking and Fixing Requirements Links in a Simulink Model" on page 5-2, three requirements consistency checks generate warnings in the slvnvdemo_fuelsys_officereq model.

- "Resolve Warning: Identify requirement links that specify invalid locations within documents" on page 5-6

- "Resolve Warning: Identify selection-based links having description fields that do not match their requirements document text" on page 5-7

Resolve Warning: Identify requirement links that specify invalid locations within documents. To fix the warning about attempting to link to an invalid location in a requirements document:

1 In the Model Advisor, select **Identify requirement links that specify invalid locations within documents** to display the description of the warning.

Inconsistencies:

The following requirements link to invalid locations within their documents. The specified location (e.g., bookmark, line number, anchor) within the requirements document could not be found. To resolve this issue, edit each requirement and specify a valid location within its requirements document.

Block	**Requirements**
slvnvdemo_fuelsys_officereq/fuel rate controller/Sensor correction and Fault Redundancy/Terminator 1	This section will be deleted

This check identifies a link that specifies a location that does not exist in the Microsoft Word requirements document, slvnvdemo_FuelSys_DesignDescription.docx. The link originates in the Terminator1 block. In this example, the target location in the requirements document was deleted after the requirement was created.

2 Get more information about this link:

    a To navigate to the Terminator1 block, under **Block**, click the hyperlink.

    b To open the Requirements dialog box for this link, under **Requirements**, click the hyperlink.

3 To fix the problem from the Requirements dialog box, do one of the following:

- In the **Location** field, specify a valid location in the requirements document.

- Delete the requirements link by selecting the link and clicking **Delete**.

4 In the Model Advisor, select the **Requirement consistency checking** category of checks.

5 Click **Run Selected Checks** again, and verify that the warning no longer occurs.

Resolve Warning: Identify selection-based links having description fields that do not match their requirements document text. To fix the warnings about the **Description** field not matching the requirements document text:

1 In the Model Advisor, click **Identify selection-based links having description fields that do not match their requirements document text** to display the description of the warning.

Unable to check:

- Failed to locate item **@Simulink_requirement_item_7** in **fuelsys_req_docs\slvnvdemo_FuelSys_DesignDescription.docx**

Inconsistencies:

The following selection-based links have descriptions that differ from their corresponding selections in the requirements documents. If this reflects a change in the requirements document, click **Update** to replace the current description in the selection-based link with the text from the requirements document (the external description).

Block	Current description	External description	
slvnvdemo_fuelsys_officereq/Test inputs	Normal mode of operation	The simulation is run with a throttle input that ramps from 10 to 20 degrees over a period of two seconds, then back to 10 degrees over the next two seconds. This cycle repeats continuously while the engine is held at a constant speed.	Update
slvnvdemo_fuelsys_officereq/fuel rate controller/Sensor correction and Fault Redundancy/MAP Estimate	Manifold pressure failure	Manifold pressure failure mode	Update

The first message indicated that the model contains a link to a bookmark named **Simulink_requirement_item_7** in the requirements document that does not exist.

In addition, this check identified the following mismatching text between the requirements blocks and the requirements document:

- The **Description** field in the Test inputs Signal Builder block link is **Normal mode of operation**. The requirement text is **The simulation is run with a throttle input that ramps from 10 to 20 degrees over a period of two seconds, then back to 10 degrees over the next**

two seconds. **This cycle repeats continuously while the engine is held at a constant speed.**

- The **Description** field in the MAP Estimate block link is **Manifold pressure failure**. The requirement text in `slvnvdemo_FuelSys_DesignDescription.docx` is **Manifold pressure failure mode**.

2 Get more information about this link:

  a To navigate to a block, under **Block**, click the hyperlink.

  b To open the Requirements dialog box for this link, under **Current Description**, click the hyperlink.

3 Fix this problem in one of two ways:

- In the Model Advisor, click **Update**. This action automatically updates the **Description** field for that link so that it matches the requirement.

- In the Requirements dialog box, manually edit the link from the block so that the **Description** field matches the selected requirements text.

4 In the Model Advisor, select the **Requirement consistency checking** category of checks.

5 Click **Run Selected Checks** again, and verify that the warning no longer occurs.

Checking and Fixing ActiveX-Based Links in Requirements Documents

- "When to Check Links in a Requirements Document" on page 5-10
- "How the rmi Function Checks a Requirements Document" on page 5-10
- "Checking Links in a Requirements Document" on page 5-11
- "When Multiple Objects Have Links to the Same Requirement" on page 5-12
- "Fixing Requirements Links in a Requirements Document" on page 5-13

When to Check Links in a Requirements Document

When you enable **Modify destination for bi-directional linking** and create a link between a requirement and a Simulink model object, the RMI software inserts a navigation control into your requirements document. These links may become invalid if your model changes.

To check these links, the `'checkDoc'` option of the `rmi` function reviews a requirements document to verify that all the navigation controls represent valid links to model objects. The `checkDoc` command can check the following types of requirements documents:

- Microsoft Word
- Microsoft Excel
- IBM Rational DOORS

The `rmi` function only checks requirements documents that contain navigation controls; to check links in your Simulink model, see "Checking and Fixing Requirements Links in a Simulink Model" on page 5-2.

Note For more information about inserting navigation controls in requirements documents, see:

- "Inserting Navigation Objects in Microsoft Office Requirements Documents" on page 7-12
- "Inserting Navigation Objects into DOORS Requirements" on page 7-5

How the rmi Function Checks a Requirements Document

`rmi` performs the following actions:

- Locates all links to Simulink objects in the specified requirements document.
- Checks each link to verify that the target object is present in a Simulink model. If the target object is present, `rmi` checks that the link label matches the target object.

- Modifies the navigation controls in the requirements document to identify any detected problems. This allows you to see invalid links at a glance:

  - Valid link:

  - Invalid link:

Checking Links in a Requirements Document

To check the links in a requirements document:

1 At the MATLAB command prompt, enter

```
rmi('checkdoc', docName)
```

docName is a string that represents one of the following:

- Module ID for a DOORS requirements document

- Full path name for a Microsoft Word requirements document

- Full pat name for a Microsoft Excel requirements document

The rmi function creates and displays an HTML report that lists all requirements links in the document.

The report highlights invalid links in red. For each invalid link, the report includes brief details about the problem and a hyperlink to the invalid link in the requirements document. The report groups together links that have the same problem.

2 Double-click the hyperlink under **Document content** to open the requirements document at the invalid link.

The navigation controls for the invalid link has a different appearance than the navigation controls for the valid links.

3 When there are invalid links in your requirements document, you have the following options:

If you want to...	Do the following...
Fix the invalid links	Follow the instructions in "Fixing Requirements Links in a Requirements Document" on page 5-13.
Keep the changes to the navigation controls without fixing the invalid links	Save the requirements document.
Ignore the invalid links	Close the requirements document without saving it.

When Multiple Objects Have Links to the Same Requirement

When you link multiple objects to the same requirement, as described in "Linking Multiple Model Objects to a Requirements Document" on page 2-21, only one navigation object is inserted into the requirements document. When you double-click that navigation object, all of the linked model objects are highlighted.

If you check the requirements document using the 'checkdoc' option of the rmi function and the check detects a navigation object that points to multiple objects, the check stops and displays the following dialog box.
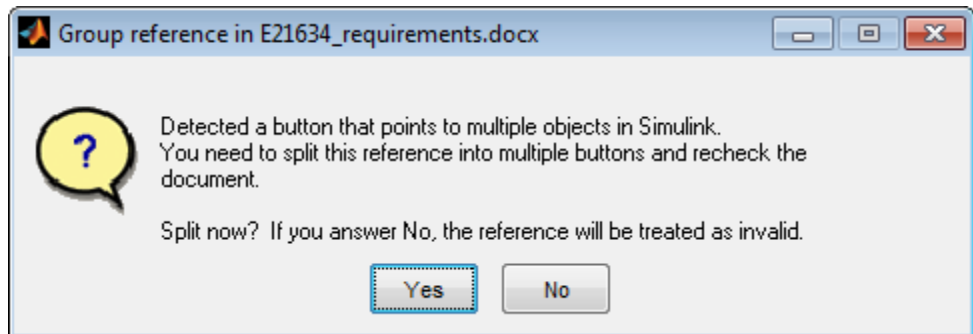
You have two options:

- If you click **Yes**, or you close this dialog box, the RMI creates additional navigation objects, one for each model object that links to that requirement.

The document check continues, but the RMI does not recheck that navigation; the report only shows one link for that requirement. To rerun the check so that all requirements are checked, at the top of the report, click **Refresh**.

- If you click **No**, the document check continues, and the report identifies that navigation object as a broken link.

Fixing Requirements Links in a Requirements Document

Using the report that the `rmi` function creates, you may be able to fix the invalid links in your requirements document.

In the following example, `rmi` cannot locate the model specified in two links.

To fix invalid links:

1 In the report, under **Document content**, click the hyperlink associated with the invalid requirement link.

The requirements document opens with the requirement text highlighted.

2 In the requirements document, depending on the document format, take these steps:

- In DOORS:

  a Select the navigation control for an invalid link.

  b Select **MATLAB > Select item**.

- In Microsoft Word, double-click the navigation control.

A dialog box opens that allows you to fix, reset, or ignore all the invalid links with a given problem.

3 Click one of the following options.

To...	Click...
Navigate to and select a new target model or new target objects for these broken links.	**Fix all**
Reset the navigation controls for these invalid links to their original state, the state before you checked the requirements document.	**Reset all**
Make no changes to the requirements document. Any modifications rmi made to the navigation controls remain in the requirements document.	**Cancel**

4 Save the requirements document to preserve the changes made by the rmi function.

Resolving the Document Path

In this section...
"Relative (Partial) Path Example" on page 5-16
"Relative (No) Path Example" on page 5-16
"Absolute Path Example" on page 5-16

When you create a requirements link, the RMI stores the location of the requirements document with the link. If you use selection-based linking or browse to select a requirements document, the RMI stores the document location as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab. The available settings are:

- Absolute path

- Path relative to current folder

- Path relative to model folder

- Filename only (on MATLAB path)

Note For detailed information about these settings, see "Resolving the Document Path" on page 5-15.

You can also manually enter an absolute or relative path for the document location. A relative path can be a partial path or no path at all, but you must specify the file name of the requirements document. 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 folder.

2 When there is no path specification and the document is not in the current folder, the software uses the MATLAB search path to locate the file.

3 If the RMI cannot locate the document relative to the current folder or the MATLAB search path, the RMI resolves the path relative to the model file folder.

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

Relative (Partial) Path Example

Current MATLAB folder	`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`

Relative (No) Path Example

Current MATLAB folder	`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 Example

Current MATLAB folder	`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`

Deleting Requirement Links from Simulink Objects

In this section...

Deleting a Single Link from a Simulink Object

If you have an obsolete link to a requirement, delete it from the model object.

To delete a single link to a requirement from a Simulink model object:

1 Right-click a model object and select **Requirements > Edit/Add Links**.

2 In the top-most pane of the Requirements dialog box, select the link that you want to delete.

3 Click **Delete**.

4 Click **Apply** or **OK** to complete the deletion.

Deleting All Links from a Simulink Object

To delete all links to requirements from a Simulink model object:

1 Right-click the model object and select **Requirements > Delete All Links**

2 Click **OK** to confirm the deletion.

This action deletes all requirements at the top level of the object. For example, if you delete requirements for a subsystem, this action does not delete any requirements for objects inside the subsystem; it only deletes requirements for the subsystem itself. To delete requirements for child objects inside a subsystem, Model block, or Stateflow chart, you must navigate to each child object and perform these steps for each object from which you want to delete requirements.

Deleting All Links from Multiple Simulink Objects

To delete all requirements links from a group of Simulink model objects in the same model diagram or Stateflow chart:

1 Select the model objects whose requirements links you want to delete.

2 Right-click one of the objects and select **Requirements > Delete All**.

3 Click **OK** to confirm the deletion.

This action deletes all requirements at the top level of each object. It does not delete requirements for any child objects inside subsystems, Model blocks, or Stateflow charts.

Understanding Requirements in Library Blocks and Reference Blocks

In this section...
"Introduction to Library Blocks and Reference Blocks" on page 5-19
"Library Blocks and Requirements" on page 5-19
"Copying Library Blocks with Requirements" on page 5-20
"Managing Requirements on Reference Blocks" on page 5-20
"Managing Requirements Inside Reference Blocks" on page 5-21
"Links from Requirements to Library Blocks" on page 5-24

Introduction to Library Blocks and Reference Blocks

Simulink allows you to create your own block libraries. If you create a block library, you can reuse the functionality of a block, subsystem, or Stateflow atomic subchart in multiple models.

When you copy a library block to a Simulink model, the new block is called a *reference block*. You can create several *instances* of this library block in one or more models.

The reference block is linked to the library block using a *library link*. If you change a library block, any reference block that is linked to the library block is updated with those changes when you open or update the model that contains the reference block.

Note For more information about reference blocks and library links, see "Working with Block Libraries" in the Simulink documentation.

Library Blocks and Requirements

Library blocks themselves can have links to requirements. In addition, if a library block is a subsystem or atomic subchart, the objects inside the library blocks can have library links. You use the Requirements Management

Interface (RMI) to create and manage requirements links in libraries and in models.

The following sections describe how to manage requirements links on and inside library blocks and reference blocks.

Copying Library Blocks with Requirements

When you copy a library block to a model, you can highlight, view and navigate requirements links on the library block and on objects inside the library block. However, those links are not associated with that model. The links are stored with the library, not with the model

You cannot add, modify, or delete requirements links on the library block from the context of the reference block. If you disable the link from the reference block to the library block, you can modify requirements on objects that are inside library blocks just as you can for other block attributes when a library link has been disabled.

Managing Requirements on Reference Blocks

You use the RMI to manage requirements links on a reference block just like any other model object. You can view and navigate both local and library requirements on a reference block.

For example, in the following graphic, right-clicking the reference block shows that the reference block has locally created requirements links and requirements links on the library block:

- Locally created requirements links — Can be modified or deleted without changing the library block:

  - **Manifold absolute pressure sensor**

  - **Mass airflow estimation**

- Requirements links on the library block — Cannot be modified or deleted from the context of the reference block:

  - **Speed sensor**

  - **Throttle sensor**

- **Oxygen sensor**

Requirements	▶	1. "Manifold absolute pressure sensor"		
Design Verifier	▶	2. "Mass airflow estimation"		
Code Generation	▶	Library Block Requirements	▶	1. "Speed sensor"
HDL Code Generation	▶			2. "Throttle sensor"
PLC Code Generation	▶	Add link to Word selection		3. "Oxygen sensor"
		Add link to active Excel cell		
Fixed-Point Tool...		Add link to current DOORS object		
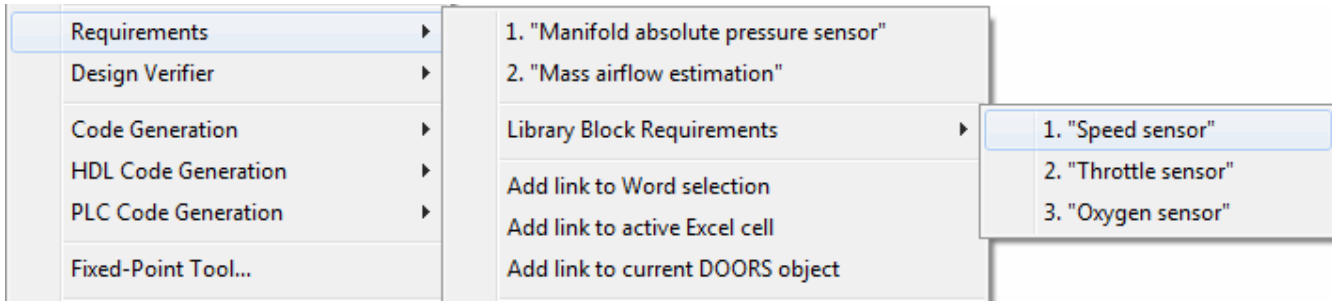
Managing Requirements Inside Reference Blocks

If your library block is a subsystem or a Stateflow atomic subchart, you can create requirements links on objects *inside* the subsystem or subchart. If you disable the link from the reference block to the library, you can add, modify, or delete requirements links on objects inside a reference block. Once you have disabled the link, the RMI treats those links as locally created links.

After you make changes to requirements links on objects inside a reference block, you can resolve the link so that those changes are pushed to the library block. The next time you create an instance of that library block, the changes you made are copied to the new instance of the library block.

The workflow for creating a requirement link on an object inside a reference block is:

1 Within a library you have a subsystem S1. Drag S1 to a model, creating a new subsystem. This subsystem is the reference block.
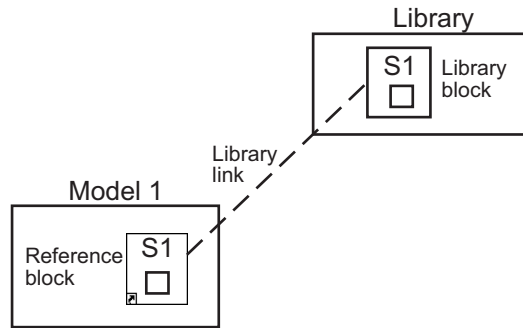
2 Disable the library link between the reference block and the library block. To do this, select the reference block and select **Edit > Link Options > Disable Link**.

3 Create a link from the object inside the reference block to the requirements document.
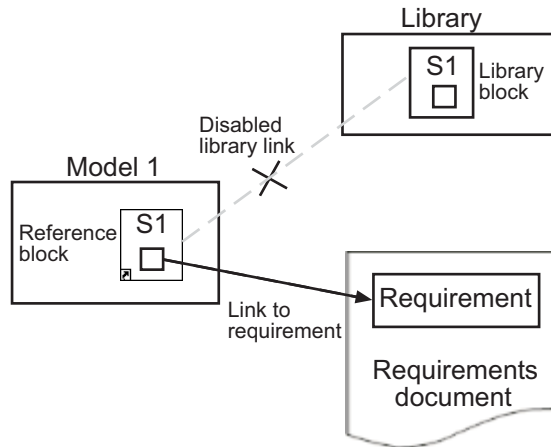
Note When linking to a requirement from inside a reference block, you can create links only in one direction: from the model to the requirements document. The RMI does not support inserting navigation objects into requirements documents for objects inside reference blocks.

4 Resolve the library link between the reference block and the library block:

   a Select the reference block.

   b Select **Edit > Link Options > Resolve Link**.

   c In the **Action** column, click **Push**.

   d Click **OK** to resolve the link to the library block and push the newly added requirement to the object inside the library block.

   When you resolve the library link between the library block and the subsystem, Simulink pushes the new requirement link to the library block S1. The following graphic shows the new link from inside the library block S1 to the requirement.

Library

S1   Library block

Restored library link

Link to requirement

Model 1

Reference block   S1

Link to requirement

Requirement

Requirements document
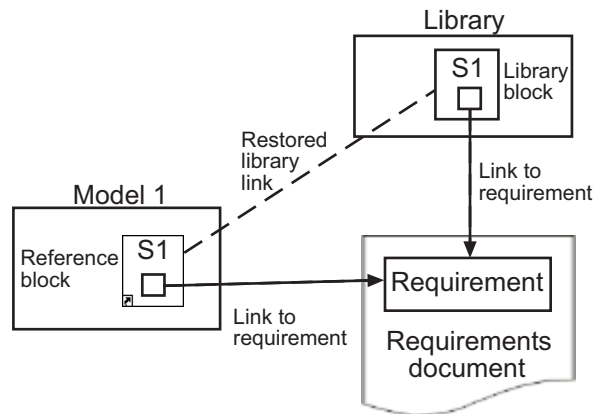
Note If you see a message that the library is locked, you must unlock the library before you can push the changes to the library block.

5 If you reuse library block S1, which now has an object with a requirement link, in another model, the new subsystem contains an object that links to that requirement.
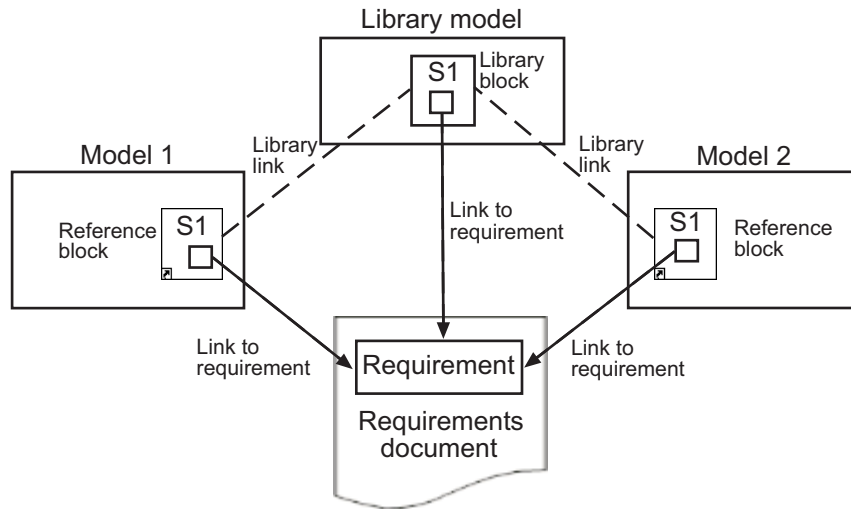
Links from Requirements to Library Blocks

If you have a requirement that links to a library block and you drag that library block to a model, the requirement does not link to the reference block; the requirement links *only* to the library block.

For example, consider the situation where you have established linking between a library block (B1 in the following graphic) and a requirement in both directions.
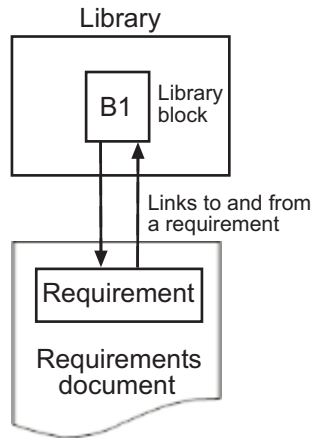
When you use library block B1 in a model, you can navigate from the reference block to the requirement. However, the link from the requirement still points only to library block B1, not to the reference block.
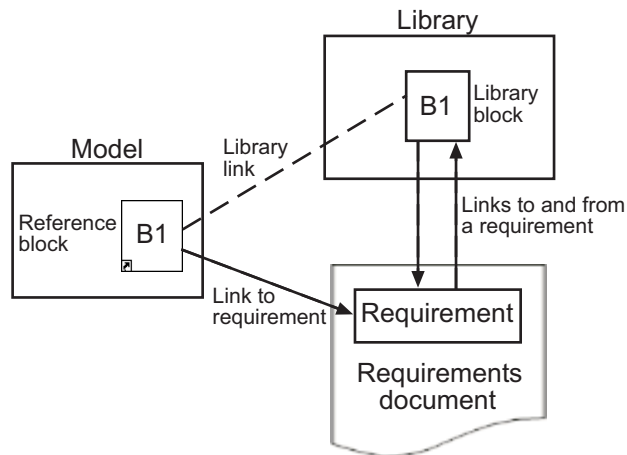
As discussed in the previous section, you can create requirements links on objects inside instances of library block after disabling library links. However, the RMI prohibits you from creating a link from the requirements document to such an object because that link would become invalid when you restored the library link.

6

IBM Rational DOORS Surrogate Module Synchronization

What Is Synchronization?

Synchronization is a user-initiated process that creates or updates a DOORS surrogate module. A *surrogate module* is a DOORS formal module that is a representation of a Simulink model hierarchy.

When you synchronize a model for the first time, the DOORS software creates a surrogate module. The surrogate module contains a representation of the model, depending on your synchronization settings. (To learn how to customize the links and level of detail in the synchronization, see "Customizing the Synchronization" on page 6-8.)

If you create or remove model objects or links, keep your surrogate module up to date by resynchronizing. The updated surrogate module reflects any changes in the requirements links since the previous synchronization.

Note The RMI and DOORS software both use the term *object*. In the RMI, and in this document, the term *object* refers to a Simulink model or block, or to a Stateflow chart or its contents.

In the DOORS software, *object* refers to numbered elements in modules. The DOORS software assigns each of these objects a unique object ID. In this document, these objects are referred to as *DOORS objects*.

You use standard DOORS capabilities to navigate between the Simulink objects in the surrogate module and requirements in other formal modules. The surrogate module facilitates navigation between the Simulink model object and the requirements, as the following diagram illustrates.

Objects in a Simulink Model
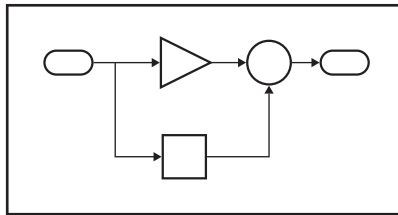
DOORS Surrogate Module

Object ID	Block Name	
200	1	Model
202	1.1	Subsystem
203	1.1.1	Block
204	1.1.2	Block
205	1.1.3	Block
206	1.2	Subsystem
207	1.2.1	Block
208	1.3	Block

A surrogate module is a representation of a Simulink model hierarchy.

DOORS Formal Module(s) with Requirements

Object ID		Requirement	
D1		1	**Requirement Name**
D2	◄	1.1	Requirement text ...
			...
			...
D3		1.2	Requirement text ...

Enter requirements in the DOORS formal module and link them to objects in the DOORS surrogate module, so you can navigate from requirements to Simulink objects.

Advantages of Synchronizing Your Model with a Surrogate Module

Synchronizing your Simulink model with a surrogate module offers the following advantages:

- You can navigate from a requirement to a Simulink object without modifying the requirements modules.

- You avoid cluttering your requirements modules with inserted navigation objects.

- The DOORS database contains complete information about requirements links. You can review requirements links and verify traceability, even if the Simulink software is not running.

- You can use DOORS reporting features to analyze requirements coverage.

- You can separate the requirements tracking work from the Simulink model developers' work, as follows:

  - Systems engineers can establish requirements links to models without using the Simulink software.

  - Model developers can capture the requirements information using synchronization and store it with the model.

- You can resynchronize a model with a new surrogate module, updating any model changes or specifying different synchronization options.

Synchronizing a Simulink Model to Create a Surrogate Module

The first time that you synchronize your model with the DOORS software, the DOORS software creates a surrogate module.

In this tutorial, you synchronize the sf_car model with the DOORS software.

Note Before you begin, make sure you know how to create links from a Simulink model object to a requirement in a DOORS database. For a tutorial on creating links to DOORS requirements, see "Linking to Requirements in IBM Rational DOORS Databases" on page 2-30.

---

1 To create a surrogate module, start the DOORS software and open a project. If the DOORS software is not already running, start the DOORS software and open a project.

2 Open the sf_car model.

3 Rename the model to sf_car_doors, and save the model in a writable folder.

4 Create links to a DOORS formal module from two objects in sf_car_doors:

- The transmission subsystem

- The engine torque block inside the Engine subsystem

5 Save the changes to the model.

6 In the Model Editor, select **Tools > Requirements > Synchronize with DOORS**.

The DOORS synchronization settings dialog box opens.

7 For this tutorial, accept the default synchronization options.

The default option under **Extra mapping additionally to objects with links**, None, creates objects in the surrogate module only for the model and any model objects with links to DOORS requirements.

> **Note** For more information about the synchronization options, see "Customizing the Synchronization" on page 6-8.

8 Click **Synchronize** to create and open a surrogate module for all DOORS requirements that have links to objects in the sf_car_doors model.

After synchronization with the None option, the surrogate module, a formal module named sf_car_doors, contains:

- A top-level object for the model (sf_car_doors)

- Objects that represent model objects with links to DOORS requirements (transmission, engine torque), and their parent objects (Engine).
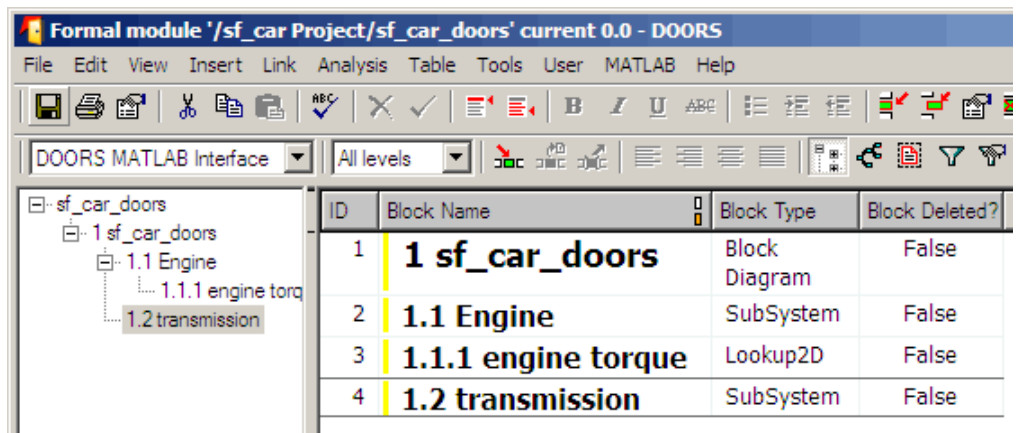
9 Save the surrogate module and the model.

Creating Links Between the Surrogate Module and Formal Module in a DOORS Database During Synchronization

The surrogate module is the interface between the DOORS formal module that contains your requirements and the Simulink model. To establish links between the surrogate module and the requirements module, copy the link information from the model to the surrogate module:

1 Open the sf_car_doors model.

2 In the Model Editor, select **Tools > Requirements > Synchronize with DOORS**.

3 In the DOORS synchronization settings dialog box, select two options:

- **Update links during synchronization**
- **from Simulink to DOORS**.

4 Click **Synchronize**.

The RMI creates links from the DOORS surrogate module to the formal module. These links correspond to links from the Simulink model to the formal module. In this example, the DOORS software copies the links from the engine torque block and transmission subsystems to the formal module, as indicated by the red triangles.

Customizing the Synchronization

DOORS Synchronization Settings

When you synchronize your Simulink model with a DOORS database, you can:

- Customize the level of detail for your surrogate module.

- Update links in the surrogate module or in the model to verify the consistency of requirements links among the model, and the surrogate and formal modules.

The DOORS synchronization settings dialog box provides the following options during synchronization.

DOORS Settings Option	Description
DOORS surrogate module path and name	Specifies a unique DOORS path to a new or an existing surrogate module.
	For information about how the RMI resolves the path to the requirements document, see "Resolving the Document Path" on page 5-15.
Extra mapping additionally to objects with links	Determines the completeness of the Simulink model representation in the DOORS surrogate module. None specifies synchronizing only those Simulink objects that have linked requirements, and their parent objects. For more information about these synchronization options, see "Customizing the Level of Detail in Synchronization" on page 6-11.

DOORS Settings Option	Description
Update links during synchronization	Specifies updating any unmatched links the RMI encounters during synchronization, as designated in the **Copy unmatched links** and **Delete unmatched links** options.
Copy unmatched links	During synchronization, selecting the following options has the following results: • **from Simulink to DOORS**: For links between the model and the formal module, the RMI creates matching links between the DOORS surrogate and formal modules. • **from DOORS to Simulink**: For links between the DOORS surrogate and formal modules, the RMI creates matching links between the model and the DOORS modules.
Delete unmatched links	During synchronization, selecting the following options has the following results: • **Remove unmatched in DOORS**: For links between the formal and surrogate modules, when there is not a corresponding link between the model and the DOORS modules, the RMI deletes the link in DOORS. This option is available only if you select the **from Simulink to DOORS** option. • **Remove unmatched in Simulink**: For links between the model and the DOORS modules, when there is not a corresponding link between the formal and surrogate modules, the RMI deletes the link from the model. This option is available only if you select the **from DOORS to Simulink** option.

DOORS Settings Option	Description
Save DOORS surrogate module	After the synchronization, saves all changes to the surrogate module and updates the version of the surrogate module in the DOORS database.
Save Simulink model (recommended)	After the synchronization, saves all changes to the model. If you use a version control system, selecting this option changes the version of the model.

Resynchronizing a Model with a Different Surrogate Module

You can synchronize the same Simulink model with a new DOORS surrogate module. For example, you might want the surrogate module to contain only objects that have requirements to DOORS, rather than all objects in the model. In this case, you can change the synchronization options to reduce the level of detail in the surrogate module:

1 In the DOORS synchronization settings dialog box, change the **DOORS surrogate module path and name** to the path and name of the new surrogate module in the DOORS database.

2 Specify a module with either a relative path (starting with ./) or a full path (starting with /).

The software appends relative paths to the current DOORS project. Absolute paths must specify a project and a module name.

When you synchronize a model, the RMI automatically updates the **DOORS surrogate module path and name** with the actual full path. The RMI saves the unique module ID with the module.

3 If you select a new module path or if you have renamed the surrogate module, and you click **Synchronize**, the Requirements: Surrogate Module Mismatch dialog box opens.
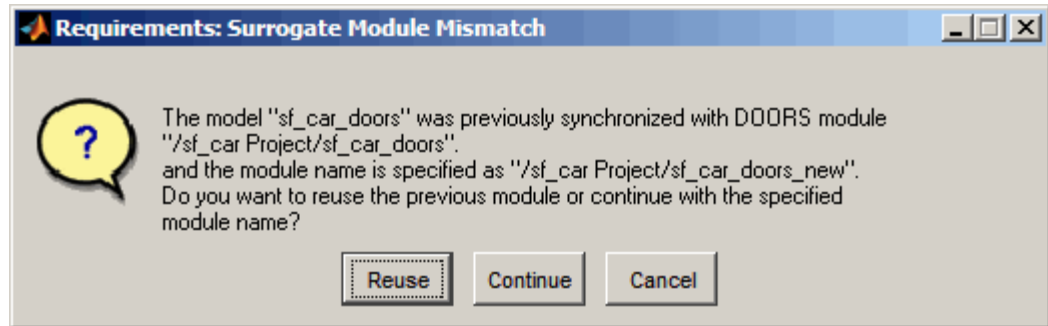
4 Click **Continue** to create a new surrogate module with the new path or name.

Customizing the Level of Detail in Synchronization

You can customize the level of detail in a surrogate module so that the module reflects the full or partial Simulink model hierarchy.

In "Synchronizing a Simulink Model to Create a Surrogate Module" on page 6-5, you synchronized the model with the **Extra mapping additionally to objects with links** option set to None. As a result, the surrogate module contains only Simulink objects that have requirement links, and their parent objects. Additional synchronization options, described in this section, can increase the level of surrogate detail. Increasing the level of surrogate detail can slow down synchronization.

The **Extra mapping additionally to objects with links** option can have one of the following values. Each subsequent option adds additional Simulink objects to the surrogate module. You choose None to minimize the surrogate size or Complete to create a full representation of your model. The Complete option adds all Simulink objects to the surrogate module, creating a one-to-one mapping of the Simulink model in the surrogate module. The intermediate options provide more levels of detail.

Drop-Down List Option	Description
None (Recommended for better performance)	Maps only Simulink objects that have requirements links and their parent objects to the surrogate module.
Minimal - Non-empty unmasked subsystems and Stateflow charts	Adds all nonempty Stateflow charts and unmasked Simulink subsystems to the surrogate module.
Moderate - Unmasked subsystems, Stateflow charts, and superstates	Adds Stateflow superstates to the surrogate module.
Average - Nontrivial Simulink blocks, Stateflow charts and states	Adds all Stateflow charts and states and Simulink blocks, except for trivial blocks such as ports, bus objects, and data-type converters, to the surrogate module.
Extensive - All unmasked blocks, subsystems, states and transitions	Adds all unmasked blocks, subsystems, states, and transitions to the surrogate module.
Complete - All blocks, subsystems, states and transitions	Copies *all* blocks, subsystems, states, and transitions to the surrogate module.

Resynchronizing to Include All Simulink Objects

This tutorial shows how you can include *all* Simulink objects in the DOORS surrogate module. Before you start these steps, make sure you have completed the tutorials "Synchronizing a Simulink Model to Create a Surrogate Module" on page 6-5 and "Creating Links Between the Surrogate Module and Formal Module in a DOORS Database During Synchronization" on page 6-7.

1 Open the sf_car_doors model that you synchronized in "Synchronizing a Simulink Model to Create a Surrogate Module" on page 6-5 and again in "Creating Links Between the Surrogate Module and Formal Module in a DOORS Database During Synchronization" on page 6-7.

2 In the Model Editor, select **Tools > Requirements > Synchronize with DOORS**.

The DOORS synchronization settings dialog box opens.

3 Resynchronize with the same surrogate module, making sure that the **DOORS surrogate module path and name** specifies the surrogate module path and name that you used in "Synchronizing a Simulink Model to Create a Surrogate Module" on page 6-5.

For information about how the RMI resolves the path to the requirements document, see "Resolving the Document Path" on page 5-15.

4 Update the surrogate module to include *all* objects in your model. To do this, under **Extra mapping additionally to objects with links**, from the drop-down list, select `Complete - All blocks, subsystems, states and transitions`.

5 Click **Synchronize**.

After synchronization, the DOORS surrogate module for the `sf_car_doors` model opens with the updates. All Simulink objects and all Stateflow objects in the `sf_car_doors` model are now mapped in the surrogate module.

ID	Block Name	Block Type	Block Deleted?
1	**1 sf_car_doors**	Block Diagram	False
2	**1.1 Engine**	SubSystem	False
5	**1.1.1 Ti**	Inport	False
6	**1.1.2 throttle**	Inport	False
7	**1.1.3 Integrator**	Integrator	False
8	**1.1.4 Sum**	Sum	False
3	**1.1.5 engine torque** ▶	Lookup2D	False
9	**1.1.6 engine + impeller inertia**	Gain	False
10	**1.1.7 Ne**	Outport	False
11	**1.2 Mux**	Mux	False
12	**1.3 User Inputs:Passing Maneuver**	Signal Group	False
13	**1.4 User Inputs:Gradual Acceleration**	Signal Group	False
14	**1.5 User Inputs:Hard**	Signal Group	False

6 Scroll through the surrogate module. Notice that the objects with requirements (the engine torque block and transmission subsystem) retain their links to the DOORS formal module, as indicated by the red triangles.

7 Save the surrogate module.

Detailed Information About The Surrogate Module You Created

Notice the following information about the surrogate module that you created in "Resynchronizing to Include All Simulink Objects" on page 6-12:

- The name of the surrogate module is sf_car_doors, as you specified in the DOORS synchronization settings dialog box.

- DOORS object headers are the names of the corresponding Simulink objects.

- The **Block Type** column identifies each object as a particular block type or a subsystem.

- If you delete a previously synchronized object from your Simulink model and then resynchronize, the **Block Deleted** column reads **true**. Otherwise, it reads **false**.

  These objects are not deleted from the surrogate module. The DOORS software retains these surrogate module objects so that the RMI can recover these links if you later restore the model object.

- Each Simulink object has a unique ID in the surrogate module. For example, the ID for the surrogate module object associated with the Mux block in the preceding figure is 11.

- Before the complete synchronization, the surrogate module contained the transmission subsystem, with an ID of 3. After the complete synchronization, the transmission object retains its ID (3), but is listed farther down in the surrogate module. This order reflects the model hierarchy. The transmission object in the surrogate module retains the red arrow that indicates that it links to a DOORS formal module object.

Resychronizing to Reflect Model Changes

If you change your model after synchronization, the RMI does not display a warning message. If you want the surrogate module to reflect changes to the Simulink model, resynchronize your model.

In this tutorial, you add a new block to the sf_car_doors model, and later delete it, resynchronizing after each step:

1 In the sf_car_doors model, make a copy of the vehicle mph (yellow) & throttle % Scope block and paste it into the model. The name of the new Scope block is vehicle mph (yellow) & throttle %1.

2 Select **Tools > Requirements > Synchronize with DOORS**.

3 In the DOORS settings dialog box, leave the **Extra mapping additionally to objects with links** option set to Complete - All blocks, subsystems, states, and transitions. Click **Synchronize**.

After the synchronization, the surrogate module includes the new block.

89	**1.10.6 Ti**	Outport	False
90	**1.10.7 Tout**	Outport	False
91	**1.11 vehicle mph (yellow) & throttle %**	Scope	False
92	**1.12 vehicle mph (yellow) & throttle %1**	Scope	True

4 In the sf_car_doors model, delete the newly added Scope block and resynchronize.

The block that you delete appears at the bottom of the list of objects in the surrogate module. Its entry in the **Block Deleted** column reads True.

89	**1.10.6 Ti**	Outport	False
90	**1.10.7 Tout**	Outport	False
91	**1.11 vehicle mph (yellow) & throttle %**	Scope	False
92	**1.12 vehicle mph (yellow) & throttle %1**	Scope	False

5 Delete the copied object (vehicle mph (yellow) & throttle %1 and resynchronize the model.

6 Save the surrogate module.

7 Save the sf_car_doors model.

Navigating with the Surrogate Module

In this section...
"Navigating Between Requirements and the Surrogate Module in the DOORS Database" on page 6-18
"Navigation Between DOORS Requirements and the Simulink Module via the Surrogate Module" on page 6-19

Navigating Between Requirements and the Surrogate Module in the DOORS Database

The surrogate module and the requirements in the formal module are both in the DOORS database. When you synchronize your model, the DOORS software creates links between the surrogate module objects and the requirements in the DOORS database.

Navigating between the requirements and the surrogate module allows you to review the requirements that have links to the model without starting the Simulink software.

To navigate from the surrogate module transmission object to the requirement in the formal module:

1 In the surrogate module object for the transmission subsystem, right-click the right-facing red arrow.

65	1.9.3.7 up_th	Outport	False	
4	**1.10 transmission**	/sf_car Project/sf_car Requirements ▶		1: Transmission Requirements: Shoul
66	1.10.1 Ne	Inport	False	

2 Select the requirement name.

The formal module opens, at the `Transmission Requirements` object.

To navigate from the requirement in the formal module to the surrogate module:

1 In the Transmission Requirements object in the formal module, right-click the left-facing orange arrow.
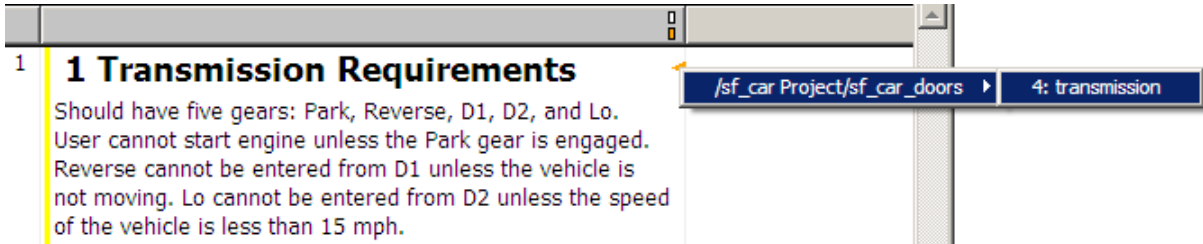
2 Select the object name.

The surrogate module for sf_car_doors opens, at the object associated with the transmission subsystem.

Navigation Between DOORS Requirements and the Simulink Module via the Surrogate Module

You can create links that allow you to navigate from Simulink objects to DOORS requirements and from DOORS requirements to the model. If you synchronize your model, the surrogate module serves as an intermediary for the navigation in both directions. The surrogate module allows you to navigate in both directions even if you remove the direct link from the model object to the DOORS formal module.

Navigating from a Simulink Object to a Requirement via the Surrogate Module

To navigate from the transmission subsystem in the sf_car_doors model to a requirement in the DOORS formal module:

1 In the sf_car_doors model, right-click the transmission subsystem and select **Requirements > 1. "DOORS Surrogate Item"**. (The direct link to the DOORS formal module is also available.)

The surrogate module opens, at the object associated with the transmission subsystem.

2 To display the individual requirement, in the surrogate module, right-click the right-facing red arrow and select the requirement.

The formal module opens, at `Transmission Requirements`.

Navigating from a Requirement to the Model via the Surrogate Module

To navigate from the `Transmission Requirements` requirement in the formal module to the transmission subsystem in the `sf_car_doors` model:

1 In the formal module, in the `Transmission Requirements` object, right-click the left-facing orange arrow.

2 Select the path to the linked surrogate object: **/sf_car Project/sf_car_doors > 4. transmission**.

The surrogate module opens, at the transmission object.

3 In the surrogate module, select **MATLAB > Select item**.

The linked object is highlighted in `sf_car_doors`.

Navigation from Requirements Documents

- "IBM Rational DOORS" on page 7-2
- "Microsoft Office" on page 7-11

IBM Rational DOORS

In this section...

Configuring the Requirements Management Interface for DOORS Software

- "Before You Begin" on page 7-2
- "Manually Installing Additional Files for DOORS Software" on page 7-2

Before You Begin

If you plan to use DOORS software with the RMI, make sure to install additional files to establish communication between the DOORS application and the Simulink software. Follow the instructions in "Configuring the Requirements Management Interface (RMI)" on page 2-13.

Manually Installing Additional Files for DOORS Software

The setup script automatically copies all the required DOORS files to the installation folders. However, the script may fail because of file permissions in your DOORS installation. If the script fails, change the file permissions on the DOORS installation folders and rerun the script.

Alternative, you can install the additional files into the folders specified manually, as described in the following steps:

1 If the DOORS software is running, close the application.

2 Copy the following files from *matlabroot*\toolbox\slvnv\reqmgt to the *<doors_install_dir>*\lib\dxl\addins folder.

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

If you have not modified the files, replace any existing versions of the files; otherwise, merge the contents of both files into a single file.

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

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

Replace any existing versions of these files.

4 Open the *<doors_install_dir>*\lib\dxl\startup.dxl file. In the user-defined files section, add the following include statement:

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

If you upgrade from Version 7.1 to a later version of the DOORS software, perform these additional steps:

a In your DOORS installation folder, navigate to the ...\lib\dxl\startupFiles subfolder.

b In a text editor, open the copiedFromDoors7.dxl file.

c Add // before this line to comment it out:

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

d Save and close the file.

5 Start the DOORS and MATLAB software.

6 Run the setup script:

```
rmi setup
```

Enabling Linking Between DOORS Databases and Simulink Models

By default, the RMI does not insert navigation objects into requirements documents. If you want the RMI to insert navigation objects when you create a link from any model object to a requirement, you must configure the RMI to do this.

Enable the RMI to insert navigation objects into the DOORS database:

1 Open any model. For example, open the Simulink model:

```
sldemo_fuelsys
```

> **Note** You can modify requirements settings in the Requirements Settings dialog box, available only from the Model Editor. Even though you have a model open, any settings you change persist for all models you open subsequently. Changes you make are global and not specific to open models.

2 Select **Tools > Requirements > Settings**.

The Requirements Settings dialog box opens.

3 Click the **Selection Linking** tab.

4 Enable **Modify destination for bi-directional linking**.

When you select this option, every time you create a selection-based link from a model object to a requirement, the RMI inserts navigation objects at the designated location. Using this option requires write access to the requirements document.

5 Set the **Model file reference** option to none (on MATLAB path).

For this exercise, you save a copy of the demo model on the MATLAB path.

If you are adding requirements to a model that is not on the MATLAB path, select absolute, to indicate an absolute path to the model.

6 In the **Apply this user tag to new links** field, enter one or more user tags to apply to the links that you create.

For more information about user tags, see "Understanding Requirements Filtering" on page 4-25.

7 Click **Close** to close the Requirements Settings dialog box.

Keep the sldemo_fuelsys model open.

Inserting Navigation Objects into DOORS Requirements

When you enable **Modify destination for bi-directional linking**, the RMI inserts a navigation object into both the model and the requirement. For this tutorial, you need a formal module that contains requirements. The examples in the tutorial show a demo model used for the purposes of illustration.

1 Rename the sldemo_fuelsys model and save it in a writable folder on the MATLAB path.

2 Start the DOORS software and open a formal module that contains requirements.

3 Select the requirement that you want to link to by left-clicking that requirement in the DOORS database.

4 In the sldemo_fuelsys model, select an object in the model.

This example creates a requirement from the fuel_rate_control subsystem.

5 Right-click the model object and select **Requirements > Add link to current DOORS object**.

The RMI creates the link for the fuel_rate_control subsystem. It also inserts a new DOORS object into the formal module—a Simulink reference object ( ) that enables you to navigate from the requirement to the model.

6 Close the model.

Note When you navigate to a DOORS requirement, the DOORS module opens in read only mode. If you want to modify the DOORS module, open the module using DOORS software.

Inserting Navigation Objects to Multiple Simulink Objects

If you have several model objects that correspond to a requirement, you can link all the model objects to that requirement with one navigation object. This eliminates the need to insert multiple navigation objects for a single requirement. The model objects must be available in the same model diagram or Stateflow chart.

The workflow for linking multiple objects to a DOORS requirement is as follows:

1 Make sure that you have enabled **Modify destination for bi-directional linking**.

2 Select the DOORS requirement to link to.

3 Select the model objects that need to link to that requirement.

4 Right-click one of the objects and select **Requirements > Add link to current DOORS object**.

A single navigation object is inserted at the selected requirement.

5 Double-click the navigation object in DOORS to highlight the model objects that are linked to that requirement.

Customizing Navigation Objects

If the Requirements Management Interface (RMI) is configured to modify destination for bi-directional linking, the RMI inserts a navigation object into your requirements document. This object looks like the icon for the Simulink software:

Note In IBMRationalDOORS requirements documents, clicking a navigation objects does not navigate back to your Simulink model; you must select **MATLAB > Select object** to find the model object that contains the requirements link.

In Microsoft Office requirements documents, clicking a navigation object highlights the model object that contains the requirements link.

To use an icon of your own choosing for the navigation object:

1 Select **Tools > Requirements > Settings**.

2 Select the **Selection Linking** tab.

3 Select **Modify destination for bi-directional linking**.

Selecting this option enables the **Use custom bitmap for navigation controls in documents** option.

4 Select **Use custom bitmap for navigation controls in documents**.

5 Click **Browse** to locate the file you want to use for the navigation objects.

For best results, use an icon file (`.ico`) or a small (16×16 or 32×32) bitmap image (`.bmp`) file for the navigation object. Other types of image files may give unpredictable results.

6 Select the desired file to use for navigation objects and click **Open**.

7 Close the Requirements Settings dialog box.

The next time you insert a navigation object into a requirements document, the RMI uses the file you selected.

Navigating Between a DOORS Requirement and a Model Object

In "Inserting Navigation Objects into DOORS Requirements" on page 7-5, you created a link between a DOORS requirement and the fuel_rate_control subsystem in the `sldemo_fuelsys` model. Navigate the links in both directions:

1 With the `sldemo_fuelsys` model closed, go to the DOORS requirement in the formal module.

2 Left-click the Simulink reference object that you inserted to select it.

3 Select **MATLAB > Select item**.
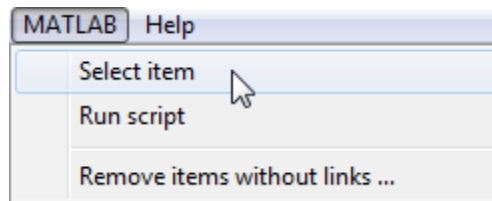
Your version of the `sldemo_fuelsys` model opens, with the fuel_rate_control subsystem highlighted.
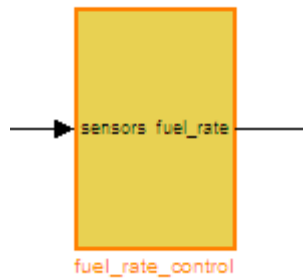
4 Log in to the DOORS software.

5 Navigate from the model to the DOORS requirement. In the Model Editor, right-click the fuel_rate_control subsystem and select **Requirements > 1. "*<requirement name>*"** where *<requirement name>* is the name of the DOORS requirement that you created.

The DOORS formal module opens with the requirement object and its child objects highlighted in red.

Why Add Navigation Objects to DOORS Requirements?

IBM Rational DOORS software is a requirements management application that you use to capture, track, and manage requirements. The Requirements Management Interface (RMI) allows you to link objects in a Simulink model to requirements managed by external applications, including the DOORS software.

When you create a link from a model object to a DOORS requirement, the RMI stores information about the DOORS object in the model. This information allows you to navigate from the model to the associated requirement.

You can configure the RMI to insert a navigation object in the DOORS database. This object helps you to navigate from the DOORS requirement to the model object.

To insert navigation objects into the DOORS database, you must have write access to the DOORS database.

Troubleshooting Your DOORS Installation

DXL Errors
If you try to synchronize your Simulink model to a DOORS project, you may see the following errors:

```
-E- DXL: <Line:2> incorrectly concatenated tokens
-E- DXL: <Line:2> undeclared variable (dmiRefreshModule)
-I- DXL: all done with 2 errors and 0 warnings
```

If you see these errors, exit the DOORS software, rerun the `rmi setup` command at the MATLAB command prompt, and restart the DOORS software.

Microsoft Office

Enabling Linking from Microsoft Office Documents to Simulink Models

By default, the RMI does not insert navigation objects into requirements documents. So that the RMI inserts navigation objects into the requirements document when you create a link from a model object to a requirement, you must change this setting in the RMI.

The RMI can insert navigation objects into the following applications:

- IBM Rational DOORS
- Microsoft Excel
- Microsoft Word

To enable linking from Microsoft Office documents to models:

1 Open any model. For example, open the Simulink model:

```
slvnvdemo_fuelsys_officereq
```

> **Note** You can modify requirements settings in the Requirements Settings dialog box, available only from the Model Editor. Even though you have a model open, any settings you change persist for all models you open subsequently. Changes you make are global and not specific to open models.

2 Select **Tools > Requirements > Settings**.

The Requirements Settings dialog box opens.

3 On the **Selection Linking** tab of the Requirements Settings dialog box:

- Set the **Document file reference** option to `path relative to model folder`.
- Enable **Modify destination for bi-directional linking**.

  When you select this option, every time you create a selection-based link from a model object to a requirement, the RMI inserts navigation objects at the designated location.

- To specify one or more user tags to apply to the links that you create, in the **Apply this user tag to new links** field, enter the tag names.

  For more information about user tags, see "Understanding Requirements Filtering" on page 4-25.

  For more information about the settings, see "The Requirement Settings Dialog Box" on page 2-17.

4 Click **Close** to close the Requirements Settings dialog box.

Keep the `slvnvdemo_fuelsys_officereq` model open.

Inserting Navigation Objects in Microsoft Office Requirements Documents

Use selection-based linking to create a requirement from the `slvnvdemo_fuelsys_officereq` model to a requirements document. If you have configured the RMI as described in "Enabling Linking Between DOORS Databases and Simulink Models" on page 7-4, the RMI inserts a navigation object into both the model and the requirement.

1 Open the Microsoft Word requirements document:

   matlabroot/toolbox/slvnv/rmidemos/fuelsys_req_docs/
   slvnvdemo_FuelSys_RequirementsSpecification.docx

2 Select the **Throttle Sensor** header.

3 In the slvnvdemo_fuelsys_officereq model, open the engine gas
   dynamics subsystem.

4 Right-click the Throttle & Manifold subsystem and select
   Requirements > Add link to Word selection.

5 The RMI inserts an URL-based link into the requirements document.

1.1.6. **Throttle Sensor**

Inserting Navigation Object That Link to Multiple Simulink Objects

If you have several model objects that correspond to a requirement, you can
link all the model objects to that requirement with one navigation object.
This eliminates the need to insert multiple navigation objects for a single
requirement. The model objects must be available in the same model diagram
or Stateflow chart.

The workflow for linking multiple objects to a Microsoft Word requirement
is as follows:

1 Make sure that the RMI is configured to insert navigation objects into
   requirements documents, as described in "Enabling Linking from Microsoft
   Office Documents to Simulink Models" on page 7-11.

2 Select the Microsoft Word requirement to link to.

3 Select the model objects that need to link to that requirement.

4 Right-click one of the objects and select **Requirements > Add link to
   Word selection**.

A single navigation object is inserted at the selected requirement.

5 Double-click the navigation object in Microsoft Word to highlight the model objects that are linked to that requirement.

Customizing Navigation Objects

If the RMI is configured to modify destination for bi-directional linking, the RMI inserts a navigation object into your requirements document. This object looks like the icon for the Simulink software:

Note In DOORS requirements documents, clicking a navigation objects does not navigate back to your Simulink model; you must select **MATLAB > Select object** to find the model object that contains the requirements link.

In Microsoft Office requirements documents, clicking a navigation object highlights the model object that contains the requirements link.

To use an icon of your own choosing for the navigation object:

1 Select **Tools > Requirements > Settings**.

2 Select the **Selection Linking** tab.

3 Select **Modify destination for bi-directional linking**.

Selecting this option enables the **Use custom bitmap for navigation controls in documents** option.

4 Select **Use custom bitmap for navigation controls in documents**.

5 Click **Browse** to locate the file you want to use for the navigation objects.

For best results, use an icon file (.ico) or a small (16×16 or 32×32) bitmap image (.bmp) file for the navigation object. Other types of image files may give unpredictable results.

6 Select the desired file to use for navigation objects and click **Open**.

7 Close the Requirements Settings dialog box.

The next time you insert a navigation object into a requirements document, the RMI uses the file you selected.

Navigating Between a Microsoft Word Requirement and a Model

In "Inserting Navigation Objects in Microsoft Office Requirements Documents" on page 7-12, you created a link between a Microsoft Word requirement and the Throttle & Manifold subsystem in the slvnvdemo_fuelsys_officereq demo model. Navigate these links in both directions:

1 In the slvnvdemo_fuelsys_officereq model, right-click the Throttle & Manifold subsystem and select **Requirements > 1. "Throttle Sensor"**.

The requirements document opens, and the header in the requirements document is highlighted.

2 In the requirements document, next to **Throttle Sensor**, click the navigation object.

The engine gas dynamics subsystem opens, with the Throttle & Manifold subsystem highlighted.
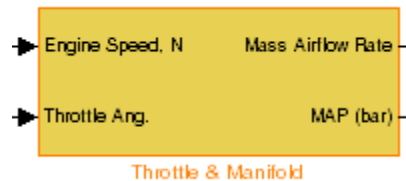
Why Add Navigation Objects to Microsoft Office Requirements?

You can use the Microsoft Word and Microsoft Excel applications to capture, track, and manage requirements. The RMI allows you to link objects in a Simulink model to requirements managed by external applications.

When you create a link from a model to a requirement in a Microsoft Office document, the RMI stores information about the requirement in the model. With this information, you can navigate from the model to the associated requirement.

You can configure the RMI to insert a navigation reference in the Microsoft Office document. With this object, you can navigate from the requirement to the model object.

To insert a navigation reference into both the model and a requirements document, you must have write access to the requirements document.

Troubleshooting Navigation Objects Created with ActiveX in Microsoft Office 2007 and 2010

Saving Requirements Documents to Microsoft Word 2007 or 2010 Format

If you create a requirements document with an earlier version of Microsoft Word than Word 2007, links to the Simulink model automatically work. If you open a document created in an earlier version and then save it in Microsoft Word 2007 format, make sure that the links to the models continue to work:

1 Open a requirements document saved in a release of Microsoft Word earlier than Microsoft Word 2007.

2 Depending on which version of Microsoft Word you are running, do one of the following:

- In Microsoft Word 2007, in the upper-left corner, click the **Microsoft Office Button**.

- In Microsoft Word 2010, select the **File** tab.

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

You see the following dialog box.
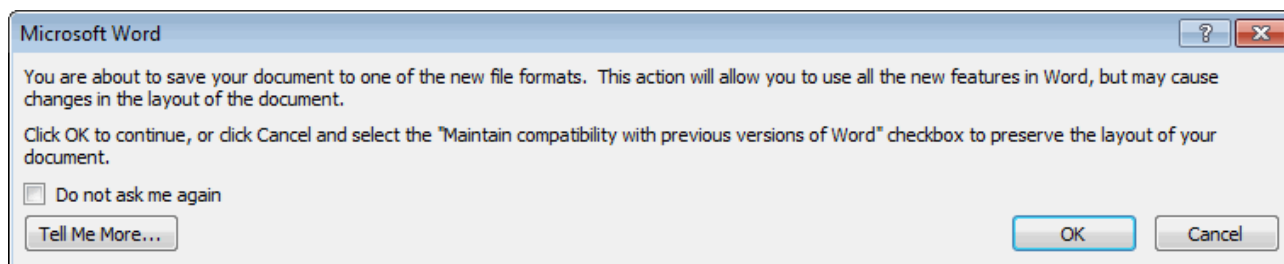
4 Click **OK**.

If you are running Microsoft Word 2007, you see the following dialog box.
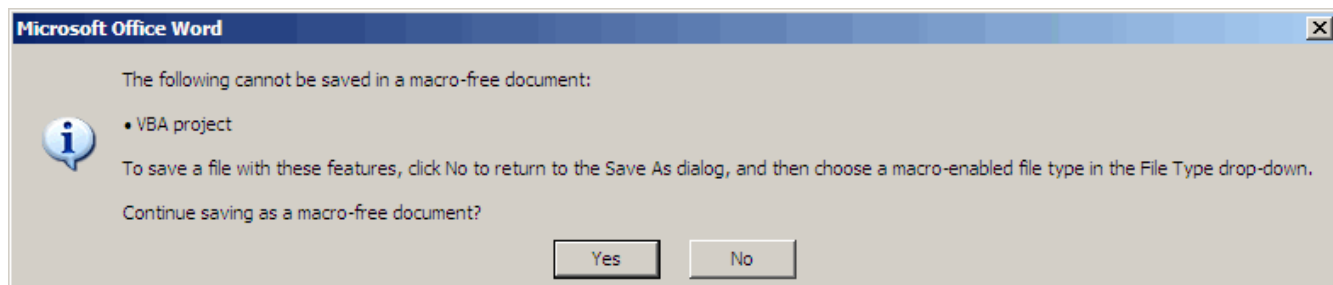
5 Click **Yes** to save the current document in the current Microsoft Word format, with a .docx extension.

Note You might need to enable ActiveX controls in the Microsoft Office Trust Center.

Field Codes in Requirements Documents

If your Microsoft Word requirements document displays the field codes in addition to, or instead of, the ActiveX icon, clear the **Show field codes instead of their values** option.

The following graphic shows a requirements document created in Microsoft Word 2003, with the field codes (`CONTROL mwSimulink1.SLRefButton \s`) displayed.

The following graphic shows a requirements document created in Microsoft Word 2007, with the field codes (`CONTROL mwSimulink1.SLRefButton`) displayed.

To hide the field codes and display the ActiveX icon:

1 Depending on which version of Microsoft Word you are running, do one of the following:

- In Microsoft Word 2007, in the upper-left corner, click the **Microsoft Office Button** and at the bottom of the window, click **Word Options**.

- In Microsoft Word 2010, select **File > Options**.

2 In the left-hand portion of the pane, click **Advanced**.

3 In the **Advanced** pane, scroll to the **Show document content** section and clear the **Show field codes instead of their values** option.
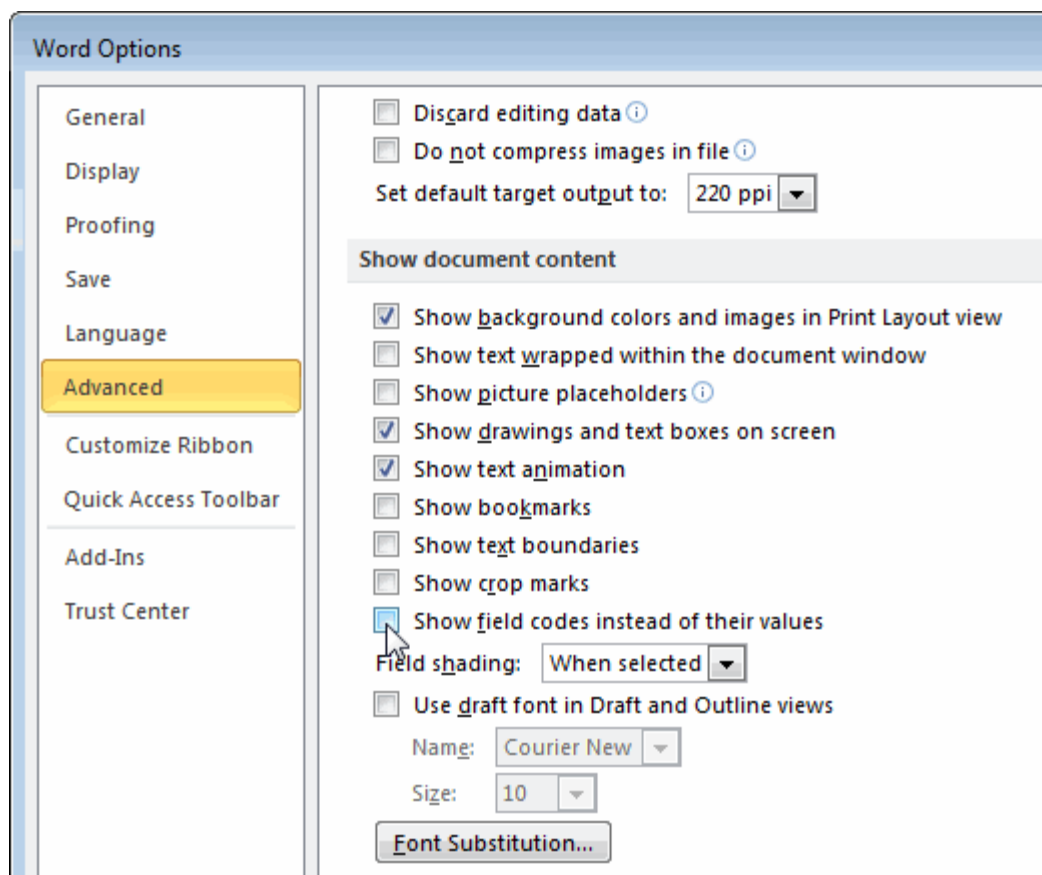
ActiveX Control Does Not Link to Model Object

If you click an ActiveX control that links to a Simulink or Stateflow object, and the object does not open, do one of the following:

- Store your requirements documents in trusted locations, as described in the Microsoft Office 2007 documentation. The Trust Center does not check files for ActiveX controls stored in trusted locations, so you can maintain your Trust Center restrictions.

- Enable ActiveX controls:

  1 Depending on which version of Microsoft Office you are using, do one of the following:

  - In Microsoft Word 2007 or Microsoft Excel 2007, in the upper-left corner, click the **Microsoft Office Button**.

    In the pane that opens, at the bottom, click **Word Options** or **Excel Options**, depending on which program you are running.

  - In Microsoft Word 2010 or Microsoft Excel 2010, select **File > Options**.

  2 In the left-hand portion of the pane, click **Trust Center**.

  3 In the **Trust Center** pane, click **Trust Center Settings**.

  4 In the **Trust Center** pane, on the right, select **ActiveX Settings**.
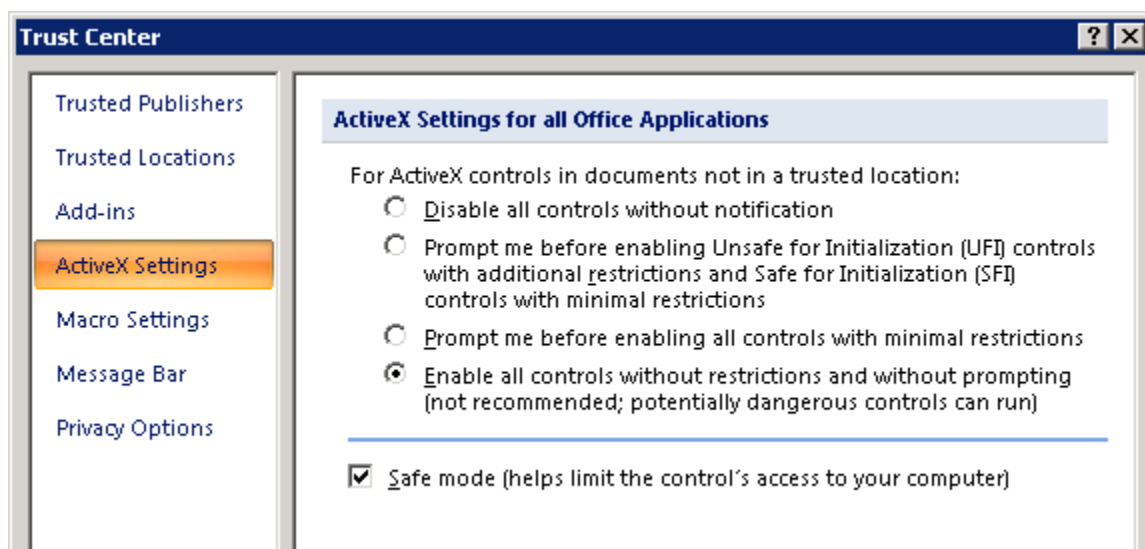
5 Select the setting that you want for ActiveX controls:

- **Prompt me for enabling all controls with minimal restrictions** to decide each time you click an ActiveX control if you want to enable all controls.

- **Enable all controls without restrictions and without prompting** to enable all ActiveX controls whenever you open the document.

6 Close the open dialog boxes.

7 Restart the application for the settings to take effect.

Deleting an ActiveX Control from Microsoft Excel 2007

Your document may have an ActiveX control in a worksheet cell:
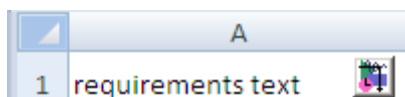
To remove an ActiveX control from your Microsoft Excel 2007 spreadsheet:

1 In Microsoft Excel 2007, in the upper-left corner, click the **Microsoft Office Button**.

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

3 In the Excel Options dialog box, in the left-hand pane, click **Popular**.

4 On the **Popular** pane, in the **Top options for working with Excel** section, select **Show Developer tab in the Ribbon**.

5 Click **OK**.

6 In the Ribbon, on the **Developer** tab, select **Design Mode**.

When you select **Design Mode**, the ActiveX control is no longer visible in the cell.

7 Click where the ActiveX control was, and you see four handles showing the location of the control.
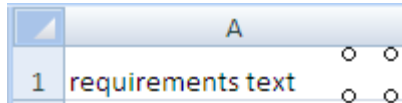
8 Select **Home** > **Cut** to delete the control.

Deleting an ActiveX Control from Microsoft Excel 2010

Your document may have an ActiveX control in a worksheet cell:
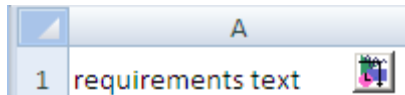
To remove an ActiveX control from your Microsoft Excel 2010 spreadsheet:

1 Select the control by clicking it.

2 Select **Home > Cut** to delete the control.

8

Custom Types of Requirements Documents

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 Requirements Management Interface (RMI). Then you can create requirement links from your model to these types of documents.

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 RMI does not support

The custom link type API allows you to define MATLAB functions that enable linking between your Simulink model and your custom requirements document type. These functions also enable new link creation and navigation between the model and documents.

For example, navigation involves opening a requirements document and finding the specific requirement record. When you click your custom link in the content menu of a linked object in the model, Simulink uses your custom link type navigation function to open the document and highlight the target requirement based on the implementation provided. The navigation function you implement uses the available API to communicate with your requirements storage application.

Typically, MATLAB runs an operating system shell command or uses ActiveX communication for sending navigation requests to external applications.

Alternatively, if your requirements are stored as custom variants of text or HTML files, you can use the built-in editor or Web browser to open the requirements document.

Workflow for Implementing Custom Link Types

To implement a custom link type:

1 Create a MATLAB function file based on the custom link type template, as described in "Custom Link Type Functions" on page 8-4.

2 Customize the custom link type file to specify the link type properties and custom callback functions required for the custom link type, as described in "Link Type Properties" on page 8-6.

3 Register the custom link type using the `rmi` command `'register'` option, as described in "Custom Link Type Registration" on page 8-10.

Custom Link Type Functions

To create a MATLAB function file, start with the custom link type template, located in:

matlabroot/toolbox/slvnv/reqmgt/linktype_TEMPLATE.m

Your custom link type function:

- Must exist on the MATLAB path with a unique function and file name.

- Cannot require any input arguments.

- Must return a single output argument that is an instance of the requirements link type class.

To view similar files for the built-in link types, see the following files in *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_url.m
linktype_rmi_word.m
```

Links and Link Types

Requirements links are the data structures, managed by Simulink, that identify a specific location within a document. You get and set the links on a block using the `rmi` command.

Links and link types work together to perform navigation and manage requirements. The `doc` and `id` fields of a link uniquely identify the linked item in the external document. The RMI passes both of these strings to the navigation command when you navigate a link from the model.

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 function that creates the link type. The RMI stores this name in the Simulink model.
Label	A string to identify this link type. In the Requirements dialog box, this string appears on the **Document type** drop-down list for a Simulink or Stateflow object.
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. • In the Requirements dialog box, when you click **Browse**, the file selection dialog box opens.
Extensions	An array of file extensions. Use these file extensions as filter options in the Requirements dialog box when you click **Browse**. The file extensions infer the link type based on the document name. If you registered more than one link type for the same file extension, the link type that you registered takes first 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 in the Requirements dialog box **Location** drop-down list.

Property	Description
NavigateFcn	The MATLAB callback 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)`
ContentsFcn	The MATLAB callback 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, if the link type is not a file, the **Document** field contents. The function returns three outputs: • Labels • Depths • Locations
BrowseFcn	The MATLAB callback invoked when you click **Browse** in the Requirements dialog box. You do not need this function when the link type is a file. The function takes no input arguments and returns a single output argument that identifies the selected document.
CreateURLFcn	The MATLAB callback that constructs a path name to the requirement. This function uses the document path or URL to create a specific requirement URL. The requirement URL is based on a location identifier specified in the third input argument. The input arguments are: • Full path name to the requirements document • Info about creating a URL to the document (if applicable) • Location of the requirement in the document

Property	Description
	This function returns a single output argument, the string to use when navigating to the requirement from the generated report.
IsValidDocFcn	The MATLAB callback invoked when you run a requirements consistency check. The function takes one input argument—the fully qualified name for the requirements document. It returns true if the document can be located; it returns false if the document cannot be found or the document name is invalid.
IsValidIdFcn	The MATLAB callback invoked when you run a requirements consistency check. This function takes two input arguments: • Fully qualified name for the requirements document • Location of the requirement in the document IsValidIdFcn returns true if it finds the requirement and false if it cannot find that requirement in the specified document.
IsValidDescFcn	The MATLAB callback invoked when you run a requirements consistency check. This function has three input arguments: • Full path to the requirements document • Location of the requirement in the document • Requirement description label as stored in Simulink IsValidDescFcn returns two outputs: • True if the description matches the requirement, false otherwise.

Property	Description
	• The requirement label in the document, if not matched in Simulink.
DetailsFcn	The MATLAB callback invoked when you generate the requirements report with the **Include details from linked documents** option. This function returns detailed content associated with the requirement and has three input arguments: • Full path to the requirements document • Location of the requirement in the document • Level of details to include in report (Unused) The DetailsFcn returns two outputs: • Numeric array that describes the hierarchical relationship among the fragments in the cell array • Cell array of formatted fragments (paragraphs, tables, et. al.) from the requirement
SelectionLinkFcn	The MATLAB callback invoked when you use the selection-based linking menu option for this document type. This function has two input arguments: • Handle to the model object that will have the requirement link • True if a navigation object is inserted into the requirements document, or false if no navigation object is inserted SelectionLinkFcn returns the requirements link structure for the selected requirement.

Custom Link Type Registration

Register your custom link type by passing the name of the MATLAB function file to the `rmi` command as follows:

```
rmi register mytargetfilename
```

Once you register a link type, it appears in the Requirements dialog box as an entry in the **Document type** drop-down list. A file in your preference folder contains the list of registered link types, so the custom link type is loaded each time you run MATLAB.

When you create links using custom link types, the software saves the registration name and the other link properties specified in the function file. When you attempt to navigate to such a link, the RMI resolves the link type against the registered list. If the software cannot find the link type, you see an error message.

You can remove a link type with the following MATLAB command:

```
rmi unregister mytargetfilename
```

Creating a Custom Requirements Link Type

In this example, you implement a custom link type to a hypothetical document type, a text file with the extension .abc. Within this 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 (").

You will create a document index listing 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 on the MATLAB path.

For this example, the file is rmicustabcinterface.m, containing the function, rmicustabcinterface, that implements the ABC files shipping with your installation.

2 To view this function, at the MATLAB prompt, type:

```
edit rmicustabcinterface
```

The file rmicustabcinterface.m opens in the MATLAB Editor. The content of the file is:

```
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 1984-2010 The MathWorks, Inc.
%  $Revision: 1.1.6.10.2.1 $  $Date: 2011/12/02 18:56:38 $

    % 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 > O
        if matlab.desktop.editor.isEditorAvailable
            matlab.desktop.editor.openAndGoToLine(fileName, lineNum);
        end
    else
        edit(fileName);
    end


function openFileToItem(fileName, itemName)
    reqStr = ['Requirement:: "' itemName '"'];
    lineNum = O;
    fid = fopen(fileName);
    i   = 1;
    while lineNum == O
        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 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 = [];
```

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

```
rmi register rmicustabcinterface
```

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

4 Create a text file with the .abc extension containing several requirement items marked by the Requirement:: string.

For your convenience, an example file ships with your installation. The example file is *matlabroot*\toolbox\slvnv\rmidemos\demo_req_1.abc. 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 of "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 of "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 of "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 of "Glide Slope Armed"


Requirement:: "Glide Slope Coupled"

Glide Slope control becomes coupled when the control
is armed and (Glide Slope Angle Error > O) 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 of "Glide Slope Coupled"
```

5 Open the following demo model:

aero_dap3dof

6 Right-click the Reaction Jet Control subsystem and select
Requirements > Edit/Add Links.

The Requirements dialog box opens.

7 Click **New** to add a new requirement link. The **Document type** drop-down
list now contains the ABC file (for demonstration) option.
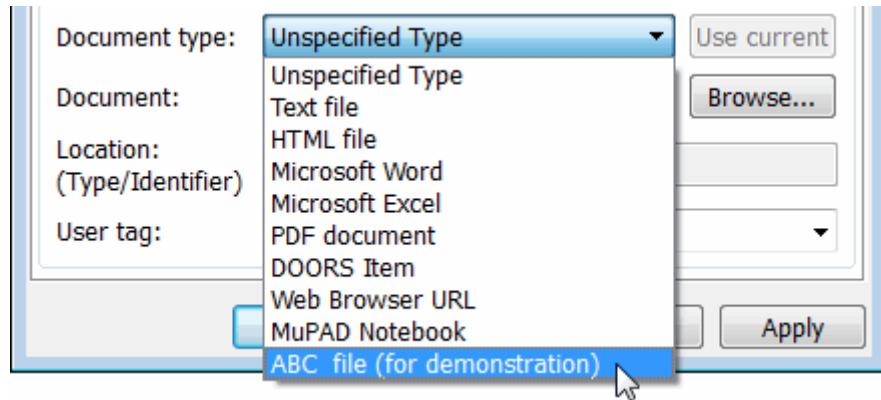
8 Set **Document type** to ABC file (for demonstration) and browse to the *matlabroot*\toolbox\slvnv\rmidemos\demo_req_1.abc file. The browser shows only the files with the .abc extension.

9 To define a particular location in the requirements document, use the **Location** field.

In this example, the rmicustabcinterface function specifies two types of location delimiters for your requirements:

- > — Line number in a file
- @ — Named item, such as a bookmark, function, or HTML anchor

---

Note The rmi reference page describes other types of requirements location delimiters.

The **Location** drop-down list contains these two types of location delimiters whenever you set **Document type** to ABC file (for demonstration).
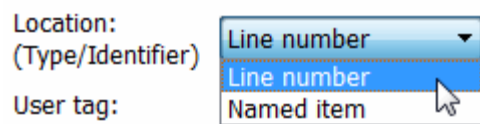
10 Select **Line number**. Enter the number 26, which corresponds with the line number for the Altitude Hold requirement in demo_req_1.abc.

11 In the **Description** field, enter Altitude Hold, to identify the requirement by name.

12 Click **Apply**.

13 Verify that the Altitude Hold requirement links to the Reaction Jet Control subsystem. Right-click the subsystem and select **Requirements > 1. "Altitude Hold"**.

Creating a Document Index

A *document index* is a list of all the requirements in a given document. To create a document index, MATLAB uses file I/O functions to read the contents of a requirements document into a MATLAB variable. The RMI extracts the list of requirement items.

The example requirements document, demo_req_1.abc, defines four requirements using the string Requirement::. To generate the document index for an ABC file, the ContentsFcn function, in the rmicustabcinterface.m file, extracts the requirements names and inserts @ before each name.

For the demo_req_1.abc file, in the **Requirements: Engine** dialog box, click the **Document Index** tab. The ContentsFcn function generates the document index automatically.

Requirements: Reaction Jet Control

| Requirements | Document Index |

Content of demo_req_1.abc

Refresh

Double click to update fields with selected item

@Altitude Hold
@Autopilot Disable
@Glide Slope Armed
@Glide Slope Coupled

OK   Cancel   Help   Apply
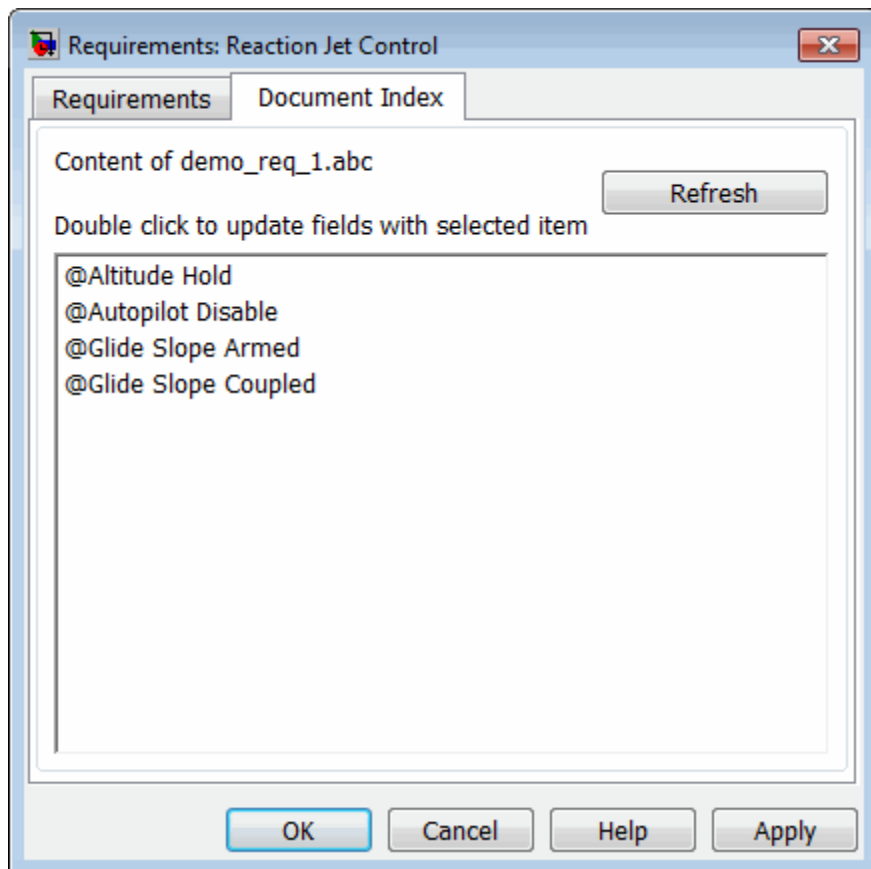
Navigating to Simulink Objects from External Documents

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, the RMI uses a globally unique identifier for that object. This identifier identified the object. The identifier does not change if you rename or move the object, or add or delete requirement links. The RMI uses the unique identifier only to resolve an object within a model.

Using the rmiobjnavigate Function

The rmiobjnavigate function identifies the Simulink or Stateflow object, highlights that object, and brings the editor window to the front of the screen. When you navigate to a Simulink model from an external application, invoke this function.

The first time you navigate to an item in a particular model, you might experience a slight delay while the software initializes the communication API and the internal data structures. You do not experience a long delay on subsequent navigation.

Determining the Navigation Command

To create a requirement link for a Simulink or Stateflow object, at the MATLAB prompt, use the following command to find the navigation command string, where obj is a handle or a uniquely resolved name for the object:

```
[ navCmd, objPath ] = rmi('navCmd', obj);
```

The return values of the navCmd method are:

- navCmd — A string that navigates to the object when evaluated by the MATLAB software.

- objPath — A string that identifies the model object

Send the `navCmd` strings to the MATLAB software for evaluation when navigating from the external application to the object `obj` in the Simulink model. Use and `objPath` string to visually identify the target object in the requirements document.

Using the ActiveX Navigation Control

The RMI uses software that includes a special Microsoft ActiveX control to enable navigation to Simulink objects from Microsoft Word and Excel documents. You can 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 control ToolTip. The `MLEvalCmd` property is the string that you pass to the MATLAB software for evaluation when you click the control.

Typical Code Sequence for Establishing Navigation Controls

When you create an interface to an external tool, you can automate the procedure for establishing links. This way, you do not need to manually update the dialog box fields. This type of automation occurs as part of the selection-based linking 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. Create a requirement link on the selected object using the RMI API as follows:

   a Create an empty link structure using the following command:

```
rmi('createempty')
```

b Fill in the link structure fields based on the target location in the requirements document.

c Attach the link to the object using the following command:

```
rmi('cat')
```

4 Determine the MATLAB navigation command string that you must embed in the external tool, using the `navCmd` method:

```
[ navCmd, objPath ] = 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 property.

When using ActiveX navigation objects provided by the external tool, set the `MLEvalCmd` property to the `navCmd` and set the `tooltipstring` property to `objPath`.

You define the MATLAB code implementation of this procedure as the `SelectionLinkFcn` function in the link type definition file. The following files in *matlabroot*\toolbox\slvnv\reqmgt\private contain examples of how to implement this functionality:

```
linktype_rmi_doors.m
linktype_rmi_excel.m
linktype_rmi_word.m
```

9

Requirements Information in Generated Code

- "How Requirements Information is Included in Generated Code" on page 9-2
- "Generating Code for Models with Requirements Links" on page 9-3

How Requirements Information is Included in Generated Code

After you simulate your model and verify its performance against the requirements, you can generate code from the model for an embedded real-time application. The Embedded Coder™ software generates code for Embedded Real-Time (ERT) targets.

If the model has any links to requirements, the Embedded Coder software inserts information about the requirements links into the code comments.

For example, if a block has a requirement link, the software generates code for that block. In the code comments for that block, the software inserts:

- Requirement description
- Hyperlink to the requirements document that contains the linked requirement associated with that block

Note You must have a license for Embedded Coder to generate code for an embedded real-time application.

Comments for the generated code include requirements descriptions and hyperlinks to the requirements documents in the following locations.

Model Object with Requirement	Location of Code Comments with Requirements Links
Model	In the main header file, `<model>.h`
Nonvirtual subsystem	At the call site for the subsystem
Virtual subsystem	At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem does not have a nonvirtual parent, requirement descriptions appear in the main header file for the model, `<model>.h`.
Nonsubsystem block	In the generated code for the block

Generating Code for Models with Requirements Links

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

1 Open the rtwdemo_requirements demo model.

2 Select **Simulation > Configuration Parameters**.

3 In the **Select** pane of the Configuration Parameters dialog box, select the **Code Generation** category.

The currently configured system target must be an ERT target.
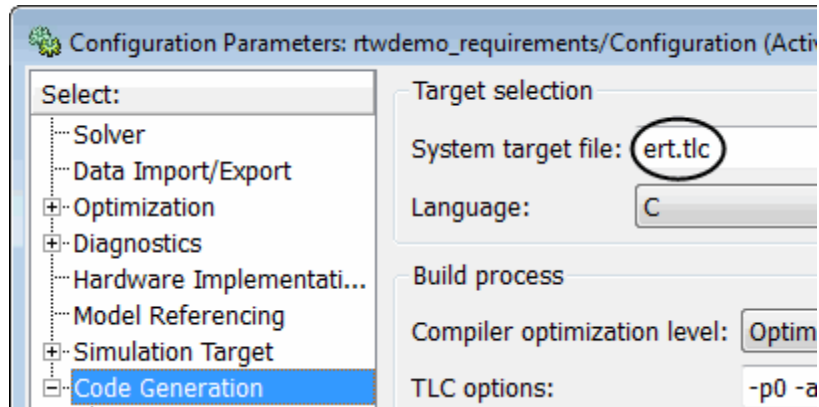
4 Under the **Code Generation** category, select **Comments**.

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

6 Under the **Code Generation** category, select **Report**.

7 On the **Report** pane, select:

- **Create code generation report**
- **Open report automatically**

8 On the **Code Generation** main pane, click **Build**.

9 In the code-generation report, open rtwdemo_requirements.c.

10 Scroll to the code for the Pulse Generator block, clock. The comments for the code associated with that block include a hyperlink to the requirement linked to that block.

```
                                    119
rtwdemo_requirements.c              120      /* DiscretePulseGenerator: '<Root>/clock'
rtwdemo_requirements.h              121       *
rtwdemo_requirements_p             122       * Block requirements for '<Root>/clock':
rtwdemo_requirements_t             123       *  1. Clock period shall be consistent with chirp tolerance
                                    124       */
```

11 Click the link Clock period shall be consistent with chirp tolerance to open the HTML requirements document to the associated requirement.

Note When you click a requirements link in the code comments, the software opens the application for the requirements document, *except* if the requirements document is a DOORS module. To view a DOORS requirement, start the DOORS software and log in before clicking the hyperlink in the code comments.

Model Component Testing

- Chapter 10, "Overview of Component Verification"
- Chapter 11, "Verifying Generated Code for a Component"

10

Overview of Component Verification

What Is Component Verification?

In this section...
"Component Verification Approaches" on page 10-2
"Using Simulink® Verification and Validation™ Tools for Component Verification" on page 10-2

Component Verification Approaches

Component verification allows you to test a design component in your model using one of two approaches:

- **Within the context of the model that contains the component** — Using systematic simulation of closed-loop controllers requires that you verify components within a control system model. Doing so allows you to test the control algorithms with your model. This approach is called *system analysis*.

- **As standalone components** — For a high level of confidence in the component algorithm, verify the component in isolation from the rest of the system. This approach is called *component analysis*.

  Verifying standalone components provides several advantages:

  - You can use the analysis to focus on portions of the design that you cannot test because of the physical limitations of the system being controlled.

  - You can use this approach for open-loop simulations to test the plant model without feedback control.

  - You can use this approach when the model is not yet available or when you need to simulate a control system model in accelerated mode for performance reasons.

Using Simulink Verification and Validation Tools for Component Verification

By isolating the component to verify and using tools that the Simulink Verification and Validation software provides, you create test cases that allow

you to expand the scope of the testing for large models. This expanded testing helps you accomplish the following:

- Achieve 100% model coverage — If certain model components do not record 100% coverage, the top-level model cannot achieve 100% coverage. By verifying these components individually, you can create test cases that fully specify the component interface, allowing the component to record 100% coverage.

- Debug the component — To verify that each model component satisfies the specified design requirements, you can create test cases that verify that specific components perform as designed.

- Test the robustness of the component — To verify that a component handles unexpected inputs and calculations properly, you can create test cases that generate data. Then, test the error-handling capabilities in the component.

Workflows for Component Verification

Common Workflow for Component Verification

The following graphic illustrates the common workflow for component verification.
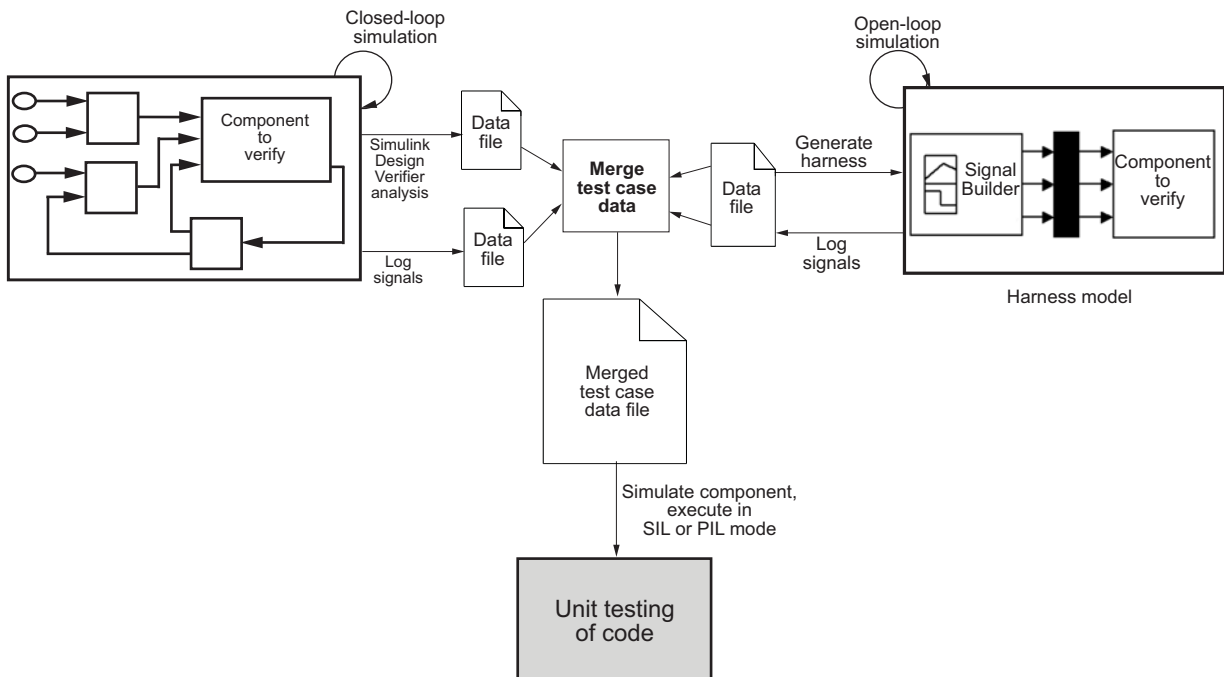
This graphic illustrates the two approaches for component verification, described in "What Is Component Verification?" on page 10-2:

1 Choose your approach for component verification:

- For closed-loop simulations, verify a component within the context of its container model by logging the signals to that component and storing them in a data file. If those signals do not constitute a complete test suite, generate a harness model and add or modify the test cases in the Signal Builder.

- For open-loop simulations, verify a component independently of the container model by extracting the component from its container model and creating a harness model for the extracted component. Add or modify test cases in the Signal Builder and log the signals to the component in the harness model.

2 Prepare component for verification.

3 Create and log test cases. If desired, merge the test case data into a single data file.

The data file contains the test case data for simulating the component. If you cannot achieve the desired results with a certain set of test cases, add new test cases or modify existing test cases in the data file, and merging them into a single data file.

Continue adding or modifying test cases until you achieve a test suite that satisfies the goals of your analysis.

4 Execute the test cases in Software-in-the-Loop or Processor-in-the-Loop mode.

5 After you have a complete test suite, you can:

- Simulate the model and execute the test cases to:

  - Record coverage.

  - Record output values to make sure that you get the expected results.

- Invoke the Code Generation Verification (CGV) API to execute the generated code for the model that contains the component in simulation, Software-in-the-Loop (SIL), or Processor-in-the-Loop (PIL) mode.

> **Note** To execute a model in different modes of execution, you use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see "Programmatic Code Generation Verification".

The next sections describe the steps for component verification in more detail:

- "Verifying a Component Independently of the Container Model" on page 10-6

- "Verifying a Model Block in the Context of the Container Model" on page 10-7

Verifying a Component Independently of the Container Model

Use component analysis to verify:

- Model blocks

- Atomic subsystems

- Stateflow atomic subcharts

The recommended steps for verifying a component independently of the container model:

1 Depending on the type of component, take one of the following actions:

- Model blocks — Open the referenced model.

- Atomic subsystems — Extract the contents of the subsystem into its own Simulink model.

- Atomic subcharts — Extract the contents of the Stateflow atomic subchart into its own Simulink model.

2 Create a harness model for:

- The referenced model

- The extracted model that contains the contents of the atomic subsystem or atomic subchart

3 Add or modify test cases in the Signal Builder in the harness model.

4 Log the input signals from the Signal Builder to the test unit.

5 Repeat steps 3 and 4 until you are satisfied with the test suite.

6 Merge the test case data into a single file.

7 Depending on your goals, take one of the following actions:

- Execute the test cases to:
  - Record coverage.
  - Record output values and make sure that they equal the expected values.
- Invoke the Code Generation Verification (CGV) API to execute the test cases in Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL) mode on the generated code for the model that contains the component.

If the test cases do not achieve the desired results, repeat steps 3 through 5.

Verifying a Model Block in the Context of the Container Model

Use system analysis to verify a Model block in the context of the block's container model. Use this technique when you analyze a closed-loop controller.

The recommended steps for system analysis:

1 Log the input signals to the component by simulating the container model.

or

Analyze the model using the Simulink Design Verifier™ software.

2 If you want to add test cases to your test suite or modify existing test cases, create a harness model using the logged signals.

3 Add or modify test cases in the Signal Builder in the harness model.

4 Log the input signals from the Signal Builder to the test unit.

5 Repeat steps 3 and 4 until you are satisfied with the test suite.

6 Merge the test case data into a single file.

7 Depending on your goals, do one of the following:

- Execute the test cases to:
  - Record coverage.
  - Record output values and make sure that they equal the expected values.
- Invoke the Code Generation Verification (CGV) API to execute the test cases in Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL) mode on the generated code for the model.

If the test cases do not achieve the desired results, repeat steps 3 through 5.

Functions for Component Verification

The Simulink Verification and Validation software provides several functions that facilitate the tasks associated with component verification.

Task	Function
Simulate a Simulink model and log input signals to a Model block in the model. If you modify the test cases in the Signal Builder harness model, use this approach for logging input signals to the harness model itself.	`slvnvlogsignals`
Create a harness model for a component, using logged input signals if specified, or using the default signals. A harness model contains four Simulink blocks as described in "Preparing the Component for Verification" on page 11-5 in Chapter 11, "Verifying Generated Code for a Component".	`slvnvmakeharness`
Merge test case data into a single data structure for batch execution or harness generation.	`slvnvmergedata`
Merge test cases from several harness models into a single harness model.	`slvnvmergeharness`
Extract an atomic subsystem or atomic subchart into a new model.	`slvnvextract`
Simulate a model, executing the specified test cases to record model coverage and outport values.	`slvnvruntest`
Invoke the Code Generation Verification (CGV) API, and execute the specified test cases on the generated code for the model.	`slvnvruncgvtest`

Component verification functions do not support the following Simulink software features:

- Variable-step solvers for `slvnvruntest`
- Component interfaces that contain:

- Complex signals
- Variable-size signals
- Array of buses
- Multiword fixed-point data types

Verifying Generated Code for a Component

Verifying Generated Code for a Component

In this section...

About the Example Model

This example uses the slvnvdemo_powerwindow demo model to show how to verify a component in the context of the model that contains that component. As you work through this example, you use the Simulink Verification and Validation component verification functions to create test cases and measure coverage for a referenced model. In addition, you execute the referenced model in both simulation mode and Software-in-the-Loop (SIL) mode using the Code Generation Verification (CGV) API and then compare the results.

Note You must have the following product licenses to run this example:

- Stateflow
- Embedded Coder
- Simulink Coder™

The component you verify is a Model block named control. This component resides inside the power_window_control_system subsystem in the top level of the slvnvdemo_powerwindow model.
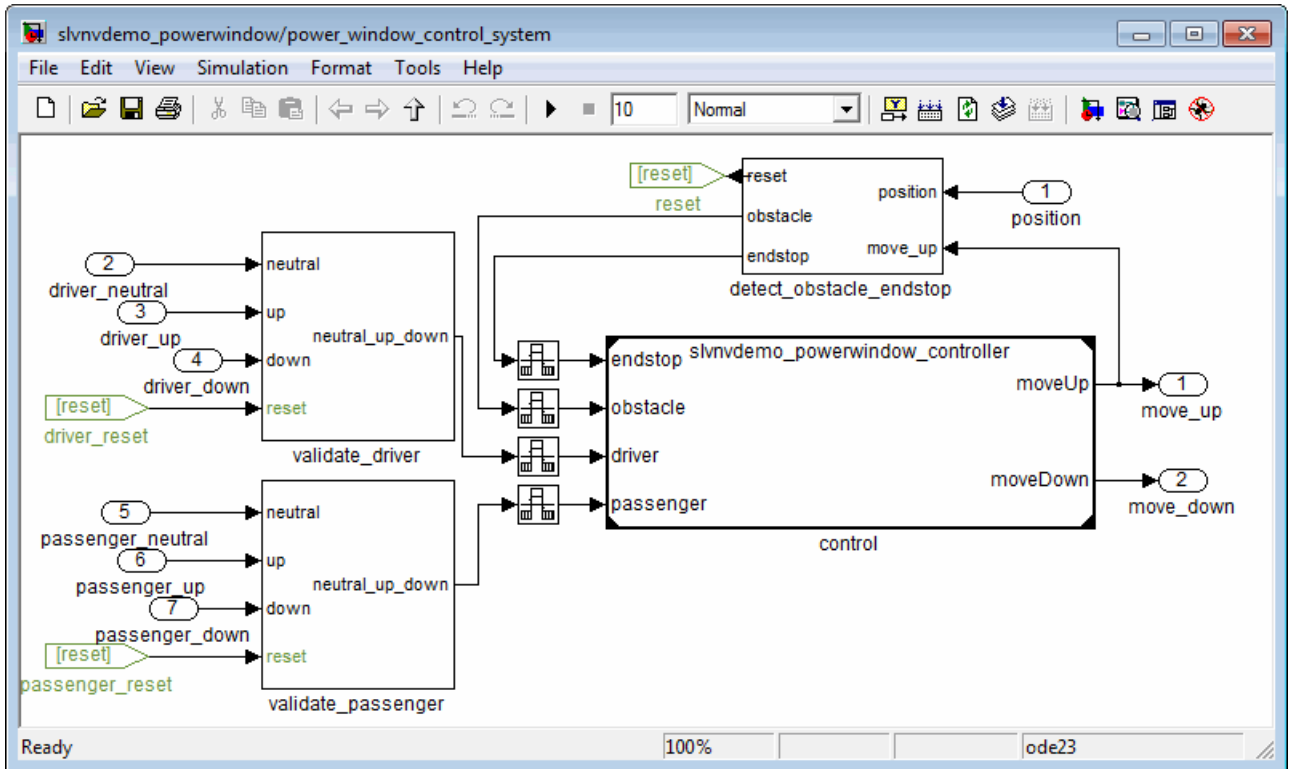
The Model block references the `slvnvdemo_powerwindow_controller` model.
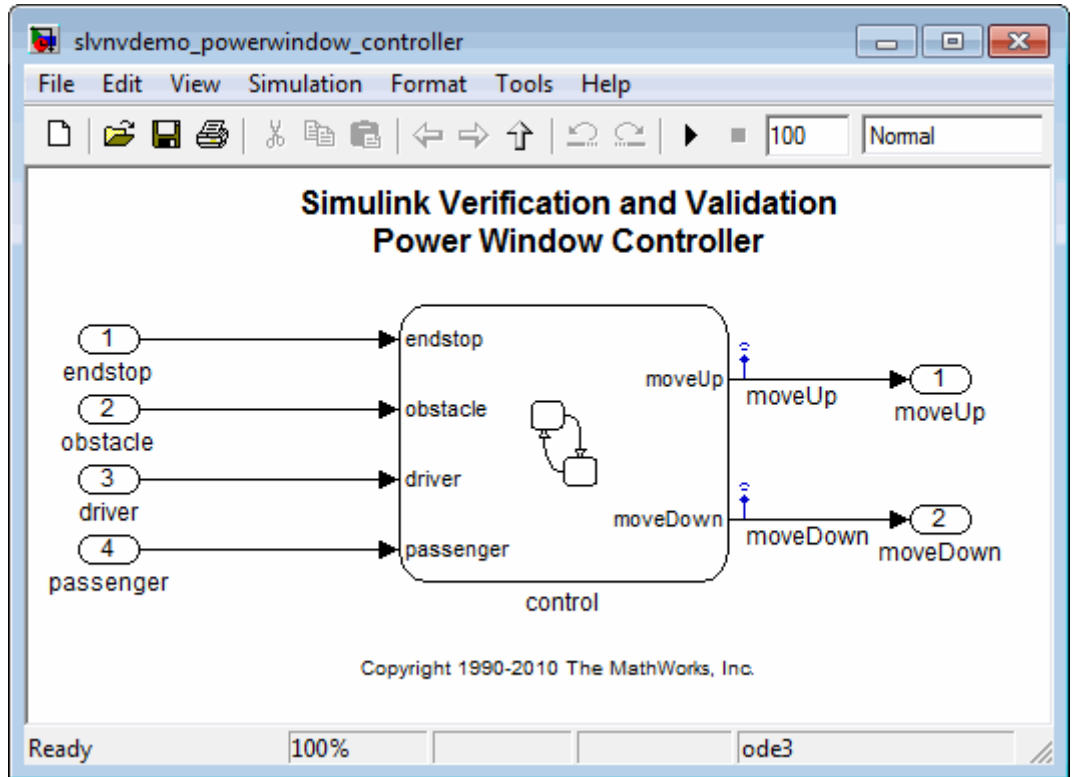
The referenced model contains a Stateflow chart `control`, which implements the logic for the power window controller.
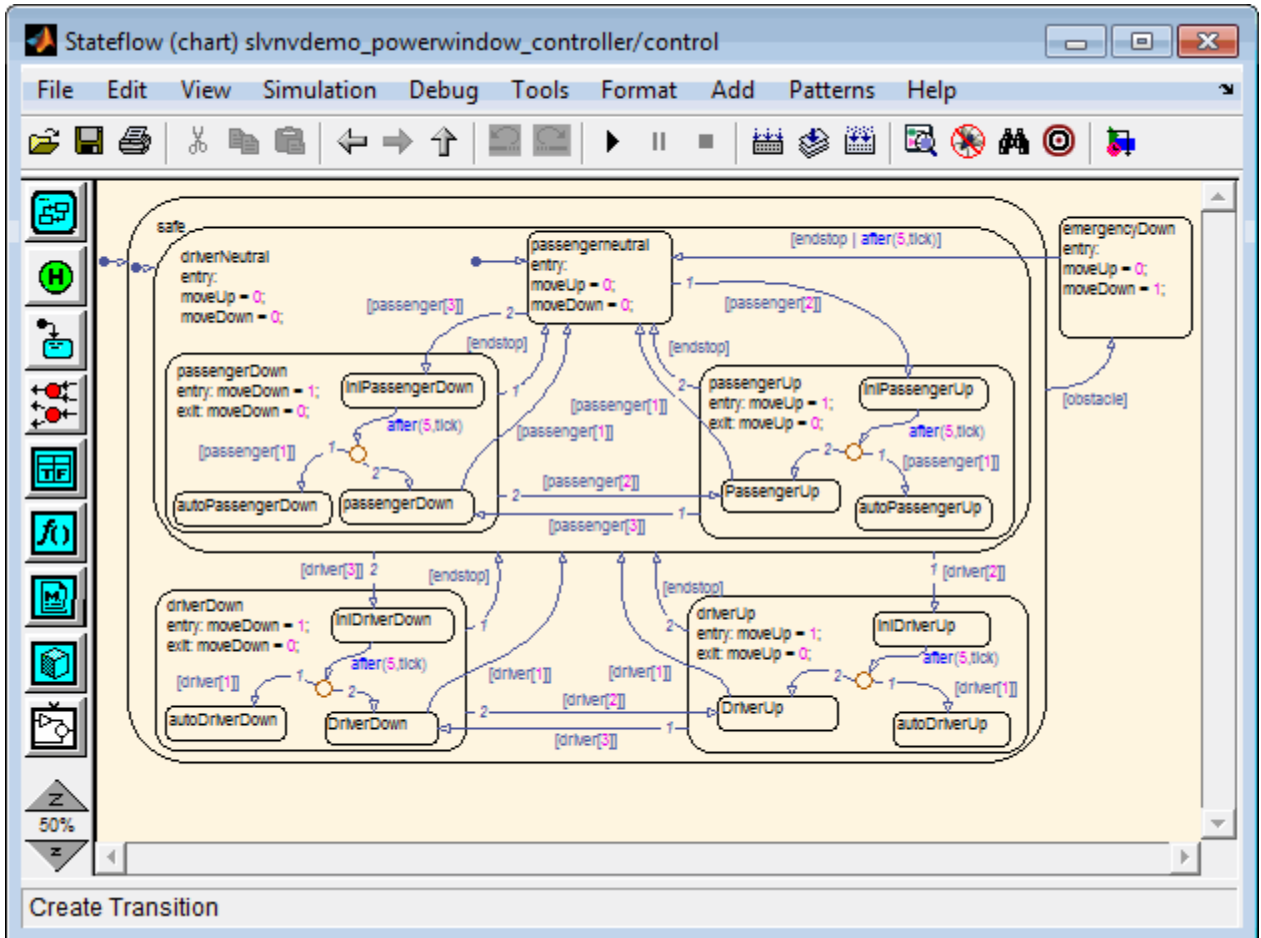
Preparing the Component for Verification

To verify the referenced model slvnvdemo_powerwindow_controller, you need to create a harness model that contains the input signals that simulate the controller in the plant model. Perform the following steps:

1 Open the slvnvdemo_powerwindow demo model:

    slvnvdemo_powerwindow

2 Open the power_window_control_system subsystem.

3 The Model block named control in the power_window_control_system subsystem references the component you verify during this example—slvnvdemo_powerwindow_controller. Load the referenced model:

```
load_system('slvnvdemo_powerwindow_controller');
```

4 Simulate the Model block that references slvnvdemo_powerwindow_controller and log the input signals to the Model block:

```
loggedSignalsPlant = ...
    slvnvlogsignals(...
        'slvnvdemo_powerwindow/power_window_control_system/control');
```

slvnvlogsignals stores the logged signals in loggedSignalsPlant.

5 Generate an empty harness model so that you can create new test cases manually:

```
harnessModelFilePath = ...
    slvnvmakeharness('slvnvdemo_powerwindow_controller');
```

slvnvmakeharness creates a harness model named slvnvdemo_powerwindow_controller_harness. The harness model includes:

- Test Unit — A Model block that references the slvnvdemo_powerwindow_controller model.

- Inputs — A Signal Builder block that contains one test case. That test case specifies the values of the input signals logged when the model slvnvdemo_powerwindow was simulated.

- Test Case Explanation — A DocBlock block that describes the test case.

- Size-Type — A Subsystem block that transmits signals from the Inputs block to the Test Unit block. The output signals from this block match the input signals for the Model block you are verifying.

- moveUp and moveDown — Two output ports that match the output ports from the Model block.
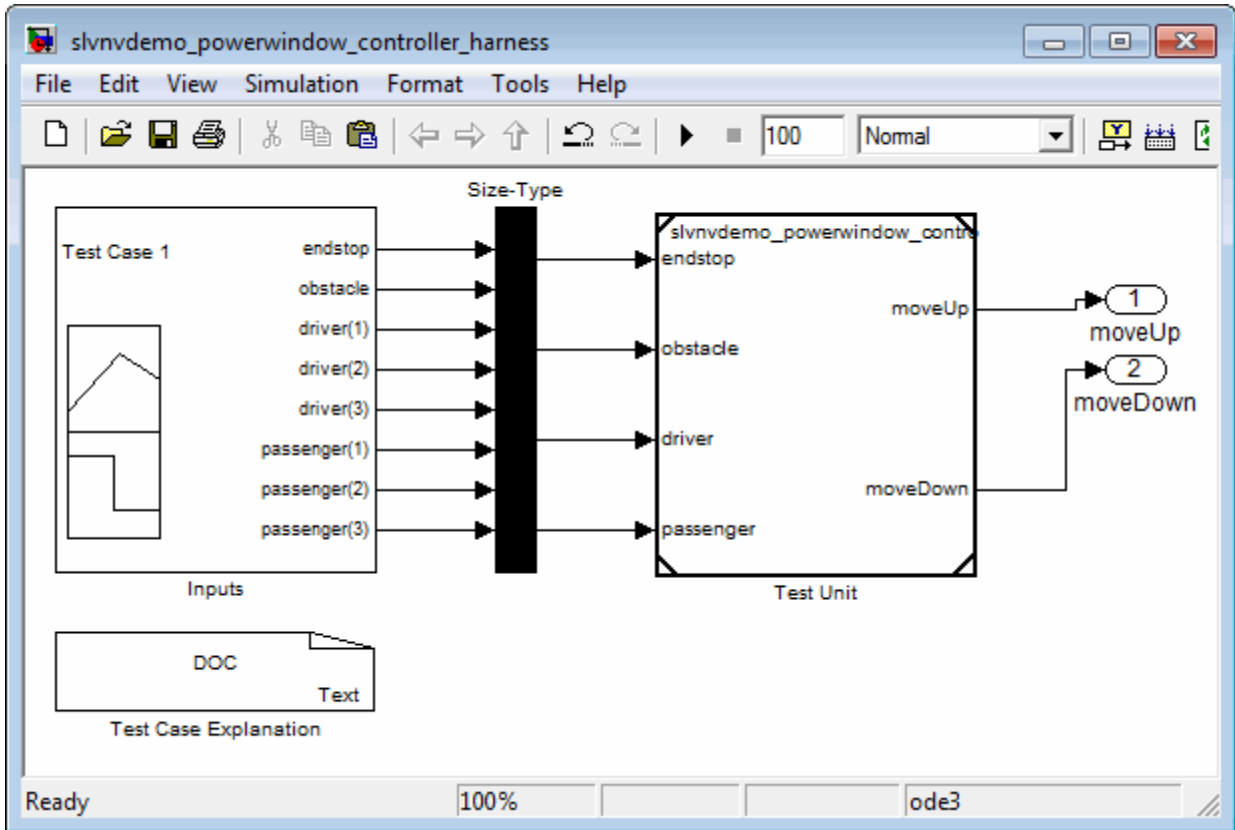
6 Save the name of the harness model for use later in this example:

```
[~,harnessModel] = fileparts(harnessModelFilePath);
```

7 Leave all models open for the next part of this example.

Next, create a test case that tests values for input signals to the component.

Creating and Logging Test Cases

Add a test case for your component to help you get closer to achieving 100% coverage.

For this example, use the `signalbuilder` function to add a new test case to the Signal Builder block in the harness model. The new test case specifies new values for the input signals to the component:

1 Load the file that contains the data for the new test case into the MATLAB workspace:

```
load('slvnvdemo_powerwindow_controller_newtestcase.mat');
```

The workspace variables `newTestData` and `newTestTime` contain the test-case data.

2 Add the new test case to the Signal Builder block in the harness model.

```
signalBuilderBlock = slvnvdemo_signalbuilder_block(harnessModel);
signalbuilder(signalBuilderBlock,'Append',...
    newTestTime, newTestData,...
    {'endstop','obstacle','driver(1)','driver(2)','driver(3)',...
    'passenger(1)','passenger(2)','passenger(3)'},'New Test Case');
```

3 Simulate the harness model with both test cases, then log the signals to the referenced model, and save the results:

```
loggedSignalsHarness = slvnvlogsignals(harnessModel);
```

Next, record coverage for the `slvnv_powerwindow_controller` model.

Merging the Test Case Data

You have two sets of test case data:

- `loggedSignalsPlant` — Logged signals to the Model block control

- `loggedSignalsHarness` — Logged signals to the test cases you added to the empty harness

To simulate all the test data at the same time, merge the two data files into a single data file:

1 Combine the test case data:

```
mergedTestCases = slvnvmergedata(loggedSignalsPlant,...
```

```
        loggedSignalsHarness);
```

2 View the merged data:

```
disp(mergedTestCases);
```

Next, simulate the referenced model with the merged data and recover coverage for the referenced model, `slvnv_powerwindow_controller`.

Recording Coverage for the Component

Model coverage is a measure of how thoroughly a test case tests a model and the percentage of pathways that a test case exercises. To record coverage for the `slvnv_powerwindow_controller` model:

1 Create a default options object, required by the `slvnvruntest` function:

```
runopts = slvnvruntestopts;
```

2 Specify to simulate the model, and record coverage:

```
runopts.coverageEnabled = true;
```

3 Simulate the model using the logged input signals:

```
[~, covdata] = slvnvruntest('slvnvdemo_powerwindow_controller',...
    mergedTestCases,runopts);
```

4 Display the HTML coverage report:

```
cvhtml('Coverage with Test Cases from Harness', covdata);
```

The `slvnv_powerwindow_controller` model achieved:

- Decision coverage: 44%
- Condition coverage: 45%
- MCDC coverage: 10%

Note For more information about decision coverage, condition coverage, and MCDC coverage, see "Types of Model Coverage" on page 15-4.

If you do not achieve the desired coverage, continue to modify the test cases in the Signal Builder in the harness model, log the input signals to the harness model, and repeat the preceding steps until you achieve the desired coverage.

To achieve additional coverage to bring your model closer to 100% coverage, modify or add test cases using the Signal Builder block in the harness model, as described in "Creating and Logging Test Cases" on page 11-7.

Executing the Component in Simulation Mode

To verify that the generated code for the model produces the same results as simulating the model, use the Code Generation Verification (CGV) API methods. When you perform this procedure, the simulation compiles and executes the model code using the merged test cases:

1 Create a default options object for `slvnvruncgvtest`:

```
runcgvopts = slvnvruntestopts('cgv');
```

2 Specify to execute the model in simulation mode:

```
runcgvopts.cgvConn = 'sim';
```

3 Execute the `slvnv_powerwindow_controller` model using the two test cases and the `runopts` object:

```
cgvSim = slvnvruncgvtest('slvnvdemo_powerwindow_controller', ...
    mergedTestCases, runcgvopts);
```

These steps save the results in the workspace variable `cgvSim`.

Next, execute the same model with the same test cases in Software-in-the-Loop (SIL) mode and compare the results from both simulations.

For more information about Normal simulation mode, see "Execute the Model".

Executing the Component in Software-in-the-Loop (SIL) Mode

When you execute a model in Software-in-the-Loop (SIL) mode, the simulation compiles and executes the generated code on your host computer.

To execute a model in SIL mode, you must have an Embedded Coder license.

In this section, you execute the slvnvdemo_powerwindow_controller model in SIL mode and compare the results to the previous section, where you executed the model in simulation mode:

1 Specify to execute the model in SIL mode:

```
runcgvopts.cgvConn = 'sil';
```

2 Execute the slvnv_powerwindow_controller model using the merged test cases and the runopts object:

```
cgvSil = slvnvruncgvtest('slvnvdemo_powerwindow_controller', ...
    mergedTestCases, runcgvopts);
```

The workspace variable cgvSil contains the results of the SIL mode execution.

3 Compare the results in cgvSil to the results in to cgvSim (the results from the simulation mode execution). Use the cgv.CGV.compare method to compare the results from the two simulations:

```
for i=1:length(loggedSignalsHarness.TestCases)
    simout = cgvSim.getOutputData(i);
    silout = cgvSil.getOutputData(i);
    [matchNames, ~, mismatchNames, ~ ] = ...
        cgv.CGV.compare(simout, silout);
```

4 Display the results of the comparison in the MATLAB command window:

```
fprintf('\nTest Case(%d):  %d Signals match, ...
        %d Signals mismatch', i, length(matchNames), ...
        length(mismatchNames));
    end
```

11-11

For more information about Software-in-the-Loop (SIL) simulations, see
"What are SIL and PIL Simulations?"

Signal Monitoring with Model Verification Blocks

- Chapter 12, "Using Model Verification Blocks"
- Chapter 13, "Constructing Simulation Tests Using the Verification Manager"
- Chapter 14, "Linking Test Cases to Requirements Documents Using the Verification Manager"

Using Model Verification Blocks

Overview of Model Verification Blocks

Model verification blocks monitor model signals and model characteristics, and check that they remain within specified bounds during simulation.

Simulink "Model Verification" library blocks monitor time-domain signals in your model, according to the specifications that you assign to the blocks.

Note To see a complete description of all Simulink model verification blocks, see the "Model Verification" category in the Simulink Block Reference documentation.

If you have Simulink Control Design™ software, you can also monitor frequency-domain characteristics such as:

- Gain and phase margins
- Peak magnitude

Note For more information about the Simulink Control Design model verification blocks, see "Model Verification" in the Simulink Control Design documentation.

You set a verification block to assert when its signal leaves the limit or range that you specify. During simulation, when the signal crosses the limit, the verification block can:

- Stop the simulation and bring immediate focus to that part of the model.
- Report the limit encounter with a logical signal output of its own. If the simulation does not encounter the limit, the signal output is true. If the simulation encounters the limit, the signal output is false.

Use these blocks with the Verification Manager tool in the Signal Builder to construct simulation tests for your model.

Example: Using the Check Static Lower Bound Block to Check for Out-of-Bounds Signal

The following example uses a Check Static Lower Bound block to stop simulation when a signal from a Sine Wave block crosses its lower bound limit.

1 Attach a Check Static Lower Bound block to the signal from a Sine Wave block.
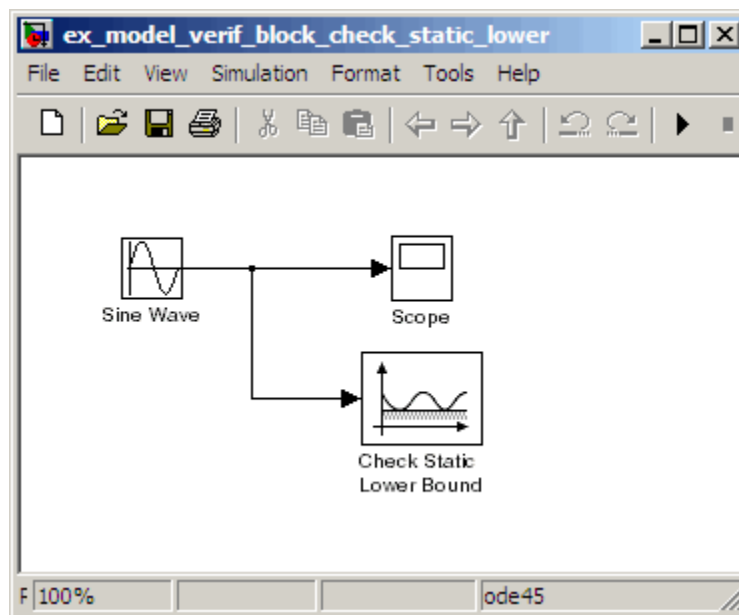
2 Set the Simulation stop time to 2 seconds.

3 Double-click the Sine Wave block and set the following parameters:

- Set the **Amplitude** to 1.

- Set the **Frequency** to pi radians per second.

4 Double-click the Check Static Lower Bound block and set the **Lower bound** parameter to -0.8.

Enable assertion is the default. This parameter enables a verification block for an assertion. You set the Check Static Lower Bound block to detect a signal value of –0.8 or lower. If the signal value reaches that value or falls below it, the simulation stops.

5 Run the simulation.

The model stops simulating after 1.295 seconds, when the output is –0.8. The software highlights the Check Static Lower Bound block.

When the simulation stops, you see the following diagnostic message.
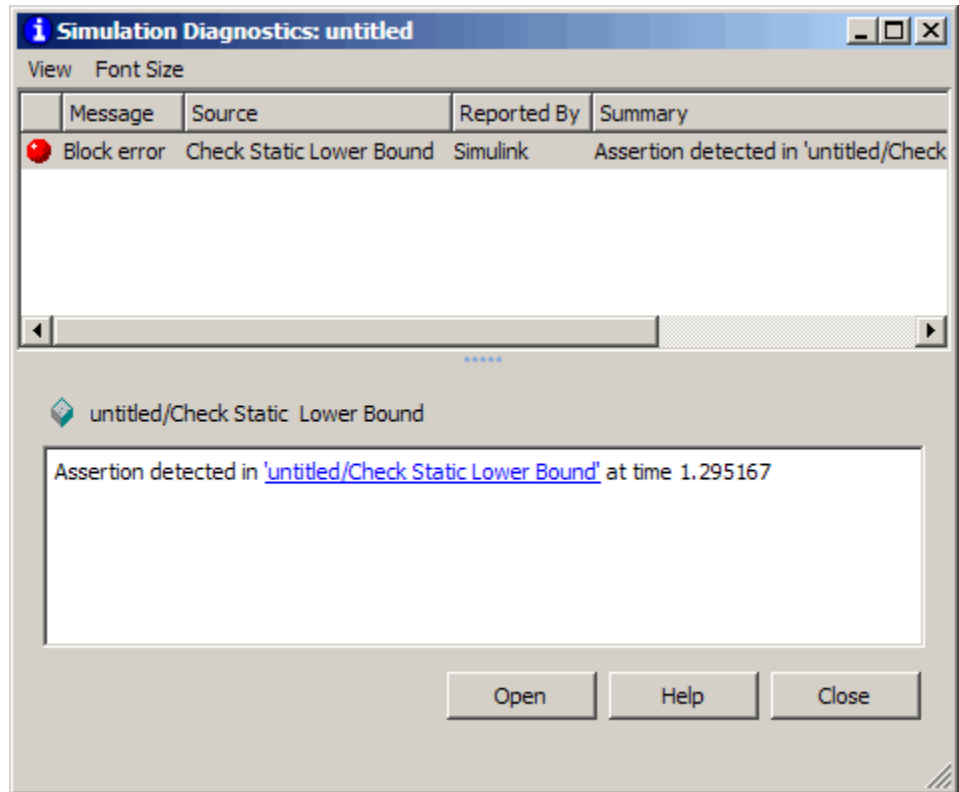
6 To verify the signal value, double-click the Scope block.
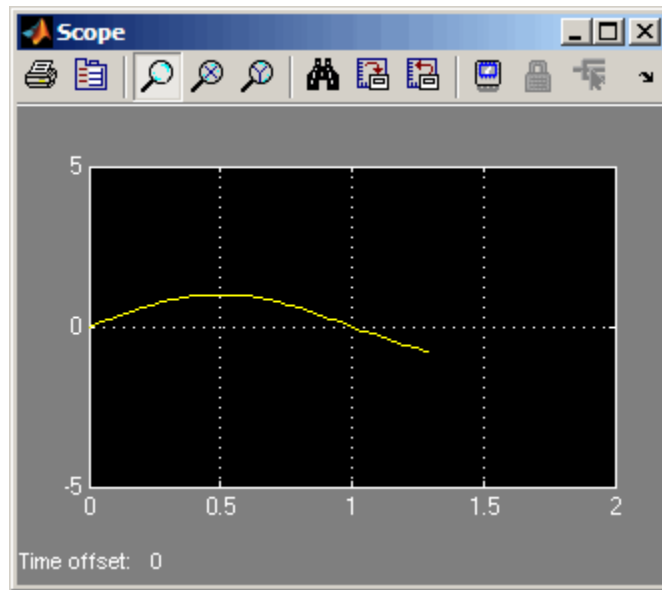
7 To disable the Check Static Lower Bound block from asserting its limit, clear the **Enable assertion** check box.

The block is crossed out in the model, as shown.
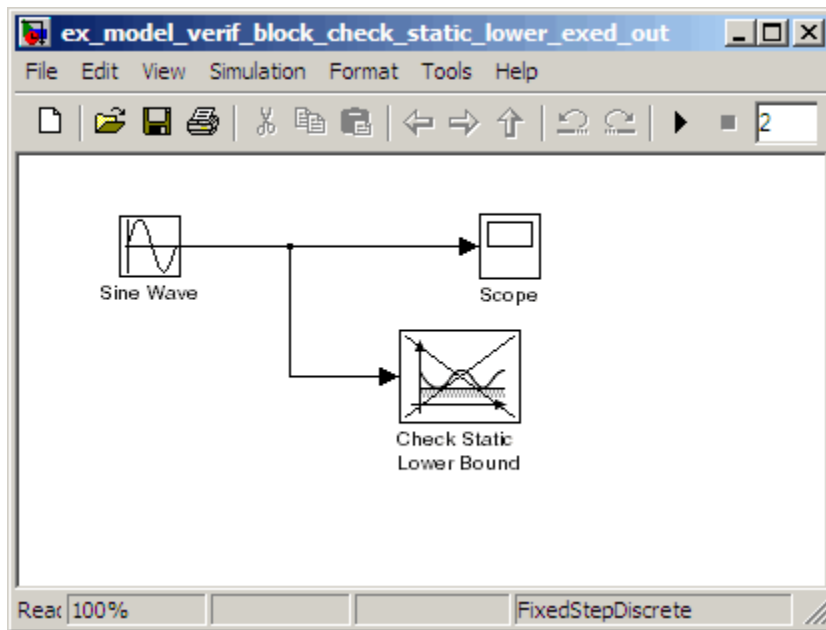
Linear System Modeling Blocks in Simulink Control Design

If you have Simulink Control Design software, using the blocks in the "Model Verification" library, you can:

- Specify bounds on linear system characteristics.

- Check that the bounds are satisfied during simulation.

For example, you can check if the linearized behavior of your model satisfied upper and lower magnitude bounds on a Bode plot or gain and phase margins. For more information, see the individual block reference pages.

Constructing Simulation Tests Using the Verification Manager

What Is the Verification Manager?

The Verification Manager is a graphical interface in the Signal Builder dialog box. Using this tool, you can manage all the Model Verification blocks in your model from a central location.

Opening the Verification Manager

Create a Simulink model that you can use to examine the Verification Manager.

1 In the Simulink software, create the following example model.
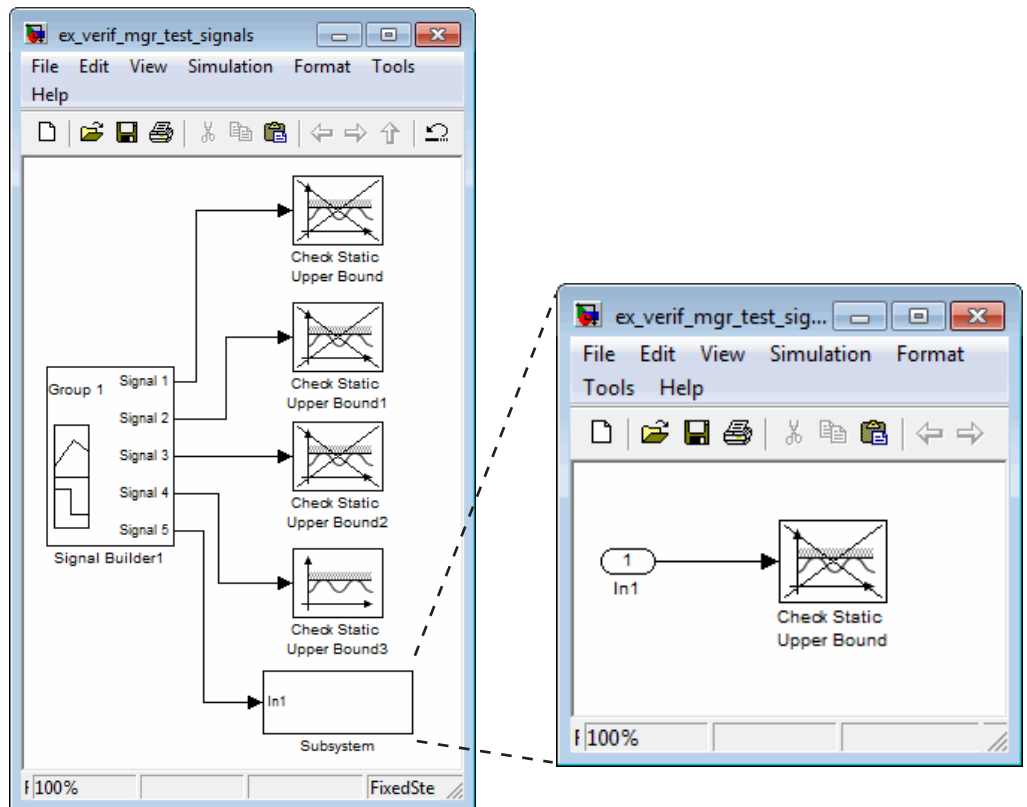
a In the Signal Builder block, create a signal group with five signals in the group.

b Make two copies of the signal group, so that you have three signal groups: **Group 1**, **Group 2**, **Group 3**.

13-3

> **Note** A Signal Builder block provides test signals for an entire model from one location. This model contains a Signal Builder block that feeds five test signals to the Model Verification blocks. The model sends the first four signals directly to Check Static Upper Bound blocks. The model sends the fifth signal to a subsystem that contains a Check Static Upper Bound block.

For more information on the Signal Builder block, see "Working with Signal Groups" in the Simulink documentation.

c To set each Check Static Upper Bound verification block to assert for an upper bound of 1, set the **Upper bound** parameter to 1.

d For the following blocks, disable the assertion by clearing the **Enable assertion** parameter:

- Check Static Upper Bound

- Check Static Upper Bound1

- Check Static Upper Bound2

- Check Static Upper Bound in the subsystem

These blocks are crossed out in the model.

e To enable the Check Static Upper Bound3 block, select the **Enable assertion** parameter.

2 Save this model and name it ex_verif_mgr_test_signals.

3 To open the model Signal Builder dialog box, double-click the Signal Builder block. The signals in the first group (**Group 1** in this example) are displayed.
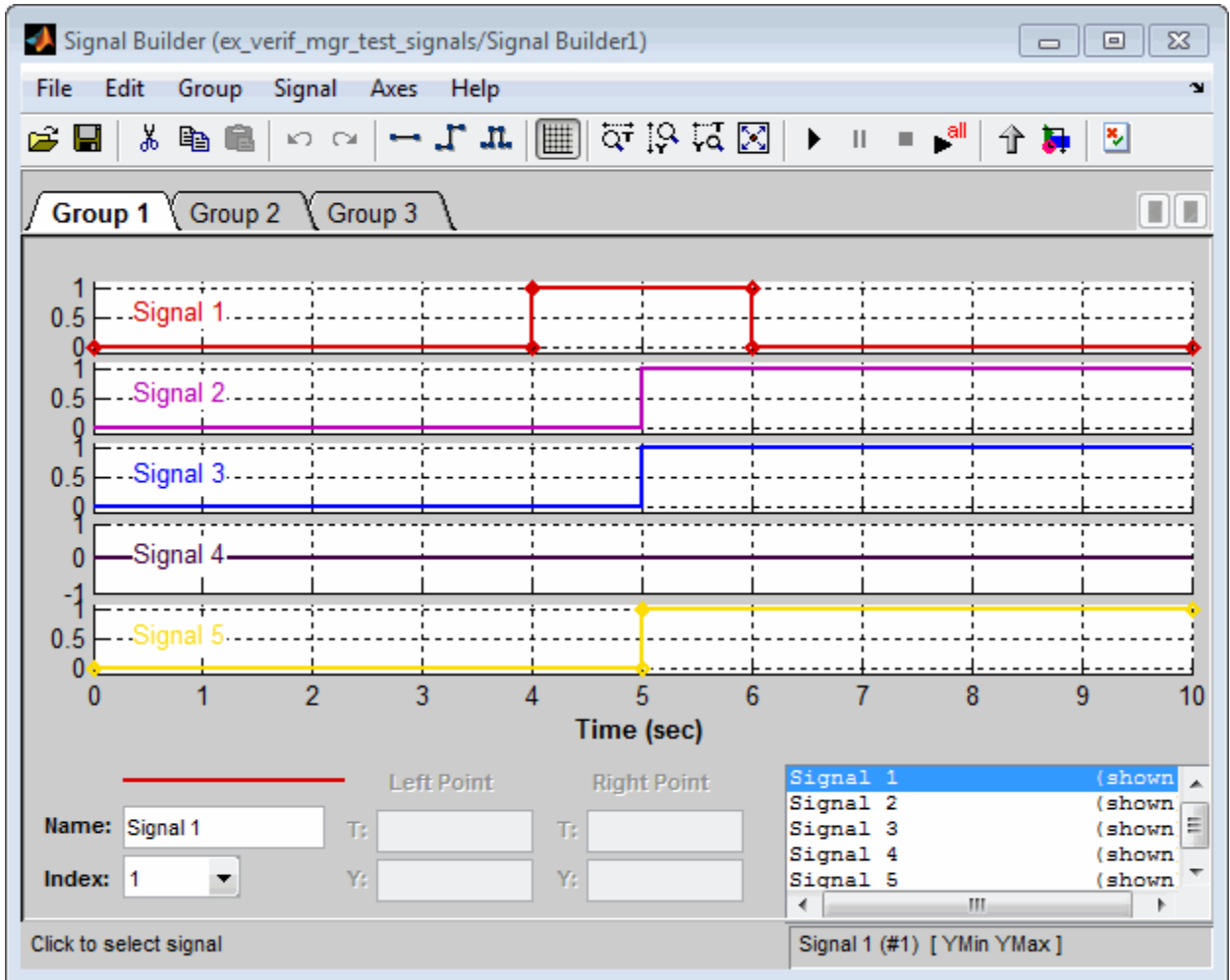
4 On the Signal Builder dialog box toolbar, select the Show Verification Settings tool ![icon].

The **Verification block settings** pane and the **Requirements** pane are displayed.
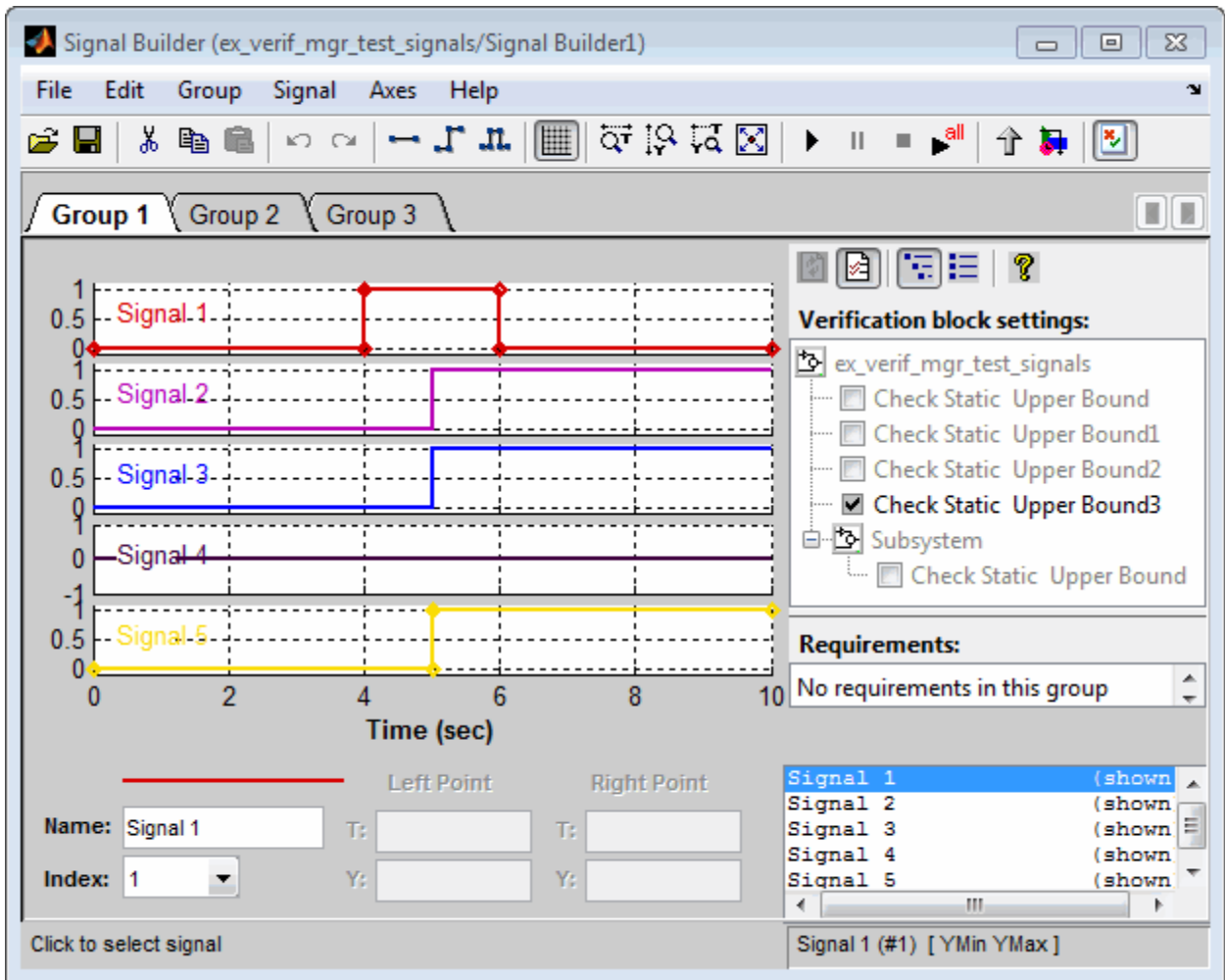
The **Verification block settings** pane lists all Model Verification blocks in the model, grouped by subsystem. If you right-click in this pane, you can select on of three options for viewing Model Verification blocks in this window:

- **Display > Tree format** — If enabled, lists the blocks as they appear in the model hierarchy.

- **Display > Overridden blocks only** — If enabled, lists only the blocks that have been disabled.

- **Display > Active blocks only** — If enabled, lists only the blocks that are enabled.

Note If both **Overridden blocks only** and **Active blocks only** are enabled, no Model Verification blocks appear. If both **Overridden blocks only** and **Active blocks only** are disabled, all Model Verification blocks appear.

In this example, the **Verification block settings** pane displays five Check Static Upper Bound blocks. Four are in the top level of the model, and one is in a subsystem.

The **Requirements** pane lists the requirements document links for the current signal group. For details on adding requirement document links in the Signal Builder dialog box, see Chapter 14, "Linking Test Cases to Requirements Documents Using the Verification Manager".

5 For this example, select 🖻 to close the **Requirements** pane.

6 To display only the enabled Model Verification blocks for the current signal group, in the **Verification block settings** toolbar, select the List Enabled Verifications tool 📋 .

Verification block settings:

☑ Check Static Upper Bound3
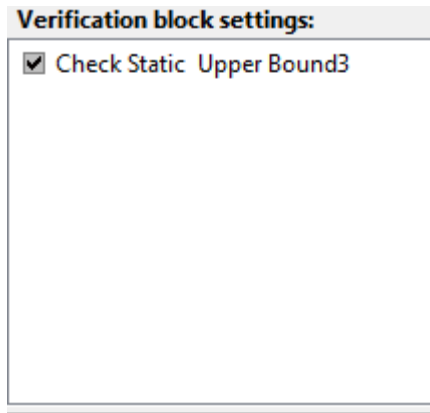
7 To redisplay all Model Verification blocks for the current group, click the Show Verification Block Hierarchy tool .

Enabling and Disabling Model Verification Blocks Using the Verification Manager

Use the Verification Manager to enable and disable individual Model Verification blocks in signal groups. To open the Verification Manager in the Signal Builder dialog box, click .

The **Verification block settings** pane lists the Model Verification blocks in the model. Each verification block has a status node that indicates whether its assertion is enabled or disabled. Each verification block's status node also indicates whether the enabled or disabled setting applies universally or to the active group. The following table describes the different types of status nodes.

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

Use the Verification Manager to enable or disable model verification blocks in the ex_verif_mgr_test_signals model that you created in "Opening the Verification Manager" on page 13-3.

1 In the Verification Manager, click the empty check box next to the Check Static Upper Bound1 node to enable that node for the current group (**Group 1**).

In the **Verification block settings** pane, when you enable a disabled block, you see the following change in how the block is displayed in the model.

Disabled but enabled
in current group (**Group 1)**

Verification Manager — **Group 1**
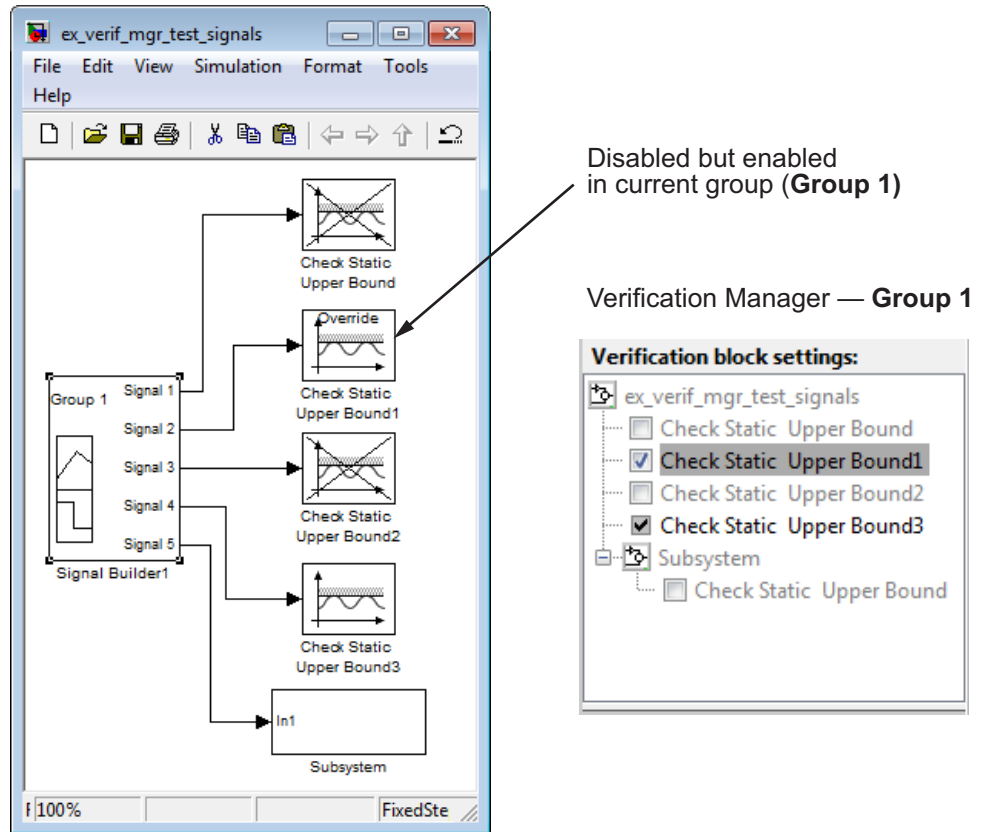
Because you enabled the Check Static Upper Bound1 block in the current
group, an **Override** label is applied to the block and it is no longer crossed
out.

2 In the Signal Builder, click the **Group 2** tab.

3 Select the empty check box next to the Check Static Upper Bound2 node to
enable that block for the current group (**Group 2**).
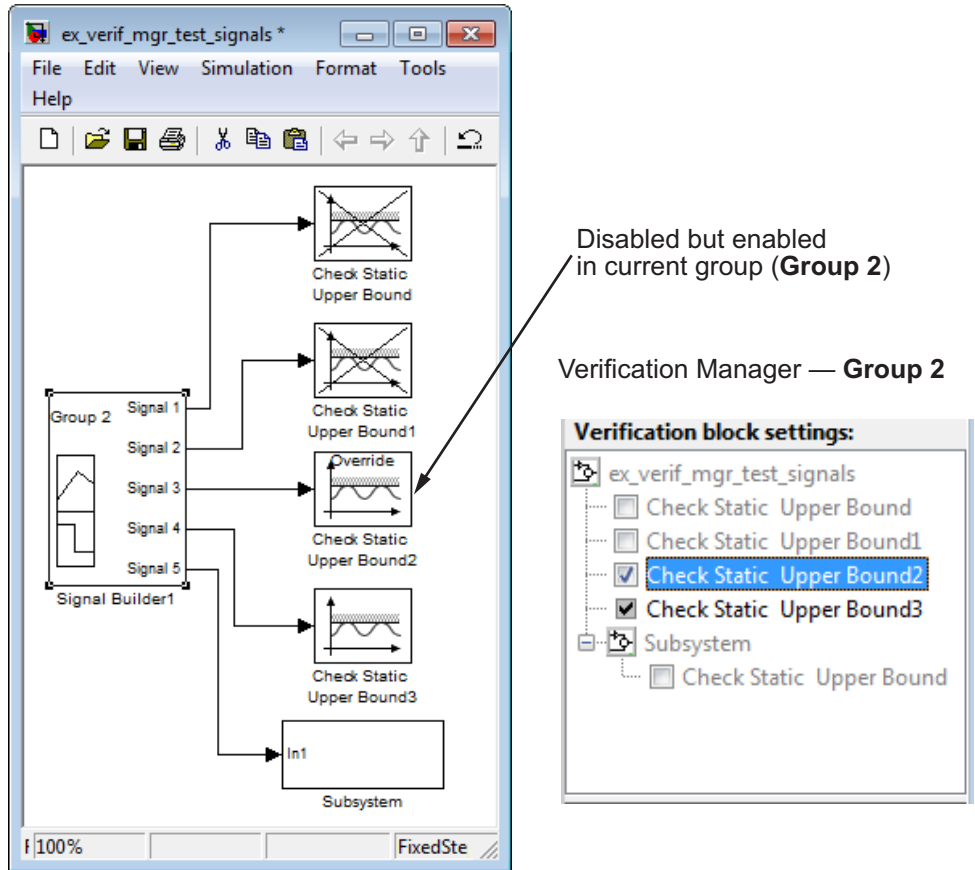
The Check Static Upper Bound2 block is no longer crossed out, indicating that the block is enabled for the current group. However, Check Static Upper Bound1 is crossed out because it is enabled in another group.

4 Save the model with these changes.

Enabling and Disabling Verification Blocks in a Subsystem

If you have a lot of verification blocks, it is tedious to enable and disable blocks individually. Using the Verification Manager, you can enable and disable blocks from context menu options. Depending on the status of the node, you have the following options.

Node Status	Context Menu Options
![icon]	• 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

For example, assume that you define the following groups in the Verification Manager for a model with five Model Verification blocks.

Group 1   Group 2   Group 3
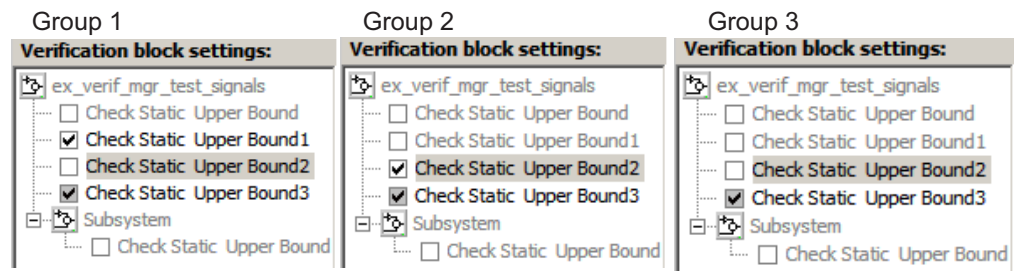
1 In the Verification Manager window, right-click the `ex_verif_mgr_test_signals` node and select **Contents enable for all groups**.

This option enables all verification blocks, for all test groups, in all subsystems; the settings for all groups look as follows:
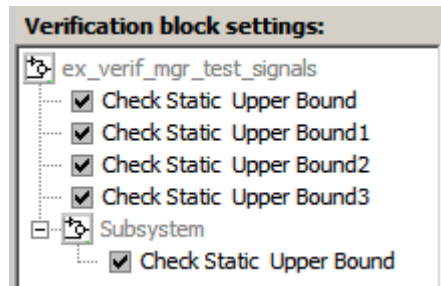
2 Right-click `ex_verif_mgr_test_signals` and select **Contents enable by group**.

This option restores the individually enabled/disabled settings for each verification block in each group.
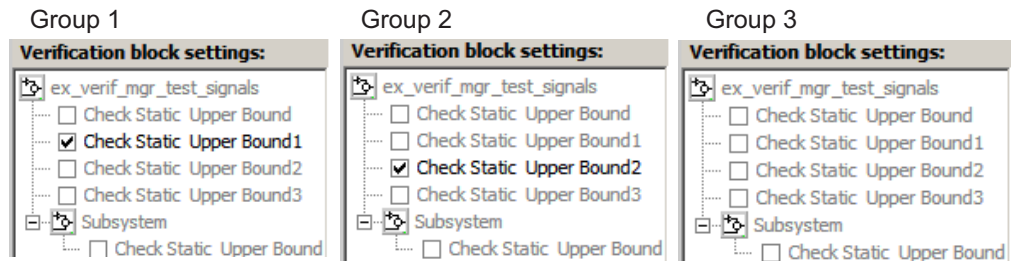
3 Click the **Group 1** tab, right-click `ex_verif_mgr_test_signals`, and select **Contents group enable**.

This option individually enables all contained blocks for only **Group 1**.

Group 1	Group 2 (unchanged)	Group 3 (unchanged)
Verification block settings:	**Verification block settings:**	**Verification block settings:**
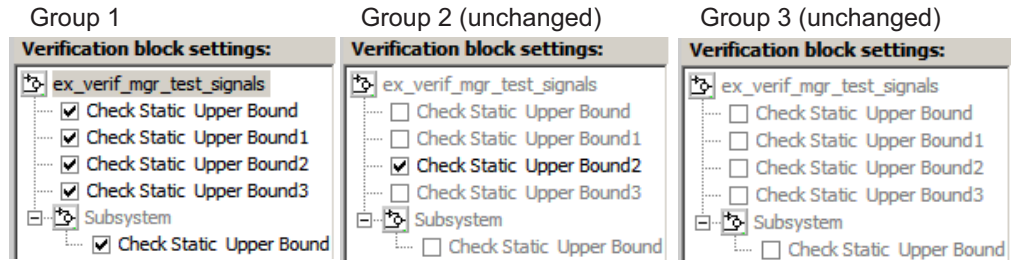
4 Click the **Group 1** tab, right-click ex_verif_mgr_test_signals and select **Contents group disable**.

This option individually disables all contained blocks for only **Group 1**.

Group 1	Group 2 (unchanged)	Group 3 (unchanged)
Verification block settings:	**Verification block settings:**	**Verification block settings:**
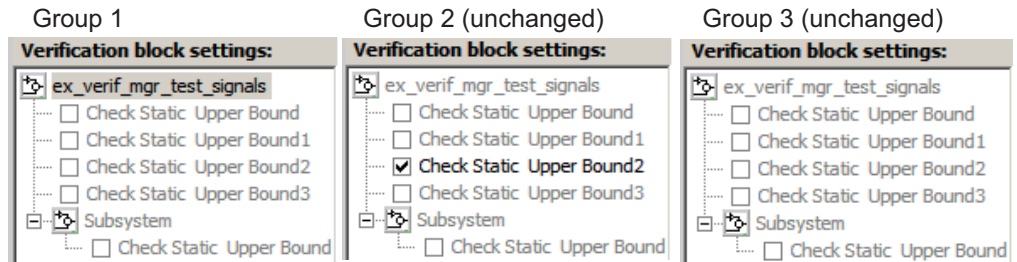
5 Click the **Group 1** tab, right-click the Check Static Upper Bound node, and select **Block enable for all groups**.

This option enables the Check Static Upper Bound block for all groups.

Group 1	Group 2	Group 3
Verification block settings:	**Verification block settings:**	**Verification block settings:**
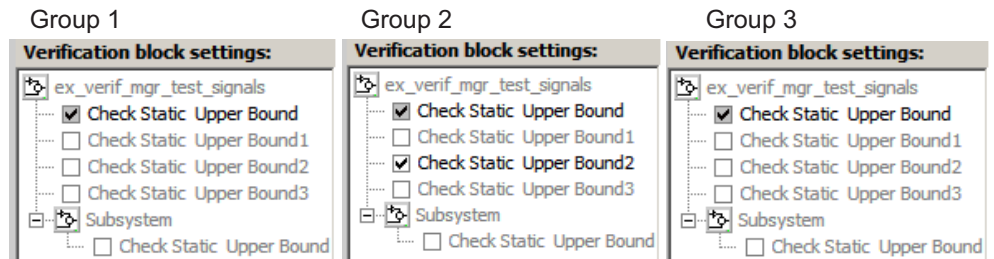
6 Click the **Group 1** tab, right-click the Check Static Upper Bound node, and select **Block enable by group**.

This option restores the individually enabled/disabled state to this block for all groups. The **Block enable by group** option lets you enable or disable this node individually for each group.

Group 1

Verification block settings:

- ex_verif_mgr_test_signals
  - ☐ Check Static Upper Bound
  - ☐ Check Static Upper Bound1
  - ☐ Check Static Upper Bound2
 - ☐ Check Static Upper Bound3
  - ⊟ Subsystem
    - ☐ Check Static Upper Bound

Group 2

Verification block settings:

- ex_verif_mgr_test_signals
  - ☐ Check Static Upper Bound
  - ☐ Check Static Upper Bound1
  - ☑ Check Static Upper Bound2
  - ☐ Check Static Upper Bound3
  - ⊟ Subsystem
    - ☐ Check Static Upper Bound

Group 3

Verification block settings:

- ex_verif_mgr_test_signals
  - ☐ Check Static Upper Bound
  - ☐ Check Static Upper Bound1
  - ☐ Check Static Upper Bound2
  - ☐ Check Static Upper Bound3
  - ⊟ Subsystem
    - ☐ Check Static Upper Bound

7 Click the **Group 1** tab, right-click the Check Static Upper Bound node, and select **Block group enable**.

This option enables the Check Static Upper Bound block for this group only.

Group 1

Verification block settings:

- ex_verif_mgr_test_signals
  - ☑ Check Static Upper Bound
  - ☐ Check Static Upper Bound1
  - ☐ Check Static Upper Bound2
  - ☐ Check Static Upper Bound3
  - ⊟ Subsystem
    - ☐ Check Static Upper Bound

Group 2 (unchanged)

Verification block settings:

- ex_verif_mgr_test_signals
  - ☐ Check Static Upper Bound
  - ☐ Check Static Upper Bound1
  - ☑ Check Static Upper Bound2
  - ☐ Check Static Upper Bound3
  - ⊟ Subsystem
    - ☐ Check Static Upper Bound

Group 3 (unchanged)

Verification block settings:

- ex_verif_mgr_test_signals
  - ☐ Check Static Upper Bound
  - ☐ Check Static Upper Bound1
  - ☐ Check Static Upper Bound2
  - ☐ Check Static Upper Bound3
  - ⊟ Subsystem
    - ☐ Check Static Upper Bound

Selecting **Block group disable** disables the specified block for this group only.

Linking Test Cases to Requirements Documents Using the Verification Manager

Linking Test Cases to Requirements Documents Using the Verification Manager

You can link requirements documents to test cases and their corresponding Model Verification blocks through the Verification Manager **Requirements** pane in the Signal Builder.

1. To display the **Requirements** pane in the Signal Builder dialog box:

   a. Click the **Show verification settings** button ().

   b. Click the **Requirements display** button ().

2. In the **Requirements** pane, right-click anywhere.

3. From the context menu, select **Edit/Add Links**.

   The Requirements dialog box opens.

4. When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

   For information about which setting to use for your working environment, see "Resolving the Document Path" on page 5-15.

5. Add links to requirements documents, as described in "Workflows for Creating Links Using Selection-Based Linking" on page 2-11.

   The names of the linked requirements appear in the **Requirements** pane.
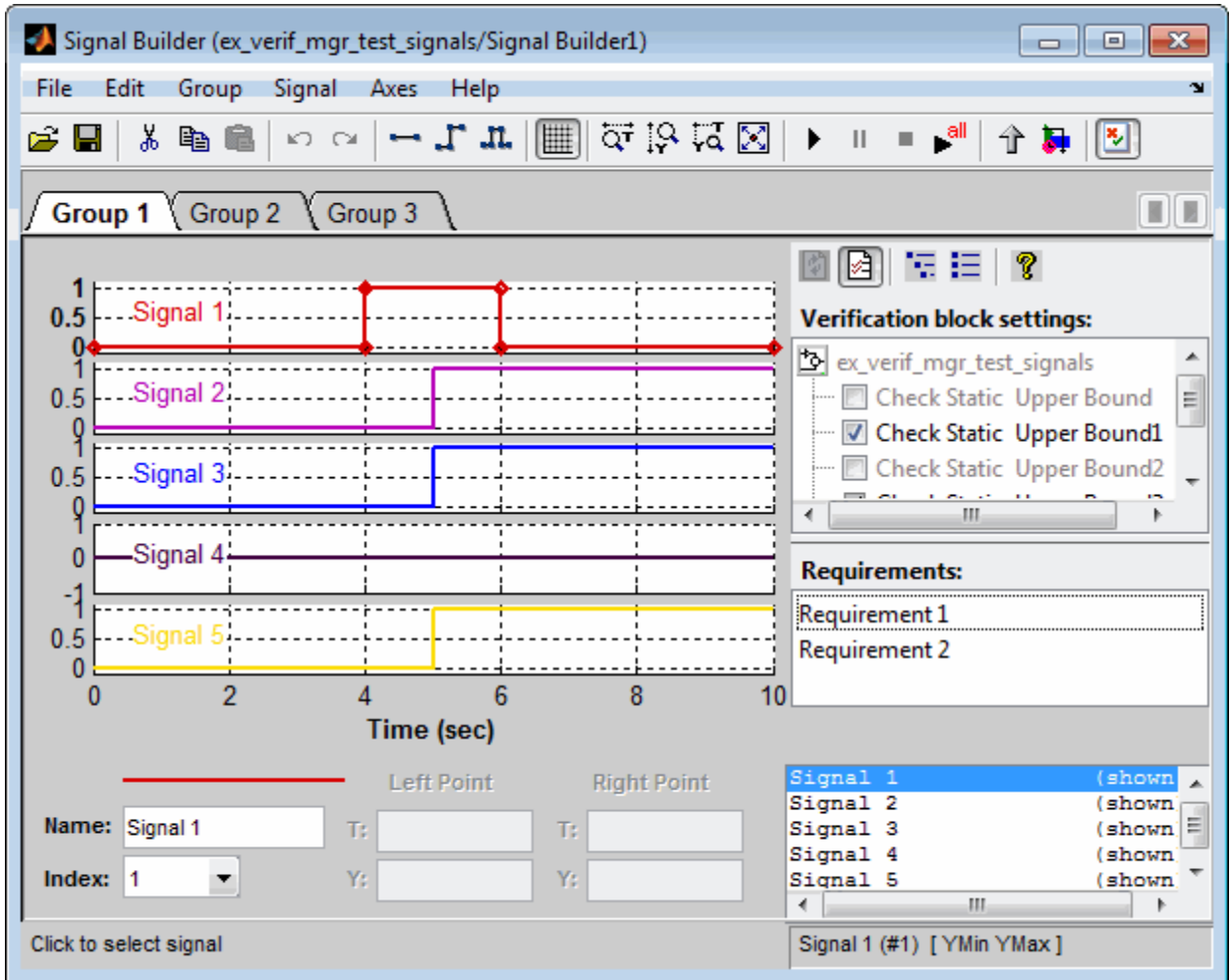
6 To view the requirements document in its native editor, right-click a requirement link and select **View**.

7 Optionally, to delete a requirement link, right-click the link and select **Delete**.

Model Coverage Analysis

Model Coverage Definition

What Is Model Coverage?

Model coverage helps you validate your model tests by measuring how thoroughly the model objects are tested. Model coverage calculates how much a model test case exercises simulation pathways through a model. Model coverage is a measure of how thoroughly a test case tests a model and the percentage of pathways that a test case exercises. Model coverage helps you validate your model tests.

How Model Coverage Works

Model coverage analyzes the execution of the following types of model objects that directly or indirectly determine simulation pathways through your model:

- Simulink blocks
- Models referenced in Model blocks
- The states and transitions of Stateflow charts

During a simulation run, the tool records the behavior of the covered objects, 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 object in the model.

The Simulink Verification and Validation software can only collect coverage for a model if its simulation mode is set to Normal. If the simulation mode is set to any mode other than Normal, coverage will not be measured during simulation.

For the types of coverage that model coverage performs, see "Types of Model Coverage" on page 15-4. For an example of a model coverage report, see Chapter 19, "Results Review".

Types of Model Coverage

Simulink Verification and Validation software can perform several types of coverage analysis:

- "Cyclomatic Complexity" on page 15-4
- "Decision Coverage (DC)" on page 15-5
- "Condition Coverage (CC)" on page 15-5
- "Modified Condition/Decision Coverage (MCDC)" on page 15-5
- "Lookup Table Coverage" on page 15-7
- "Saturate on Integer Overflow Coverage" on page 15-7
- "Signal Range Coverage" on page 15-8
- "Signal Size Coverage" on page 15-8
- "Simulink Design Verifier Coverage" on page 15-9

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 on the generated code due to error checks that the model coverage analysis does not consider.

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

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

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 for atomic subsystems and Stateflow charts.

For an example of cyclomatic complexity data in a model coverage report, see "Cyclomatic Complexity" on page 19-14.

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.

For an example of decision coverage data in a model coverage report, see "Decisions Analyzed" on page 19-16.

Condition Coverage (CC)

Condition coverage analyzes blocks that output the logical combination of their inputs (for example, the Logical Operator block) and Stateflow transitions. A test case achieves full coverage when 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.

When you collect coverage for a model, you may not be able to achieve 100% condition coverage. For example, if you specify to short-circuit logic blocks, by selecting **Treat Simulink Logic blocks as short-circuited** in the Coverage Settings dialog box, you might not be able to achieve 100% condition coverage for that block. See "Treat Simulink logic blocks as short-circuited" on page 17-15 for more information.

For an example of condition coverage data in a model coverage report, see "Conditions Analyzed" on page 19-18.

Modified Condition/Decision Coverage (MCDC)

Modified condition/decision coverage analysis by the Simulink Verification and Validation software extends the decision and condition coverage capabilities. It analyzes blocks that output the logical combination of their inputs 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 when a change in one input, independent of any other inputs, causes a change in the block's output.

- A test case achieves full coverage for a Stateflow transition when there is at least one time when a change in the condition triggers the transition for each condition.

If your model contains blocks that define expressions that have different types of logical operators and more than 12 conditions, the software cannot record MCDC coverage.

Because the Simulink Verification and Validation MCDC coverage may not achieve full decision or condition coverage, you can achieve 100% MCDC coverage *without* achieving 100% decision coverage.

Some Simulink objects support MCDC coverage, some objects support only condition coverage, and some objects support only decision coverage. The table in Chapter 16, "Model Objects That Receive Model Coverage" lists which objects receive which types of model coverage. For example, the Combinatorial Logic block can receive decision coverage and condition coverage, but not MCDC coverage.

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

- Condition Coverage
- Decision Coverage
- MCDC Coverage

When you collect coverage for a model, you may not be able to achieve 100% MCDC coverage. For example, if you specify to short-circuit logic blocks, you may not be able to achieve 100% MCDC coverage for that block.

If you run the test cases independently and accumulate all the coverage results, you can determine if your model adheres to the modified condition and decision coverage standard. For more information about the DO-178B standard, see "DO-178B Checks".

For an example of MCDC coverage data in a model coverage report, see "MCDC Analysis" on page 19-18. For an example of accumulated coverage results, see "Cumulative Coverage" on page 19-20.

Lookup Table Coverage

Lookup table coverage (LUT) examines blocks, such as the 1-D Lookup Table block, that output information from 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 when 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, indicating each interpolation. If the total number of breakpoints of an n-D Lookup Table block exceeds 1,500,000, the software cannot record coverage for that block.

For an example of lookup table coverage data in a model coverage report, see "N-Dimensional Lookup Table" on page 19-23.

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.

Saturate on Integer Overflow Coverage

Saturate on integer overflow coverage examines blocks, such as the Abs block, with the saturate on integer overflow parameter selected. Only blocks with this parameter selected receive saturate on integer overflow coverage.

Saturate on integer overflow coverage records the number of times the block saturates on integer overflow.

A test case achieves full coverage when the blocks saturates on integer overflow at least once and does not saturate at least once.

For an example of saturate on integer overflow coverage data in a model coverage report, see "Saturate on Integer Overflow Analysis" on page 19-31.

Signal Range Coverage

Signal range coverage records the minimum and maximum signal values at each block in the model, as measured during simulation. Only blocks with output signals receive signal range coverage.

If the total number of signals in your model exceeds 65535, or your model contains a signal whose width exceeds 65535, the software cannot record signal range coverage.

For an example of signal range coverage data in a model coverage report, see "Signal Range Analysis" on page 19-33.

Note When you create cumulative coverage for reusable subsystems or Stateflow constructs with single range coverage, the cumulative coverage has the largest possible range of signal values. For more information, see "Creating Cumulative Coverage for Reusable Subsystems and Stateflow Constructs" on page 21-9.

Signal Size Coverage

Signal size coverage records the minimum, maximum, and allocated size for all variable-size signals in a model. Only blocks with variable-size output signals are included in the report.

If the total number of signals in your model exceeds 65535, or your model contains a signal whose width exceeds 65535, the software cannot record signal size coverage.

For an example of signal size coverage data in a model coverage report, see "Signal Size Coverage for Variable-Dimension Signals" on page 19-34.

For more information about variable-size signals, see "Working with Variable-Size Signals".

Simulink Design Verifier Coverage

The Simulink Verification and Validation software collects model coverage data for the following Simulink Design Verifier blocks and MATLAB for code generation functions:

Simulink Design Verifier blocks	MATLAB for code generation functions
Test Condition	`sldv.condition`
Test Objective	`sldv.test`
Proof Assumption	`sldv.assume`
Proof Objective	`sldv.prove`

If you do not have a Simulink Design Verifier license, you can collect model coverage for a model that contains these blocks or functions, but you cannot analyze the model using the Simulink Design Verifier software.

By adding one or more Simulink Design Verifier blocks or functions into your model, you can:

- Check the results of a Simulink Design Verifier analysis, run generated test cases, and use the blocks to observe the results.

- Define model requirements using the Test Objective block and verify the results with model coverage data that the software collected during simulation.

- Analyze the model, create a test harness, and simulate the harness with the Test Objective block to collect model coverage data.

- Analyze the model and use the Proof Assumption block to verify any counterexamples that the Simulink Design Verifier identifies.

If you specify to collect Simulink Design Verifier coverage:

- The software collects coverage for the Simulink Design Verifier blocks and functions.

- The software checks the data type of the signal that links to each Simulink Design Verifier block. If the signal data type is fixed point, the block

parameter must also be fixed point. If the signal data type is not fixed point, the software tries to convert the block parameter data type. If the software cannot convert the block parameter data type, the software reports an error and you must explicitly assign the block parameter data type to match the signal.

- If your model contains a Verification Subsystem block, the software only records coverage for Simulink Design Verifier blocks in the Verification Subsystem block; it does not record coverage for any other blocks in the Verification Subsystem.

If you do not specify to collect Simulink Design Verifier coverage, the software does not check the data types for any Simulink Design Verifier blocks and functions in your model and does not collect coverage.

For an example of coverage data for Simulink Design Verifier blocks or functions in a model coverage report, see "Simulink® Design Verifier™ Coverage" on page 19-36.

Simulink Optimizations and Model Coverage

In the Configuration Parameters dialog box **Optimization** pane, there are three Simulink optimization parameters that can affect your model coverage data:

In this section...
"Inline parameters" on page 15-11
"Block reduction" on page 15-11
"Conditional input branch execution" on page 15-12

Inline parameters

To transform tunable model parameters into constant values for code generation, in the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, select **Inline parameters**. When you enable this option, you cannot change the values of block parameters during simulation.

When the parameters are transformed into constants, Simulink may eliminate certain decisions in your model. You cannot achieve coverage for eliminated decision, so the coverage report displays 0/0 for those decisions.

Block reduction

To achieve faster execution during model simulation and in generated code, in the Configuration Parameters dialog box, on the **Optimization** pane, select the **Block reduction** parameter. The Simulink software collapses certain groups of blocks into a single, more efficient block, or removes them entirely.

One of the model coverage options, **Force block reduction off**, allows you to ignore the **Block reduction** parameter when collecting model coverage.

If you do not select the **Block reduction** parameter, or if you select **Force block reduction off**, the Simulink Verification and Validation software provides coverage data for every block in the model that collects coverage.

If you select the **Block reduction** parameter and do not set **Force block reduction off**, the coverage report lists the reduced blocks that would have collected coverage.

Conditional input branch execution

To improve model execution when the model contains Switch and Multiport Switch blocks, in the Configuration Parameters dialog box, on the **Optimization** pane, select **Conditional input branch execution**. If you select this parameter, the simulation executes only blocks that are required to compute the control input and the data input selected by the control input.

When Conditional input branch execution is enabled, instead of executing all blocks driving the Switch block input ports at each time step, only the blocks required to compute the control input and the data input selected by the control input execute.

Several considerations affect or limit Switch block optimization:

- Only blocks with `-1` (inherited) or `inf` (Constant) sample time can be optimized.

- Blocks with outputs flagged as test points cannot be optimized.

- Multirate blocks cannot be optimized.

- Blocks with states cannot be optimized.

- Only S-functions with the `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` option enabled can be optimized.

For example, if your model has a Switch block with output flagged as a test point, such as when a Scope block is attached, that Switch block is not executed, and the model coverage data is incomplete. If you have a model with Switch blocks and you want to verify that the model coverage data is complete, clear **Conditional input branch execution**.

Model Objects That Receive Model Coverage

Model Objects That Receive Coverage

Certain Simulink objects can receive any type of model coverage. Other Simulink objects can receive only certain types of coverage, as the following table shows. Click a link in the first column to get more detailed information about coverage for a specific model objects.

For Stateflow states, events, and state temporal logic decisions, model coverage provides only decision coverage. For Stateflow transitions, model coverage provides decision, condition, and MCDC coverage. For more information, see "Model Coverage for Stateflow Charts" on page 18-40.

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier
"Abs" on page 16-8	●				
"Bias" on page 16-9	●				
"Data Type Conversion" on page 16-11	●				
"Combinatorial Logic" on page 16-10	●	●			
"Dead Zone" on page 16-12	●				
"Direct Lookup Table (n-D)" on page 16-14				●	
"Discrete Filter" on page 16-15	●				
"Discrete FIR Filter" on page 16-16	●				
"Discrete-Time Integrator" on page 16-17 (when saturation limits are enabled or reset)	●				
"Discrete Transfer Fcn" on page 16-19	●				
"Dot Product" on page 16-20	●				

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier
"Enabled Subsystem" on page 16-21	●	●	●		
"Enabled and Triggered Subsystem" on page 16-22	●	●	●		
"Fcn" on page 16-24 (Boolean operators only)		●	●		
"For Iterator, For Iterator Subsystem" on page 16-25	●				
"Gain " on page 16-26	●				
"If, If Action Subsystem" on page 16-27	●	●	●		
"Interpolation Using Prelookup" on page 16-28	●			●	
"Library-Linked Objects" on page 16-29	●	●	●	●	●
"Logical Operator" on page 16-30		●	●		
"1-D Lookup Table" on page 16-31	●			●	
"2-D Lookup Table" on page 16-32	●			●	
"n-D Lookup Table" on page 16-33	●			●	
"Math Function" on page 16-34	●				
"MATLAB Function" on page 16-35	●	●	●		
"MinMax" on page 16-36	●				

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier
"Model" on page 16-37 See also "Triggered Models" on page 16-54.	●	●	●	●	●
"Multiport Switch" on page 16-38	●				
"PID Controller, PID Controller (2 DOF)" on page 16-39	●				
"Product" on page 16-40	●				
"Proof Assumption" on page 16-41					●
"Proof Objective" on page 16-42					●
"Rate Limiter" on page 16-43	● (Relative to slew rates)				
"Relay" on page 16-44	●				
"Saturation" on page 16-45	●				
"Saturation Dynamic" on page 16-46	●				
"Simulink® Design Verifier™ Functions in MATLAB Function Blocks" on page 16-47					●
Stateflow charts	●	●	●		
"Sqrt, Signed Sqrt, Reciprocal Sqrt" on page 16-48	●				
"Sum, Add, Subtract, Sum of Elements " on page 16-49	●				

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier
"Switch" on page 16-50	●				
"SwitchCase, SwitchCase Action Subsystem" on page 16-51	●				
"Test Condition" on page 16-52					●
"Test Objective" on page 16-53					●
"Triggered Models" on page 16-54	●	●	●		
"Triggered Subsystem" on page 16-55	●	●	●		
"Truth Table" on page 16-56	●	●	●		
"Unary Minus" on page 16-57	●				
"Weighted Sample Time Math" on page 16-58	●				
"While Iterator, While Iterator Subsystem" on page 16-59	●				

Abs

The Abs block receives decision coverage. Decision coverage is based on:

- Input to the block being less than zero.
- Selection of the **Saturate on integer overflow** parameter.
- Data type of the input signal.

For input to the block being less than zero, the decision coverage measures:

- The number of time steps that the block input is less than zero, indicating a true decision.
- The number of time steps the block input is not less than zero, indicating a false decision.

If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.
- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

If the input data type to the Abs block is `uint8`, `uint16`, or `uint32`, the Simulink Verification and Validation software reports no coverage for the block. The software sets the block output equal to the block input without making any decision. If the input data type to the Abs block is Boolean, an error occurs.

Bias

The Bias block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Combinatorial Logic

The Combinatorial Logic block receives decision and condition coverage. Decision coverage is based on achieving each output row of the truth table. The decision coverage measures the number of time steps that each output row of the truth table is set to the block output.

The condition coverage measures the number of time steps that each input is false (equal to zero) and the number of times each input is true (not equal to zero). If the Combinatorial Logic block has a single input element, the Simulink Verification and Validation software reports only decision coverage, because decision and condition coverage are equivalent.

If all truth table values are set to the block output for at least one time step, decision coverage is 100%. Otherwise, the software reports the coverage as the number of truth table values output during at least one time step, divided by the total number of truth table values. Because this block always has at least one value in the truth table as output, the minimum coverage reported is one divided by the total number of truth table values.

If all block inputs are false for at least one time step and true for at least one time step, condition coverage is 100%. Otherwise, the software reports the coverage as achieving a false value at each input for at least one time step, plus achieving a true value for at least one time step, divided by two raised to the power of the total number of inputs (i.e., $2 \wedge number\_of\_inputs$). The minimum coverage reported is the total number of inputs divided by two raised to the power of the total number of inputs.

Data Type Conversion

The Data Type Conversion block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Dead Zone

The Dead Zone block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for these parameters:

- **Start of dead zone**
- **End of dead zone**
- **Saturate on integer overflow**

The **Start of dead zone** parameter specifies the lower limit of the dead zone. For the **Start of dead zone** parameter, decision coverage measures:

- The number of time steps that the block input is greater than or equal to the lower limit, indicating a true decision.
- The number of time steps that the block input is less than the lower limit, indicating a false decision.

The **End of dead zone** parameter specifies the upper limit of the dead zone. For the **End of dead zone**, decision coverage measures:

- The number of time steps that the block input is greater than the upper limit, indicating a true decision.
- The number of time steps that the block input is less than or equal to the upper limit, indicating a false decision.

When the upper limit is true, the software does not measure **Start of dead zone** coverage for that time step. Therefore, the total number of **Start of dead zone** decisions equals the number of time steps that the **End of dead zone** is false.

If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for each of the individual decisions for the Dead Zone block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Direct Lookup Table (n-D)

The Direct Lookup Table (n-D) block receives lookup table coverage. For an *n*-dimensional lookup table, the number of output break points is the product of all the number of break points for each table dimension.

Lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.

- The number of times during simulation that each combination of dimension input values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for any *n*-dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

Discrete Filter

The Discrete Filter block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Discrete FIR Filter

The Discrete FIR Filter block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Discrete-Time Integrator

The Discrete-Time Integrator block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for these parameters:

- **External reset**
- **Limit output**
- **Saturate on integer overflow**

If you set **External reset** to `none`, the Simulink Verification and Validation software does not report decision coverage for the reset decision. Otherwise, the decision coverage measures:

- The number of time steps that the block output is reset, indicating a true decision.
- The number of time steps that the block output is not reset, indicating a false decision.

If you do not select **Limit output**, the software does not report decision coverage for that decision. Otherwise, the software reports decision coverage for the **Lower saturation limit** and the **Upper saturation limit**.

For the **Upper saturation limit**, decision coverage measures:

- The number of time steps that the integration result is greater than or equal to the upper limit, indicating a true decision.
- The number of time steps that the integration result is less than the upper limit, indicating a false decision.

For the **Lower saturation limit**, decision coverage measures

- The number of time steps that the integration result is less than or equal to the lower limit, indicating a true decision.
- The number of time steps that the integration result is greater than the lower limit, indicating a false decision.

For a time step when the upper limit is true, the software does not measure **Lower saturation limit** coverage. Therefore, the total number of lower limit decisions equals the number of time steps that the upper limit is false.

If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.
- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for each of the individual decisions for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Discrete Transfer Fcn

The Discrete Transfer Fcn block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Dot Product

The Dot Product block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.
- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Enabled Subsystem

The Enabled Subsystem block receives decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the block is enabled, indicating a true decision.
- The number of time steps that the block is disabled, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The Simulink Verification and Validation software measures condition coverage for the enable input only if the enable input is a vector. For the enable input, condition coverage measures the number of time steps each element of the enable input is true and the number of time steps each element of the enable input is false. The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

The software measures MCDC coverage for the enable input only if the enable input is a vector. Because the enable of the subsystem is an OR of the vector inputs, MCDC coverage is 100% if, during at least one time step, each vector enable input is exclusively true and if, during at least one time step, all vector enable inputs are false. For MCDC coverage measurement, the software treats each element of the vector as a separate condition.

Enabled and Triggered Subsystem

The Enabled and Triggered Subsystem block receives decision, condition, and MCDC coverage. Decision coverage measures:

- The number of time steps that a trigger edge occurs while the block is enabled, indicating a true decision.

- The number of time steps that a trigger edge does not occur while the block is enabled, or the block is disabled, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The software measures condition coverage for the enable input and for the trigger input separately:

- For the enable input, condition coverage measures the number of time steps the enable input is true and the number of time steps the enable input is false.

- For the trigger input, condition coverage measures the number of time steps the trigger edge occurs, indicating true, and the number of time steps the trigger edge does not occur, indicating false.

The software reports condition coverage based on the total number of possible conditions and how many conditions are true for at least one time step and how many are false for at least one time step. The software treats each element of a vector as a separate condition coverage measurement.

The software measures MCDC coverage for the enable input and for the trigger input in combination. Because the enable input of the subsystem is an AND of these two inputs, MCDC coverage is 100% if all of the following occur:

- During at least one time step, both inputs are true.

- During at least one time step, the enable input is true and the trigger edge is false.

- During one time step, the enable input is false and the trigger edge is true.

The software treats each vector element as a separate MCDC coverage measurement. It measures each trigger edge element against each enable input element. However, if the number of elements in both the trigger and enable inputs exceeds 12, the software does not report MCDC coverage.

Fcn

The Fcn block receives condition and MCDC coverage. The Simulink Verification and Validation software reports condition or MCDC coverage for Fcn blocks only if the top-level operator is Boolean (&&, ||, or !).

Condition coverage is based on input values or arithmetic expressions that are inputs to Boolean operators in the block. The condition coverage measures:

- The number of time steps that each input to a Boolean operator is true (not equal to zero).

- The number of time steps that each input to a Boolean operator is false (equal to zero).

If all Boolean operator inputs are false for at least one time step and true for at least one time step, condition coverage is 100%. Otherwise, the software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

The software measures MCDC coverage for Boolean expressions within the Fcn block. If, during at least one time step, each condition independently sets the output of the expression to true and if, during at least one time step, each condition independently sets the output of the expression to false, MCDC coverage is 100%. Otherwise, the software reports MCDC coverage based on the total number of possible conditions and how many times each condition independently sets the output to true during at least one time step and how many conditions independently set the output to false during at least one time step.

For Iterator, For Iterator Subsystem

The For Iterator block and For Iterator Subsystem receive decision coverage. The Simulink Verification and Validation software measures decision coverage for the loop condition value, which is determined by one of the following:

- The iteration value being at or below the iteration limit, indicated as true.

- The iteration value being above the iteration limit, indicated as false.

The software reports the total number of times that each loop condition evaluates to true and to false. If the loop condition evaluates to true at least once and false at least once, decision coverage is 100%. If no loop conditions are true, or if no loop conditions are false, decision coverage is 50%.

Gain

The Gain block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

If, If Action Subsystem

The If block that causes an If Action Subsystem to execute receives condition, decision, and MCDC coverage:

- The software measures decision coverage for the `if` condition and all `elseif` conditions defined in the If block.

- If the `if` condition or any of the `elseif` conditions contains a logical expression with multiple conditions, such as `u1 & u2 & u3`, the software also measures condition and MCDC coverage for each condition in the expression, `u1`, `u2`, and `u3` in the preceding example.

The software does not directly measure the `else` condition. When there are no `elseif` conditions, the `else` condition is the direct complement of the `if` condition, or the `else` condition is the direct complement of the last `elseif` condition.

The software reports the total number of time steps that each `if` and `elseif` condition evaluates to true and to false. If the `if` or `elseif` condition evaluates to true at least once, and evaluates to false at least once, decision coverage is 100%. If no `if` or `elseif` conditions are true, or if no `if` or `elseif` conditions are false, decision coverage is 50%. If the previous `if` or `elseif` condition never evaluates as false, an `elseif` condition can have 0% decision coverage.

Interpolation Using Prelookup

The Interpolation Using Prelookup block receives lookup table coverage. For an n-D lookup table, the number of output break points equals the product of all the number of break points for each table dimension. The lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.

- The number of times during simulation that each combination of dimension input values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for any n-dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

The Interpolation Using Prelookup block also receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Library-Linked Objects

Simulink blocks and Stateflow charts that are linked to library objects receive the same coverage that they would receive if they were not linked to library objects. The Simulink Verification and Validation software records coverage individually for each library object in the model. If your model contains multiple instances of the same library object, each instance receives its own coverage data.

Logical Operator

The Logical Operator block receives condition and MCDC coverage. The Simulink Verification and Validation software measures condition coverage for each input to the block. The condition coverage measures:

- The number of time steps that each input is true (not equal to zero).

- The number of time steps that each input is false (equal to zero).

If all block inputs are false for at least one time step and true for at least one time step, the software condition coverage is 100%. Otherwise, the software reports the condition coverage based on the total number of possible conditions and how many are true at least one time step and how many are false at least one time step.

The software measures MCDC coverage for all inputs to the block. If, during at least one time step, each condition independently sets the output of the block to true and if, during at least one time step, each condition independently sets the output of the block to false, MCDC coverage is 100%. Otherwise, the software reports the MCDC coverage based on the total number of possible conditions and how many times each one of them independently set the output to true for at least one time step and how many independently set the output to false for at least one time step.

1-D Lookup Table

The 1-D Lookup Table block receives lookup table coverage; for a one-dimensional lookup table, the number of input and output break points is equal. Lookup table coverage measures:

- The number of times during simulation that the input and output values are between each of the break points.

- The number of times during simulation that the input and output values are below the lowest break point and above the highest break point.

The total number of coverage points for any one-dimensional lookup table is the number of break points in the table plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

The 1-D Lookup Table block also receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

2-D Lookup Table

The 2-D Lookup Table block receives lookup table coverage. For a two-dimensional lookup table, the number of output break points equals the number of row break points multiplied by the number of column inputs. Lookup table coverage measures:

- The number of times during simulation that each combination of row input and column input values is between each of the break points.

- The number of times during simulation that each combination of row input and column input values is below the lowest break point and above the highest break point for each row and column.

The total number of coverage points for any two-dimensional lookup table is the number of row break points in the table plus one, multiplied by the number of column break points in the table plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The 2-D Lookup Table block also receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

n-D Lookup Table

The n-D Lookup Table block receives lookup table coverage. For an *n*-dimensional lookup table, the number of output break points equals the product of all the number of break points for each table dimension. Lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.
- The number of times during simulation that each combination of dimension output values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for any *n*-dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

The n-D Lookup Table block also receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.
- The number of times the block does not saturate on integer overflow, indicating a false decision.

Math Function

The Math Function block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

MATLAB Function

For information about the type of coverage that the Simulink Verification and Validation software reports for the MATLAB Function block, see "Model Coverage for MATLAB Functions" on page 18-20.

MinMax

The MinMax block receives decision coverage based on:

- Passing each input to the output of the block.

- Selection of the **Saturate on integer overflow** parameter.

For decision coverage based on passing each input to the output of the block, the coverage measures the number of time steps that the simulation passes each input to the block output. The number of decision points is based on the number of inputs to the block and whether they are scalar, vector, or matrix.

If all inputs are passed to the block output for at least one time step, the Simulink Verification and Validation software reports the decision coverage as 100%. Otherwise, the software reports the coverage as the number of inputs passed to the output during at least one time step, divided by the total number of inputs.

If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Model

The Model block does not receive coverage directly; the model that the block references receives coverage. If the simulation mode for the referenced model is set to Normal, the Simulink Verification and Validation software reports coverage for all objects within the referenced model that receive coverage. If the simulation mode is set to any value other than Normal, the software cannot measure coverage for the referenced model.

In the Coverage Settings dialog box, select the referenced models for which you want to report coverage. The software generates a coverage report for each referenced model you select.

If your model contains multiple instances of the same referenced model, the software records coverage for all instances of that model where the simulation mode of the Model block is set to Normal. The coverage report for that referenced model combines the coverage data for all Normal mode instances of that model.

The coverage reports for referenced models are linked from a summary report for the parent model.

Note For details on how to select referenced models to report coverage, see "Coverage for referenced models" on page 17-5.

Multiport Switch

The Multiport Switch block receives decision coverage based on:

- Passing each input, excluding the first control input, to the output of the block.

- Selection of the **Saturate on integer overflow** parameter.

For decision coverage based on passing each input, excluding the first control input, to the output of the block, the coverage measures the number of time steps that each input is passed to the block output. The number of decision points is based on the number of inputs to the block and whether the control input is scalar or vector.

If all inputs, excluding the first control input, are passed to the block output for at least one time step, decision coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of inputs passed to the output during at least one time step, divided by the total number of inputs minus one.

If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

PID Controller, PID Controller (2 DOF)

The PID Controller and PID Controller (2 DOF) blocks receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Product

The Product block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Proof Assumption

The Proof Assumption block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Proof Assumption block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Proof Objective

The Proof Objective block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Proof Objective block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Rate Limiter

The Rate Limiter block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for the **Rising slew rate** and **Falling slew rate** parameters.

For the **Rising slew rate**, decision coverage measures:

- The number of time steps that the block input changes more than or equal to the rising rate, indicating a true decision.
- The number of time steps that the block input changes less than the rising rate, indicating a false decision.

For the **Falling slew rate**, decision coverage measures:

- The number of time steps that the block input changes less than or equal to the falling rate, indicating a true decision.
- The number of time steps that the block input changes more than the falling rate, indicating a false decision.

The software does not measure **Falling slew rate** coverage for a time step when the **Rising slew rate** is true. Therefore, the total number of **Falling slew rate** decisions equals the number of time steps that the **Rising slew rate** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Relay

The Relay block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for the **Switch on point** and the **Switch off point** parameters.

For the **Switch on point**, decision coverage measures:

- The number of consecutive time steps that the block input is greater than or equal to the **Switch on point**, indicating a true decision.
- The number of consecutive time steps that the block input is less than the **Switch on point**, indicating a false decision.

For the **Switch off point**, decision coverage measures:

- The number of consecutive time steps that the block input is less than or equal to the **Switch off point**, indicating a true decision.
- The number of consecutive time steps that the block input is greater than the **Switch off point**, indicating a false decision.

The software does not measure **Switch off point** coverage for a time step when the switch on threshold is true. Therefore, the total number of **Switch off point** decisions equals the number of time steps that the **Switch on point** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Saturation

The Saturation block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for the **Lower limit** and **Upper limit** parameters.

For the **Upper limit**, decision coverage measures:

- The number of time steps that the block input is greater than or equal to the upper limit, indicating a true decision.
- The number of time steps that the block input is less than the upper limit, indicating a false decision.

For the **Lower limit**, decision coverage measures:

- The number of time steps that the block input is greater than the lower limit, indicating a true decision.
- The number of time steps that the block input is less than or equal to the lower limit, indicating a false decision.

The software does not measure **Lower limit** coverage for a time step when the upper limit is true. Therefore, the total number of **Lower limit** decisions equals the number of time steps that the **Upper limit** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the Saturation block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Saturation Dynamic

The Saturation Dynamic block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Simulink Design Verifier Functions in MATLAB Function Blocks

The following functions in MATLAB Function blocks receive Simulink Design Verifier coverage:

- `sldv.condition`

- `sldv.test`

- `sldv.assume`

- `sldv.prove`

Each of these functions evaluates an expression *expr*, for example, `sldv.test(`*expr*`)`, where *expr* is any valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to true.

If *expr* is true for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Verification and Validation software reports coverage for that function as 0%.

Sqrt, Signed Sqrt, Reciprocal Sqrt

The Sqrt, Signed Sqrt, Reciprocal Sqrt blocks receive decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Sum, Add, Subtract, Sum of Elements

The Sum, Add, Subtract, Sum of Elements blocks receive decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Switch

The Switch block receives decision coverage. Decision coverage is based on:

- Control input to block
- Selection of the **Saturate on integer overflow** parameter

For the control input to the block, decision coverage measures:

- The number of time steps that the control input evaluates to true.
- The number of time steps the control input evaluates to false.

The number of decision points is based on whether the control input is scalar or vector.

If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.
- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for each of the individual decisions for the Switch block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

SwitchCase, SwitchCase Action Subsystem

The SwitchCase block and SwitchCase Action Subsystem receive decision coverage. The Simulink Verification and Validation software measures decision coverage individually for each switch case defined in the block and also for the default case. The number of decision outcomes is equal to the number of case conditions plus one for the `default` case, if one is defined.

The software reports the total number of time steps that each case evaluates to true. If each case, including the default case, evaluates to true at least once, decision coverage is 100%. The software determines the decision coverage by the number of cases that evaluate true for at least one time step divided by the total number of cases.

If the SwitchCase block does not contain a `default` case, the software measures decision coverage for the number of time steps none of the cases evaluated to true. In the coverage report, this coverage is reported as **implicit-default**.

Test Condition

The Test Condition block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Test Condition block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Test Objective

The Test Objective block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Test Objective block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Triggered Models

A Model block can reference a model that contains edge-based trigger ports at the root level of the model. Triggered models receive decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the referenced model is triggered, indicating a true decision.
- The number of time steps that the referenced model is not triggered, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the Model block that references the triggered model is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

Only if the trigger input is a vector, the Simulink Verification and Validation software measures condition coverage for the trigger port in the referenced model. For the trigger port, condition coverage measures:

- The number of time steps that each element of the trigger port is true.
- The number of time steps that each element of the trigger port is false.

The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

If the trigger port is a vector, the software measures MCDC coverage for the trigger port only. Because the trigger port of the referenced model is an OR of the vector inputs, if, during at least one time step, each vector trigger port is exclusively true and if, during at least one time step, all vector trigger port inputs are false, MCDC coverage is 100%. The software treats each element of the vector as a separate condition for MCDC coverage measurement.

Triggered Subsystem

The Triggered Subsystem block receives decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the block is triggered, indicating a true decision.
- The number of time steps that the block is not triggered, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The Simulink Verification and Validation software measures condition coverage for the trigger input only if the trigger input is a vector. For the trigger input, condition coverage measures:

- The number of time steps that each element of the trigger edge is true.
- The number of time steps that each element of the trigger edge is false.

The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

If the trigger input is a vector, the software measures MCDC coverage for the trigger input only. Because the trigger edge of the subsystem is an OR of the vector inputs, if, during at least one time step, each vector trigger edge input is exclusively true and if, during at least one time step, all vector trigger edge inputs are false, MCDC coverage is 100%. The software treats each element of the vector as a separate condition for MCDC coverage measurement.

Truth Table

The Truth Table block is a Stateflow block that enables you to use truth table logic directly in a Simulink model. The Truth Table block receives condition, decision, and MCDC coverage. For more information on model coverage with Stateflow truth tables, see "Model Coverage for Stateflow Truth Tables" on page 18-61.

Unary Minus

The Unary block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

Weighted Sample Time Math

The Weighted Sample Time Math block receives decision coverage. If the **Saturate on integer overflow** parameter is selected, the decision coverage measures:

- The number of times the block saturates on integer overflow, indicating a true decision.

- The number of times the block does not saturate on integer overflow, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

While Iterator, While Iterator Subsystem

The While Iterator block and While Iterator Subsystem receive decision coverage. Decision coverage is measured for the `while` condition value, which is determined by the `while` condition being satisfied (true), or the `while` condition not being satisfied (false). Simulink Verification and Validation software reports the total number of times that each `while` condition evaluates to true and to false. If the `while` condition evaluates to true at least once, and false at least once, decision coverage for the `while` condition is 100%. If no `while` conditions are true, or if no `while` conditions are false, decision coverage is 50%.

If the iteration limit is exceeded (true) or is not exceeded (false), the software measures decision coverage independently. If the iteration limit evaluates to true at least once, and false at least once, decision coverage for the iteration limit is 100%. If no iteration limits are true, or if no iteration limits are false, decision coverage is 50%. If you set **Maximum number of iterations** to -1 (no limit), the decision coverage for the iteration limit is true for all iterations and false for zero iterations, and decision coverage is 50%.

Model Objects That Do Not Receive Coverage

The Simulink Verification and Validation software cannot record coverage for blocks that are not listed in Chapter 16, "Model Objects That Receive Model Coverage". The following table identifies specific model objects that do not receive coverage.

| Model object	Does not receive coverage...
Model blocks	When the **Simulation mode** parameter specifies Accelerator. Coverage for Model blocks is the sum of the coverage data for the contents of the referenced model.
Subsystem block	When the **Read/Write Permissions** parameter is set to NoReadOrWrite.

Setting Model Coverage Options

Coverage Settings Dialog Box

Before starting a model coverage analysis, you must specify several model coverage recording and reporting options. In a Simulink model, select **Tools > Coverage Settings**. The Coverage Settings dialog box opens, with the **Coverage** tab displayed.

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

Coverage Tab

On the **Coverage** tab, select the model coverages calculated during simulation.
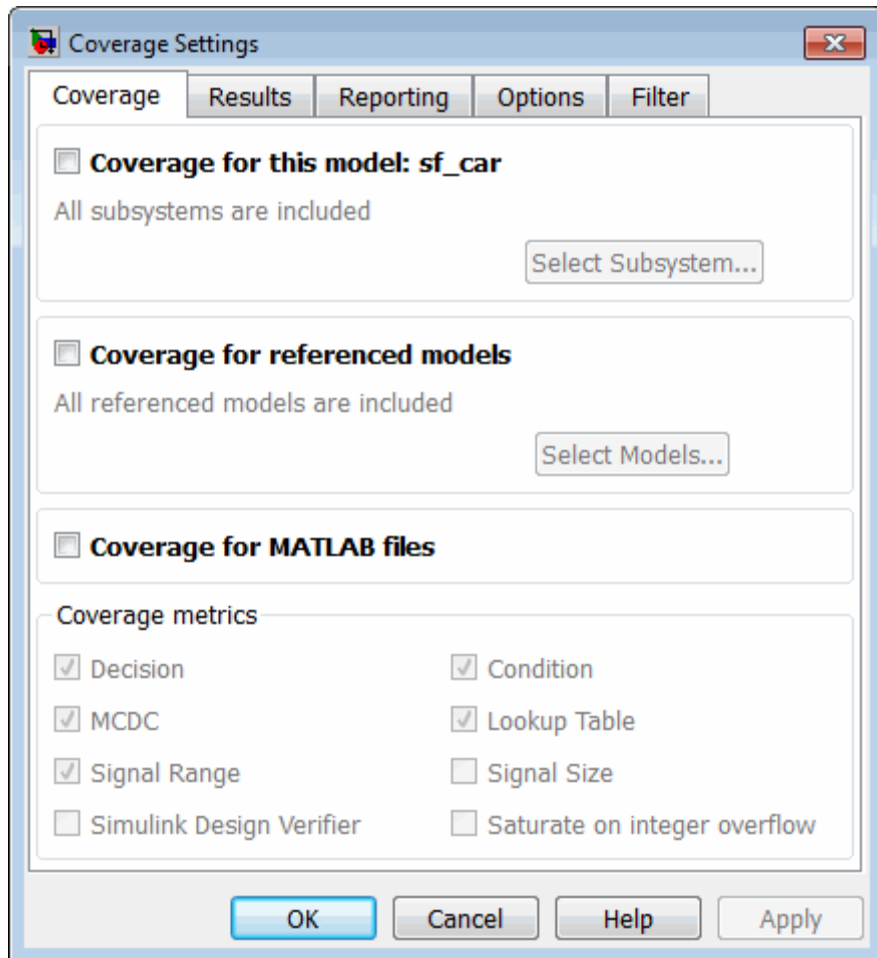
Coverage for this model

Instructs the 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.
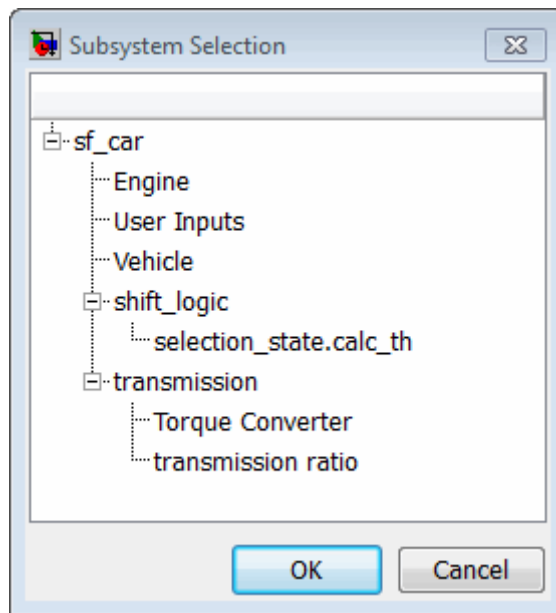
Select Subsystem

Specifies the subsystem for which the 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 On the **Coverage** tab, click **Select Subsystem**.

The Subsystem Selection dialog box opens.

2 In the Subsystem Selection dialog box, select the subsystem for which you want to enable coverage reporting and click **OK**.

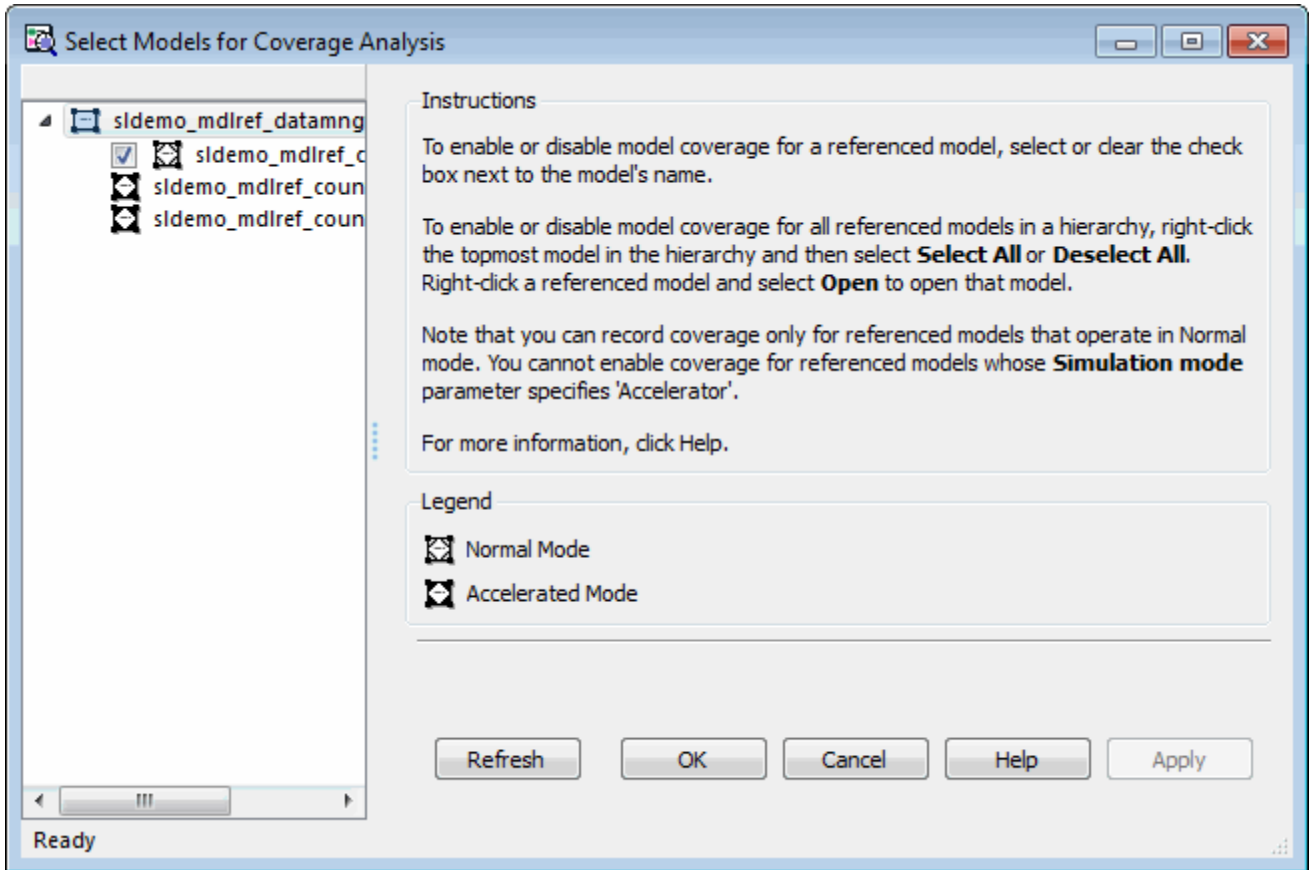## Coverage for referenced models

Causes the software to record 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

Click to specify the referenced models for which the Simulink Verification and Validation software records and reports coverage data. When you select **Coverage for referenced models**, the software, by default, generates coverage data for all referenced models where the simulation mode of the Model block is set to Normal.

To enable coverage reporting for particular referenced models:

1 On the **Coverage** pane, click **Select Models**.

2 In the Select Models for Coverage Analysis dialog box, select the referenced models for which you want to record coverage.

The icon next to the model name indicates the simulation mode for that referenced model. You can select only referenced models whose simulation mode is set to Normal.

If you have multiple Model blocks that reference the same model and whose simulation mode is set to Normal, selecting or clearing one check box for that model causes the check boxes for all Normal mode instances of that model to be selected or cleared.

3 Click **OK** to close the Select Models for Coverage Analysis dialog box and return to the Coverage Settings dialog box.

Coverage for MATLAB files

Enables coverage for any external functions in your model that MATLAB functions call. The MATLAB functions may be defined in a MATLAB Function block or in a Stateflow chart.

You must select either **Coverage for this model** or **Coverage for referenced models** to select the **Coverage for 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 15-4). The Simulink Verification and Validation software gathers and reports those types of coverage for the subsystem, model, and referenced models.

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 21-17.

Results Tab

On the **Results** tab, select the destination for model coverage results.
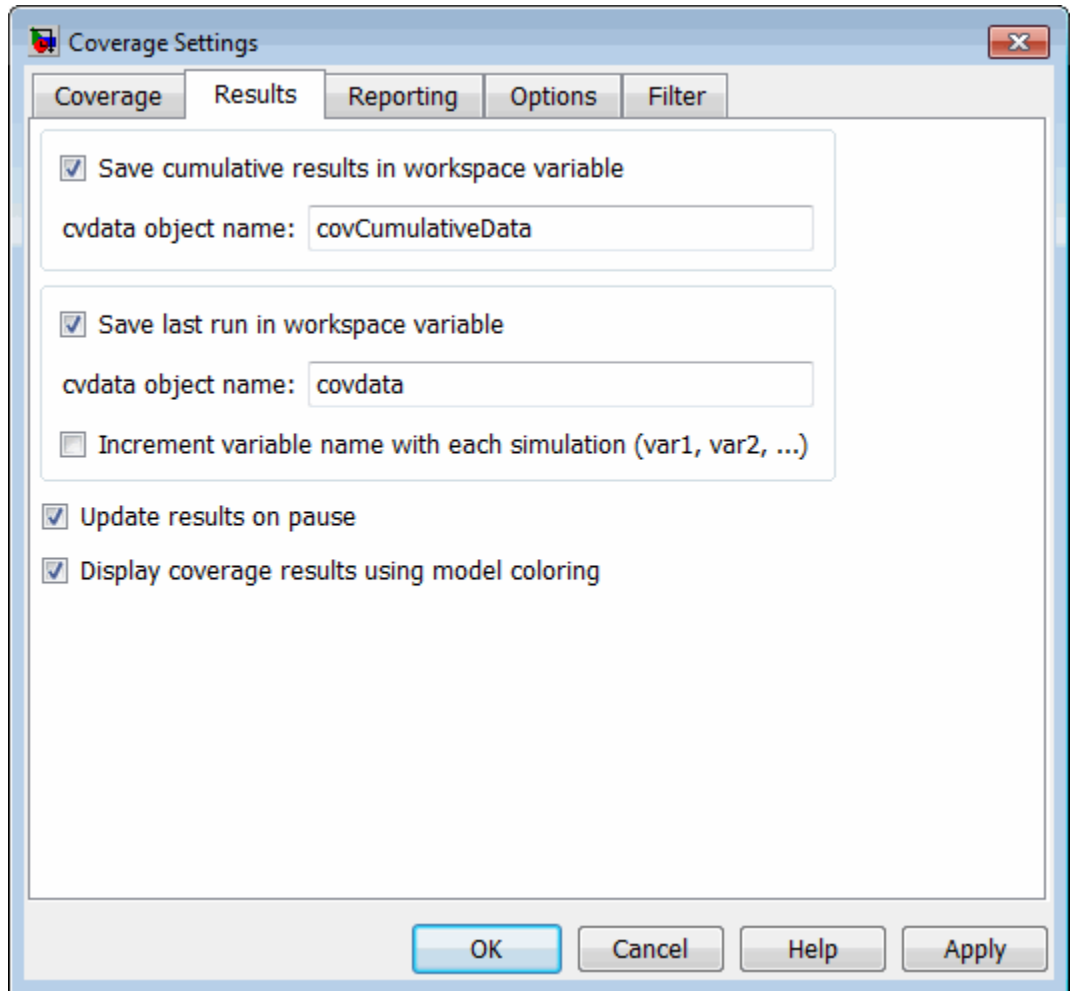
Save cumulative results in workspace variable

Causes model coverage to accumulate and save the results of successive simulations in a workspace variable. You specify the workspace variable in the **cvdata object name** field.

Save last run in workspace variable

Causes model coverage to save the results of the last simulation run in a workspace variable. You specify that workspace variable in the **cvdata object name** field below.

Increment variable name with each simulation

Causes the Simulink Verification and Validation software to increment the name of the coverage data object variable that saves the coverage data from last run with each simulation, so that the current simulation run does not overwrite the results of the previous run.

Update results on pause

Causes the model coverage results to be recorded up to the point at which you pause the simulation for the first time. When you resume simulation and later pause or stop, the model coverage report reappears, 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. See "Viewing Coverage Results in a Model" on page 18-5.

Reporting Tab

On the **Reporting** tab, specify whether the model coverage tool generates an HTML report and what data the report includes.
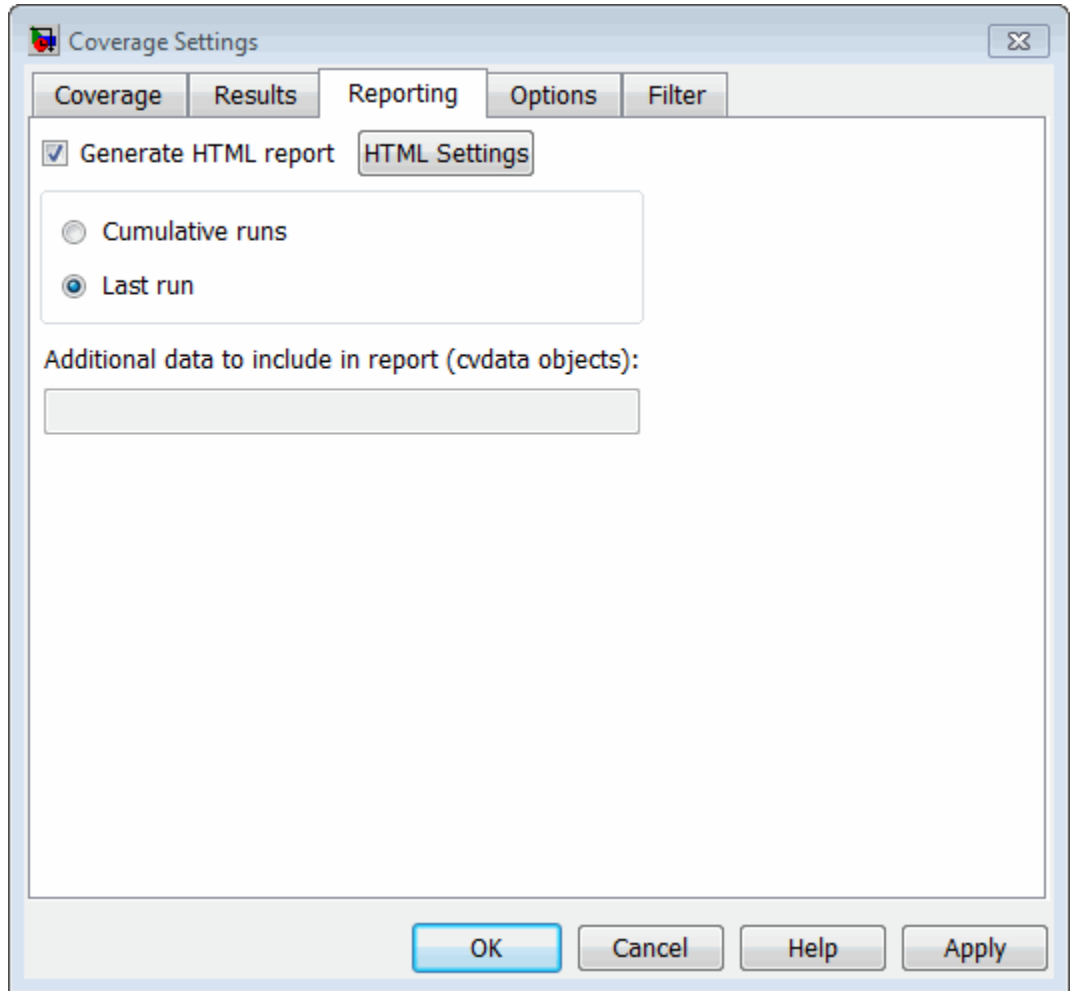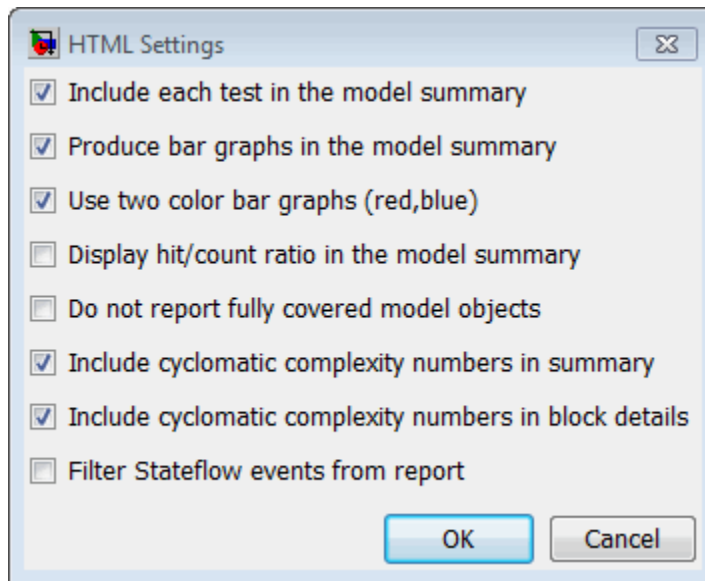
Generate HTML report

Causes the Simulink Verification and Validation software to create an HTML report containing the coverage data. At the end of the simulation, the report opens in the MATLAB Web browser. Click the **Settings** button to select various reporting options.

Settings

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

Option	Description
Include each test in the model summary	At the top of the HTML report, the model hierarchy table 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 for a visual representation of the coverage.
Use two color bar graphs (red, blue)	Red and blue bar graphs are displayed in the report instead of black and white bar graphs.
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	The coverage report includes only model objects that the simulation does not cover fully, useful when 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 15-4) 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 for Stateflow Chart blocks.

Option	Description
Include cyclomatic complexity numbers in block details	Includes the cyclomatic complexity metric in the block details section of the report.
Filter Stateflow events from report	Excludes coverage data on Stateflow events.

Cumulative Runs

Displays the coverage results from successive simulations in the report. For more information, see "Save cumulative results in workspace variable" on page 17-9.

On the **Results** tab, if you select the **Save cumulative results in workspace variable** check box, 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. The `cvload` parameter `RESTORETOTAL` must be `1` in order to restore cumulative results. At the end of the sessions, use `cvsave` to save results to a file. At the beginning of the session, `cvload` to load the 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 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. 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. For cumulative reports, this information includes all the simulations where cumulative results are stored.

You can also calculate cumulative coverage results at the command line, through the + operator:

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

Last run

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

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 creates a new set of columns in the report.

Options Tab

On the **Options** tab, select options for model coverage reports.
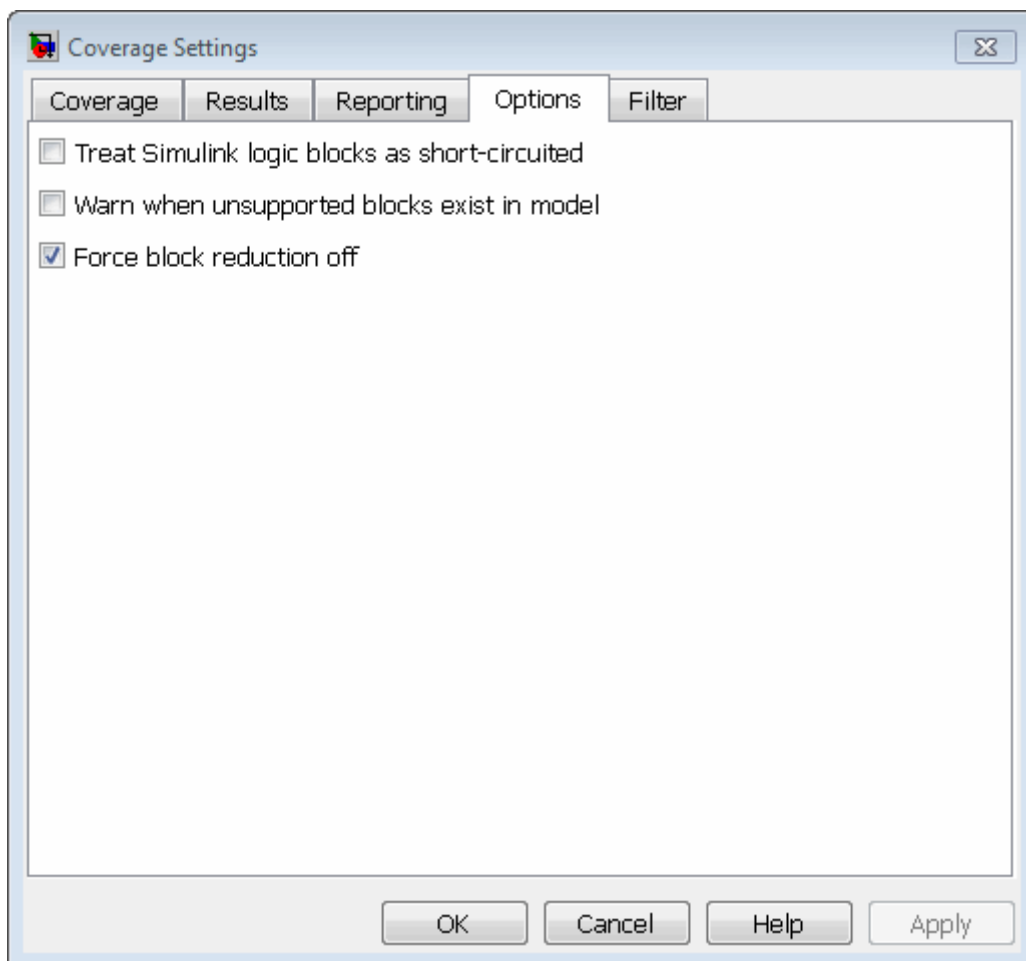
Treat Simulink logic blocks as short-circuited

The **Treat Simulink logic blocks as short-circuited** option applies only to condition and MCDC coverage. If you select this option, coverage analysis

treats Simulink logic blocks 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, MCDC coverage analysis ignores the values of the other inputs when determining MCDC coverage for a test case.

If you enable this feature and logic blocks are short-circuited while collecting model coverage, you may not be able to achieve 100% coverage for that block.

To generate code from a model, select this option. Also select this option for where you want the MCDC coverage analysis to approximate the degree of coverage that your test cases achieve for the generated code (most high-level languages short-circuit logic expressions).

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

Warn when unsupported blocks exist in model

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

Force block reduction off

To achieve faster execution during model simulation and in generated code, in the Configuration Parameters dialog box, on the **Optimization** pane, enable the **Block reduction** parameter. The Simulink software collapses certain groups of blocks into a single, more efficient block, or removes them entirely.

One of the model coverage options, **Force block reduction off**, allows you to ignore the **Block reduction** parameter when collecting model coverage.

If you do not enable the **Block reduction** parameter, or if you select **Force block reduction off**, the Simulink Verification and Validation software provides coverage data for every block in the model that collects coverage.

If you enable the **Block reduction** parameter and do not set **Force block reduction off**, the coverage report lists the reduced blocks that would have collected coverage.

The model coverage report identifies any reduced blocks. For an example of a reduced blocks report, see "Block Reduction" on page 19-29.

Filter Tab

On the **Filter** tab, enter the file name that specifies the model objects to be excluded from model coverage collection. You can use the same filter file for multiple models.
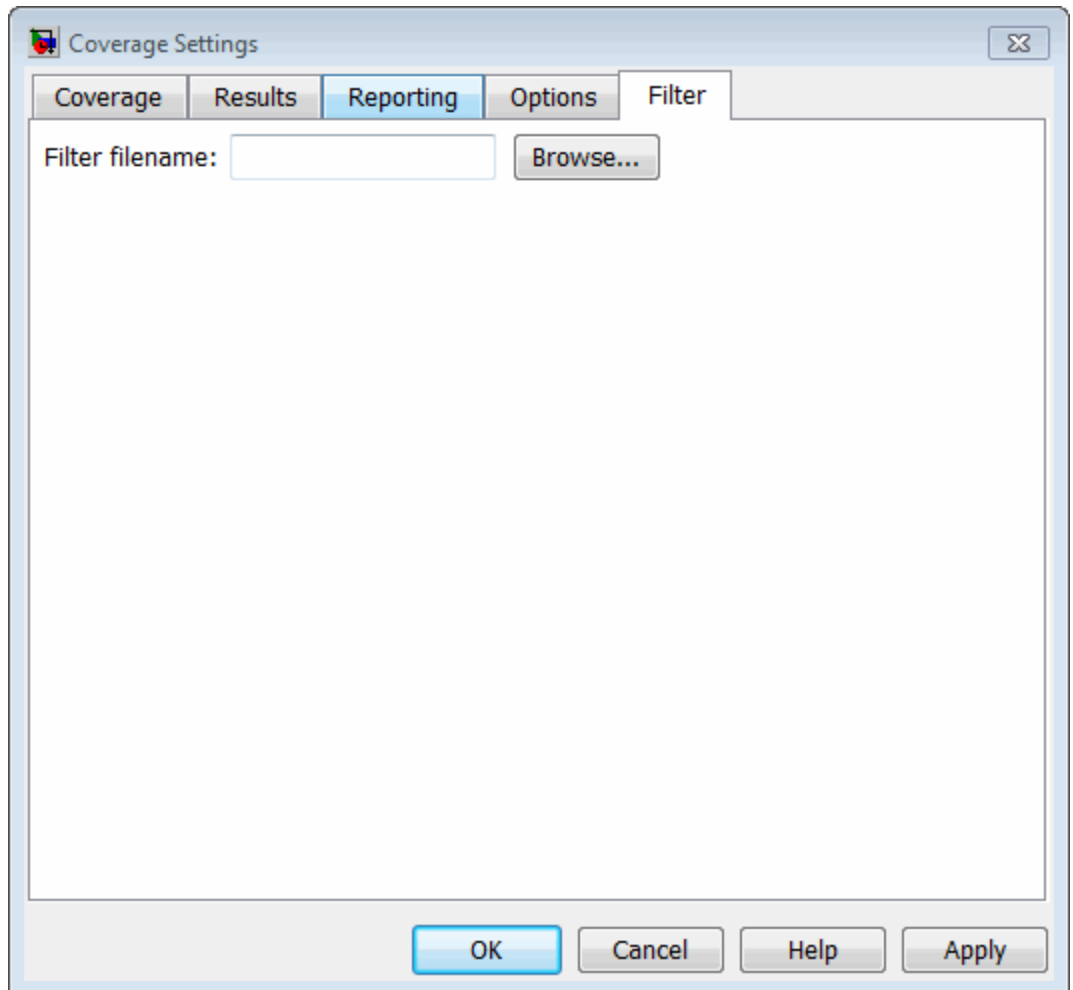
Filter file name

If you enable coverage for this model, you can create a filter file or open an existing filter file. In your model, you can then specify objects that you want to exclude from model coverage collection during simulation.

In the **Filter file name** field, enter the full path to the MAT-file that specifies the model objects to be excluded from model coverage collection. Or, click **Browse** to navigate to the file. You can only open MAT-files that have the valid filter file format.

If the current model has a filter file already associated with it, the file name appears in the **Filter file name** field, and the **Open in Editor** link is displayed. To edit the coverage filter settings, click this link.
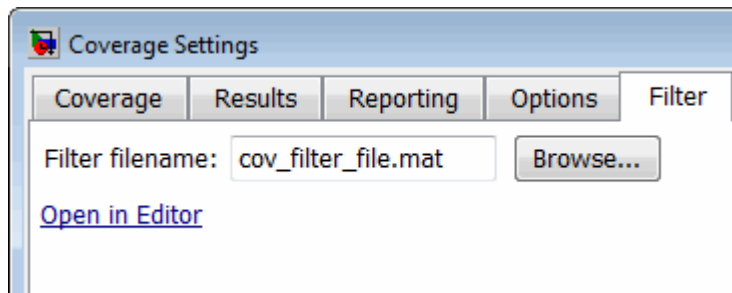
If the **Open in Editor** link is unavailable, go to the **Coverage** tab. Select **Coverage for this model** to enable coverage for the current model. You can then enter the filter file name and edit the file.

18

Coverage Collection During Simulation

Model Coverage Collection 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 18-3.)

2 Run the test cases to verify model behavior.

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 to cover areas not currently covered.

5 Repeat the preceding steps until you are satisfied with the coverage of your test suite.

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

Creating and Running Test Cases

To create and run test cases, model coverage provides two MATLAB commands, `cvtest` and `cvsim`. The `cvtest` command creates test cases that the `cvsim` command runs. (See "Running Tests with cvsim" on page 21-6.)

You can also run the coverage tool interactively:

1 Open the `sldemo_fuelsys` model.

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

The Coverage Settings dialog box **Coverage** tab appears.

3 Select **Coverage for this model: sldemo_fuelsys**, which enables:

- The **Select Subsystem** button
- The metrics options in the **Coverage metrics** section
- Fields on the **Results**, **Reporting**, and **Options** tabs of the Coverage Settings dialog box

4 Under **Coverage metrics**, select the types of coverage that you want to record in the coverage report.

For a complete description of all coverage options in the Coverage Settings dialog box, see Chapter 17, "Setting Model Coverage Options".

5 Click **OK**.

6 In the Simulink model window, select **Start > Simulation** or on the Simulink toolbar, click the **Start** button 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. At the end of the simulation, this data appears in an HTML report that opens in a browser window.

Note You cannot run simulations if you select both model coverage reporting and acceleration options. If you select **Accelerator** mode in the model window, Simulink does not record coverage.

You cannot select both block reduction and conditional branch input optimization when you perform coverage analysis because they interfere with coverage recording.

Viewing Coverage Results in a Model

In this section...
"Overview of Model Coverage Highlighting" on page 18-5
"Enabling Coverage Highlighting" on page 18-6
"Examples: Model Coverage Coloring" on page 18-6
"Coverage Display Window" on page 18-9

Overview of Model Coverage Highlighting

When you simulate a Simulink model, you can configure your model to provide visual results that allow you to see at a glance which objects recorded 100% coverage. After the simulation:

- In the model window, model objects are highlighted in certain colors according to what coverage was recorded:
  - Light green indicates that an object received full coverage during testing.
  - Light red indicates that an object received incomplete coverage.
  - Gray indicates that an object was filtered from coverage.
  - Objects with no color highlighting received no coverage.
- When you click a colored object, the Coverage Display Window provides details about the coverage recorded for that block. For subsystems and Stateflow charts, the Coverage Display Window lists the summary coverage for all objects in that subsystem or chart. For other blocks, the Coverage Display Window list specific details about the objects that did not receive 100% coverage.

The simulation highlights blocks that received the following types of model coverage:

- "Decision Coverage (DC)" on page 15-5
- "Condition Coverage (CC)" on page 15-5
- "Modified Condition/Decision Coverage (MCDC)" on page 15-5

• "Simulink Design Verifier Coverage" on page 15-9

Enabling Coverage Highlighting

To enable the model coverage colored diagram display:

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

2 In the **Coverage** tab, select **Coverage for this model**.

3 Select the **Results** tab.

**4** Select **Display coverage results using model coloring**. This is the default setting.

After you have enabled the coverage coloring, simulate your model. In the model, you can see at a glance which objects received full, partial, or no coverage.

Examples: Model Coverage Coloring

The following sections show examples of model objects that

Green: Full Coverage

In this example, the Switch block received 100% coverage, as indicated by the green highlighting and the information in the Coverage Display Window.
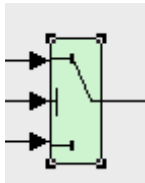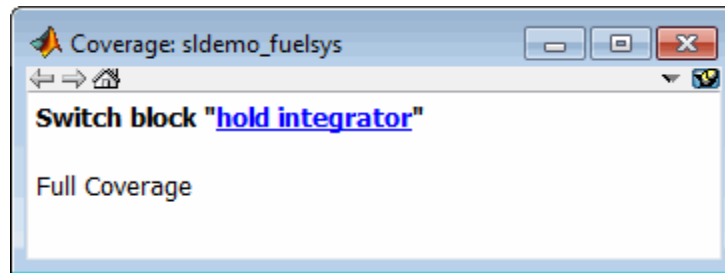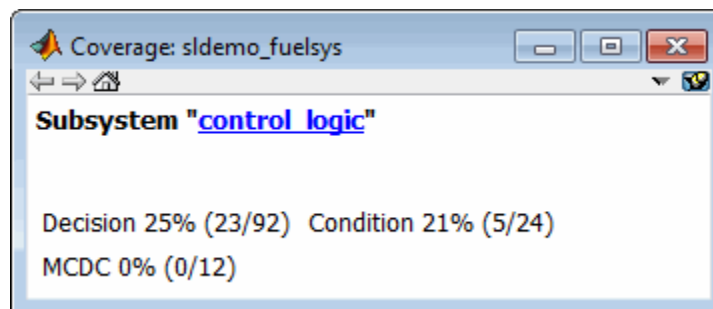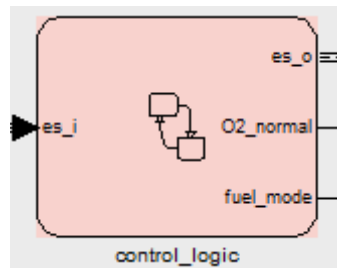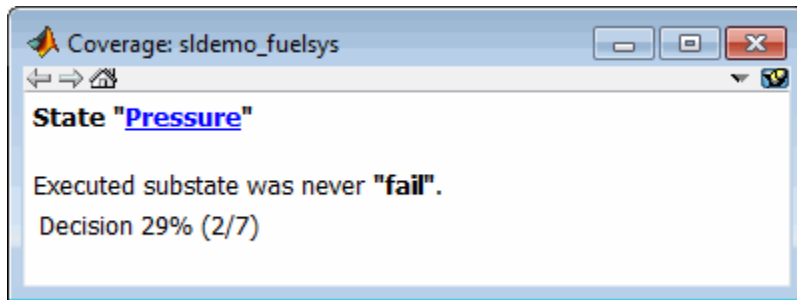
Red: Partial Coverage

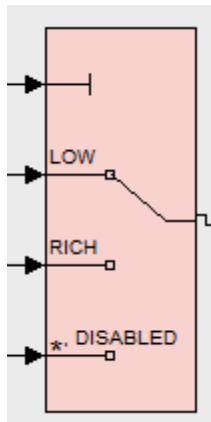In this example, the control_logic Stateflow chart received the following coverage:

- Decision: 25%
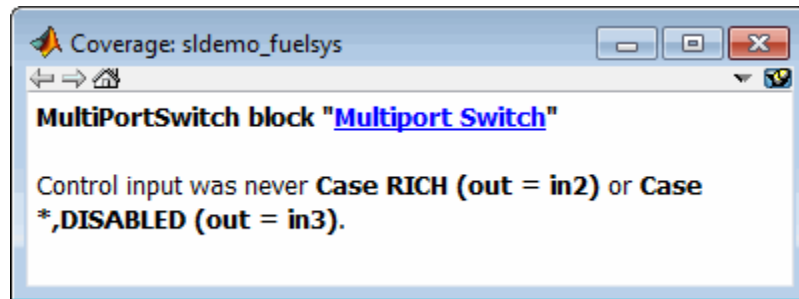
- Condition: 21%

- MCDC: 0%

Inside the control_logic subsystem, the Pressure substate was never `fail`.
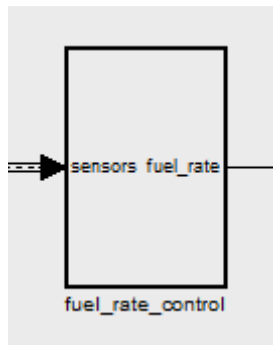
In the next example, in the Multiport Switch block, the second and third data ports were never executed.

Gray: Filtered Coverage

In this example, the fuel_rate_control subsystem is highlighted in gray because it was filtered out of coverage recording.

Coverage Display Window

After simulating the model and recording coverage, by default, the Coverage Display Window is the top-most visible window. When you click an object that recorded coverage, the Coverage Display Window displays details of the coverage recorded during simulation.

In the Coverage Display Window, you can:

• Configure the window so it is not always the top-most visible window. Next to **Always on top**, click 🔲 and removing the check.

- Configure the window to display coverage information when you click an object that recorded coverage. Click ▾ and select **Click**.

- Configure the window to display coverage information when you hover your cursor on an object that recorded coverage. Click ▾ and select **Focus**.

- Close the window. Press **Alt+F4**.

- Close the window and remove all highlighting on the model. Select **View > Remove highlighting**.

Model Coverage for Multiple Instances of a Referenced Model

In this section...
"About Coverage for Model Blocks" on page 18-11
"Recording Coverage for Multiple Instances of a Referenced Model" on page 18-11

About Coverage for Model Blocks

Model blocks do not receive coverage directly; if you set the simulation mode of the Model block to `Normal`, the Simulink Verification and Validation software records coverage for the model referenced from the Model block. If the simulation mode for the Model block is anything other than `Normal`, the software does not record coverage for the referenced model.

Your Simulink model can contain multiple Model blocks with `Normal` simulation mode that reference the same model. When the software records coverage, each instance of the referenced model can be exercised with different inputs or parameters, possibly resulting in additional coverage for the referenced model.

The Simulink Verification and Validation software records coverage for all instances of the referenced model and combines the coverage data for that referenced model in the final results.

Recording Coverage for Multiple Instances of a Referenced Model

To see how this works, simulate a model twice. The first time, you record coverage for one Model block in `Normal` simulation mode. The second time, you record coverage for two Model blocks in `Normal` simulation mode. Both Model blocks reference the same model.

• "Recording Coverage for the First Instance of the Referenced Model" on page 18-12

Recording Coverage for the First Instance of the Referenced Model

Record coverage for the Counter1 block:

1 Open the `sldemo_mdlref_datamngt` demo model:

`sldemo_mdlref_datamngt`

This model contains three Model blocks that reference the `sldemo_mdlref_counter_datamngt` model. The corners of each Model block indicate the value of their **Simulation mode** parameter:

- Counter1 — Simulation mode: `Normal`
- Counter2 — Simulation mode: `Accelerator`
- Counter3 — Simulation mode: `Accelerator`

2 Configure the `sldemo_mdlref_datamngt` model to record coverage during simulation:

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

b On the **Coverage** tab, select:

- **Coverage for this model: sldemo_mdlref_datamngt**
- **Coverage for referenced models**

c Under **Coverage for referenced models**, click **Select Models**. In the Select Models for Coverage Analysis dialog box, you can select only those referenced models whose simulation mode is `Normal`. In this example, only the first instance of `sldemo_mdlref_counter_datamngt` is available for recording coverage.
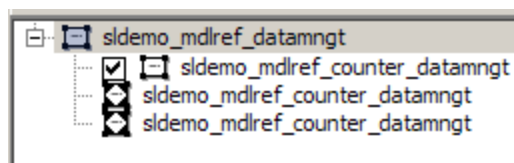
d On the **Reporting** tab, select **Last Run** so that you can compare coverage data from individual simulations, not accumulated coverage for successive simulations.

e Click **OK** to exit the Select Models for Coverage Analysis dialog box.

3 Click **OK** to save your coverage settings and exit the Coverage Settings dialog box.

4 Simulate the `sldemo_mdlref_datamngt` model.

When the simulation is complete, the HTML coverage report opens. The coverage data for the referenced model, `sldemo_mdlref_counter_datamngt`, shows that the model achieved 69% coverage.

5 Click the hyperlink in the report for the referenced model.

The detailed coverage report for the referenced model opens. Note the following about the coverage for the Range Check subsystem:

- The Saturate Count block executed 100 times. This block has four Boolean decisions. Decision coverage was 50%, because two of the four decisions were never recorded:

  - The decision `input > lower limit` was never `false`.

  - The decision `input >= upper limit` was never `true`.

- The `DetectOverflow` function executed 50 times. This script has five decisions. The `DetectOverflow` script achieved 60% coverage because two of the five decisions were never recorded:

  - The expression `count >= CounterParams.UpperLimit` was never `true`.

  - The expression `count > CounterParams.LowerLimit` was never `false`.

eM Function "DetectOverflow"

Parent:	sldemo_mdlref_counter_datamngt/Range Check/Detect Overflow
Uncovered Links:	◄

Metric	Coverage
Cyclomatic Complexity	3
Decision (D1)	60% (3/5) decision outcomes

```
 1  function result = DetectOverflow(count, CounterParams)
 2  % DETECTOVERFLOW   Check count
 3  %#eml
 4
 5  if (count >= CounterParams.UpperLimit)
 6     result = SlDemoRangeCheck.UpperLimit;
 7  elseif (count > CounterParams.LowerLimit)
 8     result = SlDemoRangeCheck.InRange;
 9  else
10     result = SlDemoRangeCheck.LowerLimit;
11  end
12
```

#1: function result = DetectOverflow(count, CounterParams)

Decisions analyzed:

function result = DetectOverflow(count, CounterParams)	100%
executed	50/50

#5: if (count >= CounterParams.UpperLimit)

Decisions analyzed:

if (count >= CounterParams.UpperLimit)	50%
false	50/50
true	0/50

#7: elseif (count > CounterParams.LowerLimit)

Decisions analyzed:

elseif (count > CounterParams.LowerLimit)	50%
false	0/50
true	50/50

Recording Coverage for the Second Instance of the Referenced Model

Set the simulation mode of a second Model block, Counter2, to `Normal` and simulate the model. In this example, the Counter2 block adds to the coverage for the model referenced from both Model blocks:

1 In the model window for `sldemo_mdlref_datamngt`, right-click the Counter2 block and select **ModelReference Parameters**.

The Function Block Parameters dialog box opens.

2 To make sure the software records coverage for the Counter2 block, set the **Simulation mode** parameter to `Normal`.

3 Click **OK** to save your change and exit the Function Block Parameters dialog box.

The corners of the Counter2 block change to indicate that the simulation mode for this block is `Normal`.

4 To make sure that the software records coverage for both instances of this model:

a Select **Tools > Coverage Settings**.

b On the **Coverage** pane, under **Coverage for referenced models**, click **Select Models**.

In the Select Models for Coverage Analysis dialog box, both instances of the referenced model, `sldemo_mdlref_counter_datamngt`, are selected. If you have multiple instances of a referenced model in `Normal` mode, you can choose to record coverage for all of them or none of them.

c Click **OK** to close the Select Models for Coverage Analysis dialog box.

5 Simulate the `sldemo_mdlref_datamngt` model again.

6 When the simulation is complete, open the HTML coverage report.

The referenced model, `sldemo_mdlref_counter_datamngt` achieved 85% coverage. Note the following about the coverage data for the Range Check subsystem:

- The Saturate Count block executed 179 times. The simulation of the Counter2 block executed the Saturate Count block an additional 79 times, for a total of 179 executions.

  The decision input >= upper limit was true 21 times during this simulation, compared to 0 during the first simulation. The fourth decision input > lower limit was still never false. Three out of four decisions were recorded during simulation, so this block achieved 75% coverage.

Saturate block "Saturate Count"

Parent:	sldemo_mdlref_counter_datamngt/Range Check
Uncovered Links:	➡

Metric	Coverage
Cyclomatic Complexity	2
Decision (D1)	75% (3/4) decision outcomes

Decisions analyzed:

input > lower limit	50%
false	0/79
true	79/79

input >= upper limit	100%
false	79/100
true	21/100

- The DetectOverflow function executed 100 times. The simulation of the Counter2 block executed the DetectOverflow function an additional 50 times.

  The DetectOverflow function has five decisions. The expression count >= CounterParams.UpperLimit was true 21 times during this simulation, compared to 0 during the first simulation. The expression

`count > CounterParams.LowerLimit` was never `false`. Four out of five decisions were recorded during simulation, so the `DetectOverflow` function achieved 80% coverage.

eM Function "DetectOverflow"

Parent:	sldemo_mdlref_counter_datamngt/Range Check/Detect Overflow
Uncovered Links:	◀

Metric	**Coverage**
Cyclomatic Complexity	3
Decision (D1)	80% (4/5) decision outcomes

```
 1   function result = DetectOverflow(count, CounterParams)
 2   % DETECTOVERFLOW  Check count
 3   %#eml
 4
 5   if (count >= CounterParams.UpperLimit)
 6     result = SlDemoRangeCheck.UpperLimit;
 7   elseif (count > CounterParams.LowerLimit)
 8     result = SlDemoRangeCheck.InRange;
 9   else
10     result = SlDemoRangeCheck.LowerLimit;
11   end
12
```

#1: function result = DetectOverflow(count, CounterParams)

Decisions analyzed:

function result = DetectOverflow(count, CounterParams)	100%
executed	100/100

#5: if (count >= CounterParams.UpperLimit)

Decisions analyzed:

if (count >= CounterParams.UpperLimit)	100%
false	79/100
true	21/100

#7: elseif (count > CounterParams.LowerLimit)

Decisions analyzed:

elseif (count > CounterParams.LowerLimit)	50%
false	0/79
true	79/79

Model Coverage for MATLAB Functions

In this section...
"About Model Coverage for MATLAB Functions" on page 18-20
"Types of Model Coverage for MATLAB Functions" on page 18-20
"How to Collect Coverage for MATLAB Functions" on page 18-22
"Examples: Model Coverage for MATLAB Functions" on page 18-23

About Model Coverage for MATLAB Functions

The Simulink Verification and Validation software simulates a Simulink model and reports model coverage data for the decisions and conditions of code in MATLAB Function blocks. Model coverage only supports coverage for MATLAB functions configured for code generation.

For example, consider the following `if` statement:

```
if (x > 0 || y > 0)
 reset = 1;
```

The `if` statement contains a decision with two conditions (`x > 0` and `y > 0`). The Simulink Verification and Validation software verifies that all decisions and conditions are taken during the simulation of the model.

Types of Model Coverage for MATLAB Functions

The types of model coverage that the Simulink Verification and Validation software records for MATLAB functions configured for code generation are:

- "Decision Coverage" on page 18-20
- "Condition and MCDC Coverage" on page 18-21
- "Simulink® Design Verifier™ Coverage" on page 18-21

Decision Coverage

During simulation, the following MATLAB Function block 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 evaluates to `false` at least once.

Condition and MCDC Coverage

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

- `if` statement conditions
- `while` statement conditions

Simulink Design Verifier Coverage

The following MATLAB functions are active in code generation and in Simulink Design Verifier:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

When you specify the **Simulink Design Verifier** coverage metric in the Coverage Settings dialog box, the Simulink Verification and Validation software records coverage for these functions.

Each of these functions evaluates an expression *expr*, for example, `sldv.test(expr)`, where *expr* is any valid Boolean MATLAB expression.

Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to `true`.

If *expr* is `true` for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Verification and Validation software reports coverage for that function as 0%.

For an example of coverage data for Simulink Design Verifier functions in a coverage report, see "Simulink® Design Verifier™ Coverage" on page 19-36.

How to Collect Coverage for MATLAB Functions

When you simulate your model, the Simulink Verification and Validation software can collect coverage data for MATLAB functions configured for code generation. To enable model coverage, select **Tools > Coverage Settings** and select **Coverage for this model**.

You collect model coverage for MATLAB functions as follows:

- Functions in a MATLAB Function block
- Functions in an external MATLAB file

  To collect coverage for an external MATLAB file, on the **Coverage** tab of the Coverage Settings dialog box, select **Coverage for External MATLAB files**.

- Simulink Design Verifier functions:
  - `sldv.condition`
  - `sldv.test`
  - `sldv.assume`
  - `sldv.prove`

  To collect coverage for these functions, on the Coverage tab of the Coverage Settings dialog box, select the **Simulink Design Verifier** coverage metric.

The following section provides model coverage examples for each of these situations.

Examples: Model Coverage for MATLAB Functions

- "Model Coverage for MATLAB Function Blocks" on page 18-23

- "Model Coverage for MATLAB Functions in an External File" on page 18-35

- "Model Coverage for Simulink® Design Verifier™ MATLAB Functions" on page 18-36

Model Coverage for MATLAB Function Blocks

Simulink Verification and Validation software measures model coverage for functions in a MATLAB Function block.

The following model contains two MATLAB functions in its MATLAB Function block:
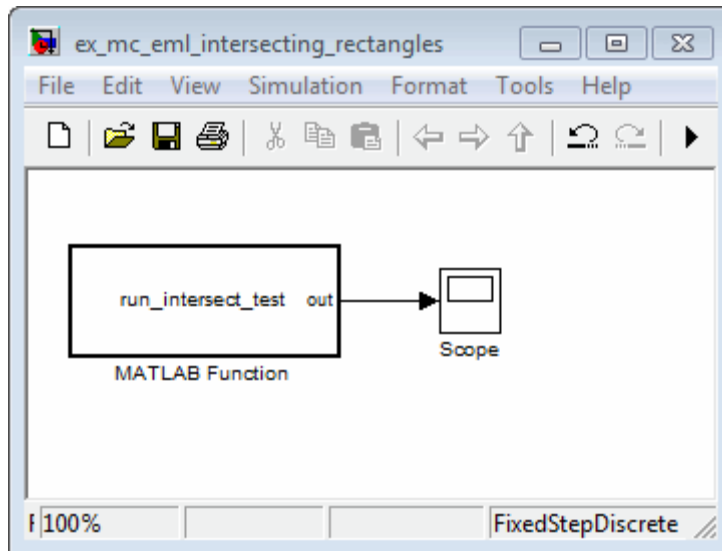
In the Configuration Parameters dialog box, on the **Solver** pane, under **Solver options**, the simulation parameters are set as follows:

- **Type** — Fixed-step

- **Solver** — discrete (no continuous states)

- **Fixed-step size (fundamental sample time)** — 1

The 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 `rect_intersect`.
- The subfunction, `rect_intersect`, tests for intersection between the two rectangles. 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 shown in the following graphic.
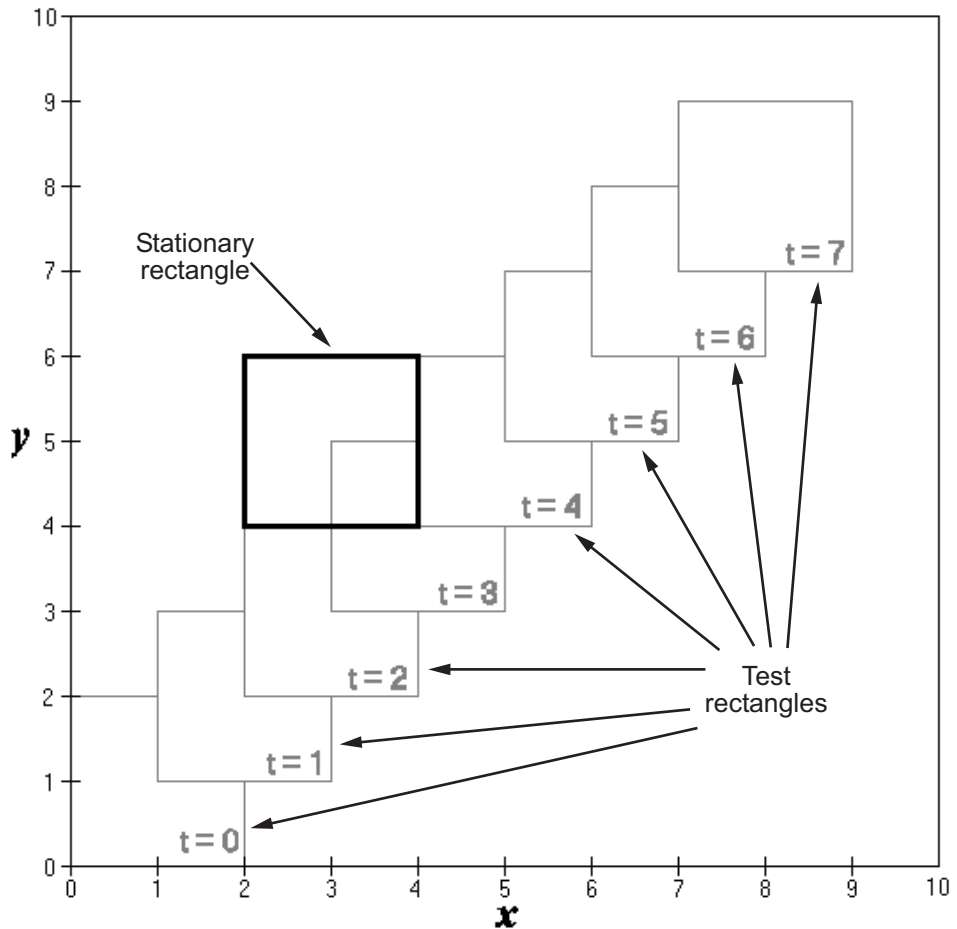
The fixed rectangle is shown in bold with a lower-left origin of (2,4) and a width and height of 2. At time t = 0, the first test rectangle has an origin of (0,0) and a width and height of 2. For each succeeding sample, the origin of the test rectangle increments by (1,1). The rectangles at sample times 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 times 2, 3, and 4 .
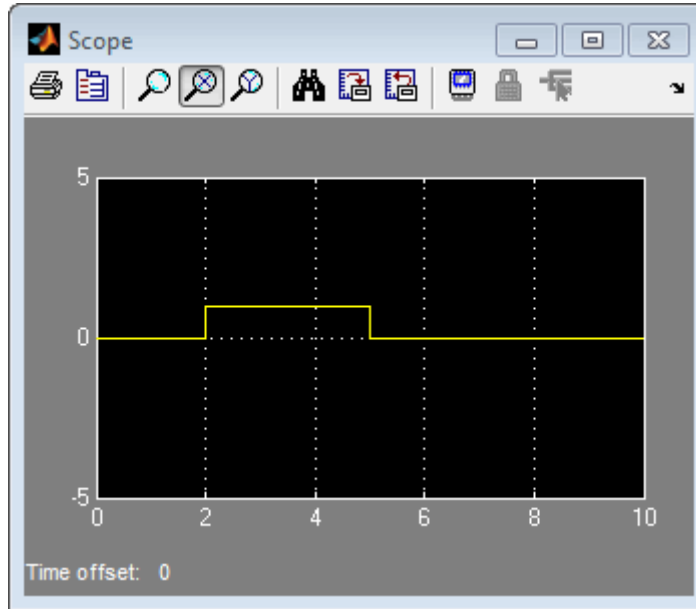
After the simulation, the model coverage report appears in a browser window. After the summary in the report, the Details section of the model coverage report reports on each parts of the model.

The model coverage report for the MATLAB Function block shows that the block itself has no decisions of its own apart from its function.

The following sections 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 at the top of each section.

Coverage for the MATLAB Function run_intersect_test. Model coverage
for the MATLAB Function block function run_intersect_test appears
under the linked name of the function. Clicking this link opens the function
in the editor.

Below the linked function name is a link to the model coverage report
for the parent MATLAB Function block that contains the code for
run_intersect_test.

MATLAB Function "run_intersect_test"

Parent:	ex_mc_eml_intersecting_rectangles/MATLAB Function
Uncovered Links:	

Metric	Coverage
Cyclomatic Complexity	7
Decision (D1)	100% (8/8) decision outcomes
Condition (C1)	88% (7/8) condition outcomes
MCDC (C1)	75% (3/4) conditions reversed the outcome

The top half of the report for the function summarizes its model coverage
results. The coverage metrics for run_intersect_test include decision,
condition, and MCDC coverage. You can best understand these metrics by
examining the code for run_intersect_test.

```
 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 x1 y1;
 6  if isempty(x1)
 7      x1 = -1, y1 = -1;
 8  end
 9
10  x1 = x1 + 1;
11  y1 = y1 + 1;
12  out = rect_intersect([x1 y1 2 2]', [2 4 2 2]');
13
14  function out = rect_intersect(rect1, rect2);
15  % Return 1 if two rectange 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
```

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.

  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 in the report for the decision in line 30.

The following sections display the coverage for each run_intersect_test decision line. The coverage for each line is titled with the line itself, which if clicked, opens the editor to the designated line.

Coverage for Line 1. The coverage metrics for line 1 are part of the coverage data for the function run_intersect_test.

The first line of every MATLAB function configured for code generation 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 at least once during simulation.

#1: function out = run_intersect_test

Decisions analyzed:

function out = run_intersect_test	100%
executed	11/11

Coverage for Line 6. 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%.

#6: if isempty(x1)

Decisions analyzed:

if isempty(x1)	100%
false	10/11
true	1/11

Coverage for Line 14. The **Decisions Analyzed** table indicates that the subfunction rect_intersect executed during testing, thus receiving 100% coverage.

#14: function out = rect_intersect(rect1, rect2);

Decisions analyzed:

function out = rect_intersect(rect1, rect2);	100%
executed	11/11

Coverage for Line 27. 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. 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 the second condition was evaluated 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.

#27: if (top1 < bottom2 || top2 < bottom1)

Decisions analyzed:

if (top1 < bottom2 || top2 < bottom1)	100%
false	5/11
true	6/11

Conditions analyzed:

Description:	True	False
top1 < bottom2	2	9
top2 < bottom1	4	5

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

Decision/Condition:	True Out	False Out
top1 < bottom2 || top2 < bottom1		
top1 < bottom2	Tx	FF
top2 < bottom1	FT	FF

Coverage for Line 30. 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 are
false. Because both the true and false outcomes are 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 do 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.

#30: if (right1 < left2 || right2 < left1)

Decisions analyzed:

if (right1 < left2 || right2 < left1)	100%
false	3/5
true	2/5

Conditions analyzed:

Description:	True	False
right1 < left2	0	5
right2 < left1	2	3

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

Decision/Condition:	True Out	False Out
right1 < left2 || right2 < left1		
right1 < left2	(Tx)	FF
right2 < left1	FT	FF

Coverage for run_intersect_test. On the **Details** tab, the metrics that summarize coverage for the entire run_intersect_test function are reported and repeated as shown.

MATLAB Function "run_intersect_test"

Parent: ex_mc_eml_intersecting_rectangles/MATLAB Function
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	7
Decision (D1)	100% (8/8) decision outcomes
Condition (C1)	88% (7/8) condition outcomes
MCDC (C1)	75% (3/4) conditions reversed the outcome

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`)

  - 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` outcomes except 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 MATLAB Functions in an External File

Using the same model in "Model Coverage for MATLAB Function Blocks" on page 18-23, suppose the MATLAB functions `run_intersect_test` and `rect_intersect` are stored in an external MATLAB file named `run_intersect_test.m`.

To collect coverage for MATLAB functions in an external file, on the Coverage Settings dialog box, on the **Coverage** tab, select **Coverage for External MATLAB files**.

After simulation, the model coverage report summary contains sections for the top-level model and for the external function.
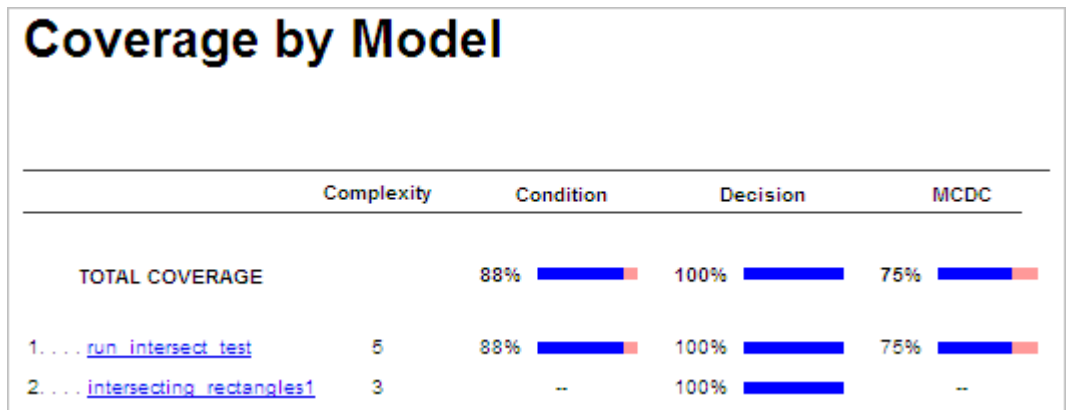
The model coverage report for `run_intersect_test.m` reports the same coverage data as if the functions were stored in the MATLAB Function block.

For a detailed example of a model coverage report for a MATLAB function in an external file, see "External MATLAB File Coverage Reports" on page 19-40.

Model Coverage for Simulink Design Verifier MATLAB Functions

If the MATLAB code includes any of the following Simulink Design Verifier functions configured for code generation, you can measure coverage:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

For this example, consider the following model that contains a MATLAB Function block.
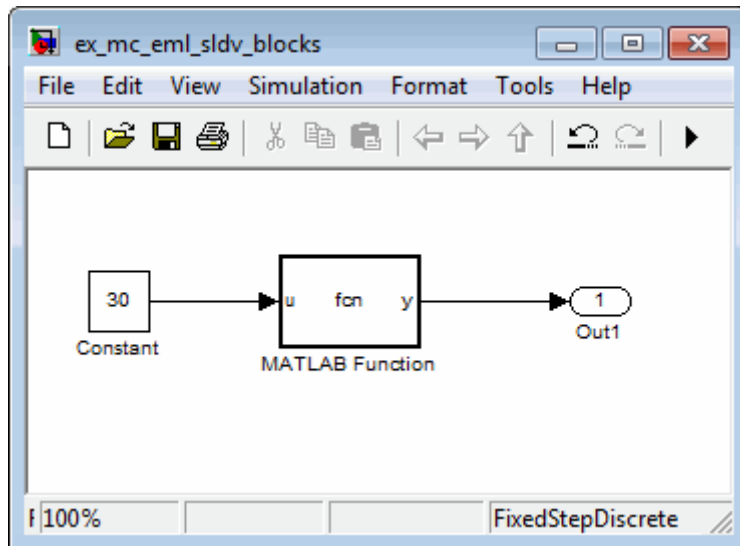
The MATLAB Function block contains the following code:

```
function y = fcn(u)
% This block supports MATLAB for code generation.

sldv.condition(u > -30)
sldv.test(u == 30)
y = 1;
```

To collect coverage for Simulink Design Verifier MATLAB functions, on the Coverage Settings dialog box, on the **Coverage** tab, select **Simulink Design Verifier**.

After simulation, the model coverage report listed coverage for the `sldv.condition` and `sldv.test` functions. For `sldv.condition`, the expression `u > -30` evaluated to `true` 51 times. For `sldv.test`, the expression `u == 30` evaluated to `true` 51 times.

eM Function "fcn"

Parent:	ex_mc_eml_sldv_blocks/MATLAB Function

Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	100% (1/1) decision outcomes
Test Objective	100% (1/1) objective outcomes
Test Condition	100% (1/1) objective outcomes

```
1   function y = fcn(u)
2   % This block supports MATLAB for code generation.
3
4   sldv.condition(u > -30)
5   sldv.test(u == 30)
6   y = 1;
```

#1: function y = fcn(u)

Decisions analyzed:

function y = fcn(u)	100%
executed	51/51

#4: sldv.condition(u > -30)

Test Condition analyzed:

sldv.condition(u > -30)	51/51

#5: sldv.test(u == 30)

Test Objective analyzed:

sldv.test(u == 30)	51/51

For an example of model coverage data for Simulink Design Verifier blocks, see "Simulink Design Verifier Coverage" on page 15-9.

Model Coverage for Stateflow Charts

How Model Coverage Reports Work for Stateflow Charts

To generate a Model Coverage report, select **Tools > Coverage Settings** and specify the desired options on the **Reporting** tab of the Coverage Settings dialog box. For Stateflow charts, the Simulink Verification and Validation software records the execution of the chart itself and the execution of states, transition decisions, and individual conditions that compose each decision. After simulation ends, the model coverage reports on how thoroughly a model was tested. The report shows:

- How many times each exclusive substate is entered, executed, and exited based on the history of the superstate

- How many times each transition decision has been evaluated as true or false

- How many times each condition has been evaluated as true or false

> **Note** To measure model coverage data for a Stateflow chart, you must have a Stateflow license.

For complete information about Stateflow software, see *Stateflow User's Guide*.

Specifying Coverage Report Settings

To specify coverage report settings, select **Tools > Coverage Settings** in a Simulink model window.

By selecting the **Generate HTML Report** option in the Coverage Settings dialog box, you can create an HTML report containing the coverage data generated during simulation of the model. The report appears in the MATLAB Help browser at the end of simulation.

By selecting the **Generate HTML Report** option, you also enable the selection of different coverages that you can specify for your reports. The following sections address only coverage metrics that affect reports for Stateflow charts. These metrics include decision coverage, condition coverage, and MCDC coverage. For a complete discussion of all dialog box fields and entries, see Chapter 17, "Setting Model Coverage Options".

Cyclomatic Complexity

Cyclomatic complexity is a measure of the complexity of a software module based on its edges, nodes, and components within a control-flow graph. It provides an indication of how many times you need to test the module.

The calculation of cyclomatic complexity is as follows:

```
CC = E - N + p
```

where `CC` is the cyclomatic complexity, `E` is the number of edges, `N` is the number of nodes, and `p` is the number of components.

Within the Model Coverage tool, each decision is exactly equivalent to a single control flow node, and each decision outcome is equivalent to a control flow

edge. Any additional structure in the control-flow graph is ignored since it contributes the same number of nodes as edges and therefore has no effect on the complexity calculation. Therefore, you can express cyclomatic complexity as follows:

```
CC = OUTCOMES - DECISIONS + p
```

For analysis purposes, each chart counts as a single component.

Decision Coverage

Decision coverage interprets a model execution in terms of underlying decisions where behavior or execution must take one outcome from a set of mutually exclusive outcomes.

Note Full coverage for an object of decision means that every decision has had at least one occurrence of each of its possible outcomes.

Decisions belong to an object making the decision based on its contents or properties. The following table lists the decisions recorded for model coverage for the Stateflow objects owning them. The sections that follow the table describe these decisions and their possible outcomes.

Object	Possible Decisions
Chart	If a chart is a triggered Simulink block, it must decide whether or not to execute its block.
	If a chart contains exclusive (OR) substates, it must decide which of its states to execute.
State	If a state is a superstate containing exclusive (OR) substates, it must decide which substate to execute.
	If a state has on *event name* actions (which might include temporal logic operators), the state must decide whether or not to execute the actions.
Transition	If a transition is a conditional transition, it must decide whether or not to exit its active source state or junction and enter another state or junction.

Chart as a Triggered Simulink Block Decision

If the chart is a triggered block in a Simulink model, the decision to execute the block is tested. If the block is not triggered, there is no decision to execute the block, and the measurement of decision coverage is not applicable (NA).

Chart Containing Exclusive OR Substates Decision

If the chart contains exclusive (OR) substates, the decision on which substate to execute is tested. If the chart contains only parallel AND substates, this coverage measurement is not applicable (NA).
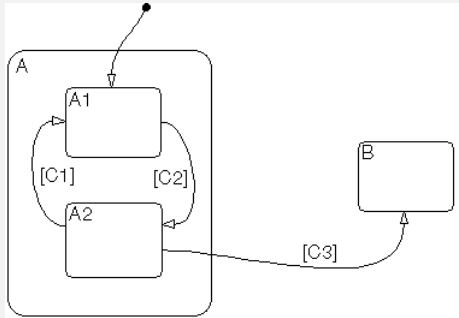
Superstate Containing Exclusive OR Substates Decision
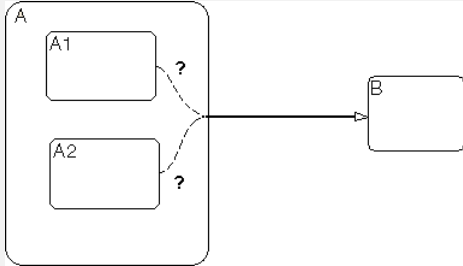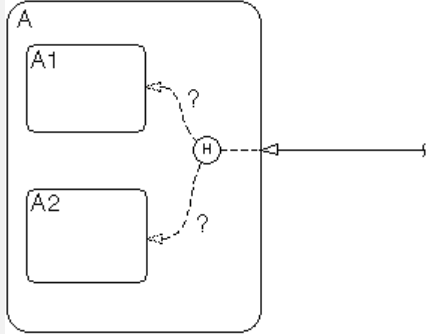
Since a chart is hierarchically processed from the top down, procedures such as exclusive (OR) substate entry, exit, and execution are sometimes decided by the parenting superstate.

Note Decision coverage for superstates applies only to exclusive (OR) substates. A superstate makes no decisions for parallel (AND) substates.

Since a superstate must decide which exclusive (OR) substate to process, the number of decision outcomes for the superstate is the number of exclusive (OR) substates that it contains. In the examples that follow, the choice of which substate to process can occur in one of three possible contexts.

Note Implicit transitions appear as dashed lines in the following examples.

Context	Example	Decisions That Occur
Active call	States A and A1 are active.	• The parent of states A and B must decide which of these states to process. This decision belongs to the parent. Since A is active, it is processed. • State A, the parent of states A1 and A2, must decide which of these states to process. This decision belongs to state A. Since A1 is active, it is processed. During processing of state A1, all outgoing transitions are tested. This decision belongs to the transition and not to the parent state A. In this case, the transition marked by condition C2 is tested and a decision is made whether to take the transition to A2 or not.
Implicit substate exit	A transition takes place whose source is superstate A and whose destination is state B.	If the superstate has two exclusive (OR) substates, it is the decision of superstate A which substate

Context	Example	Decisions That Occur
		performs the implicit transition from substate to superstate.
Substate entry with a history junction	A history junction records which substate was last active before the superstate was exited. 	If that superstate becomes the destination of one or more transitions, the history junction decides which previously active substate to enter.

For more information, see "State Details Report Section" on page 18-52.

State with On Event_Name Action Statement Decision

A state that has an on *event_name* action statement must decide whether to execute that statement based on the reception of a specified event or on an accumulation of the specified event when using temporal logic operators.

Conditional Transition Decision

A conditional transition is a transition with a triggering event and/or a guarding condition. In a conditional transition from one state to another, the decision to exit one state and enter another is credited to the transition itself.

Note Only conditional transitions receive decision coverage. Transitions without decisions are not applicable to decision coverage.

Condition Coverage

Condition coverage reports on the extent to which all possible outcomes are achieved for individual subconditions composing a transition decision.

Note Full condition coverage means that all possible outcomes occurred for each subcondition in the test of a decision.

For example, for the decision [A & B & C] on a transition, condition coverage reports on the true and false occurrences of each of the subconditions A, B, and C. This results in eight possible outcomes: true and false for each of three subconditions.

Outcome	A	B	C
1	T	T	T
2	T	T	F
3	T	F	T
4	T	F	F
5	F	T	T
6	F	T	F
7	F	F	T
8	F	F	F

For more information, see "Transition Details Report Section" on page 18-55.

MCDC Coverage

The Modified Condition Decision Coverage (MCDC) option reports a test's coverage of occurrences in which changing an individual subcondition within

a transition results in changing the entire transition trigger expression from true to false or false to true.

Note If matching true and false outcomes occur for each subcondition, coverage is 100%.

For example, if a transition executes on the condition [C1 & C2 & C3 | C4 & C5], the MCDC report for that transition shows actual occurrences for each of the five subconditions (C1, C2, C3, C4, C5) in which changing its result from true to false is able to change the result of the entire condition from true to false.

Simulink Design Verifier Coverage

You can use the following Simulink Design Verifier functions inside Stateflow charts:

- sldv.condition
- sldv.test
- sldv.assume
- sldv.prove

If you do not have a Simulink Design Verifier license, you can collect model coverage for a Stateflow chart containing these functions, but you cannot analyze the model using the Simulink Design Verifier software.

When you specify the **Simulink Design Verifier** coverage metric in the Coverage Settings dialog box, the Simulink Verification and Validation software records coverage for these functions.

Each of these functions evaluates an expression *expr*, for example, sldv.test(*expr*), where *expr* is any valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to true.

If *expr* is `true` for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Verification and Validation software reports coverage for that function as 0%.

Consider a model that contains this Stateflow chart:

To collect coverage for Simulink Design Verifier functions, on the Coverage Settings dialog box, on the **Coverage** tab, select **Simulink Design Verifier**.

After simulation, the model coverage report lists coverage for the `sldv.condition`, `sldv.assume`, `sldv.prove`, and `sldv.test` functions.

Metric	Coverage
Cyclomatic Complexity	0
Proof Assumption	100% (1/1) objective outcomes
Test Condition	100% (1/1) objective outcomes
Proof Objective	100% (1/1) objective outcomes
Test Objective	100% (1/1) objective outcomes

Proof Assumption analyzed:

sldv.assume(2==2)	1/1

Test Condition analyzed:

sldv.condition(1==1)	1/1

Proof Objective analyzed:

sldv.prove(3==3)	1/1

Test Objective analyzed:

sldv.test(4==4)	1/1

Model Coverage Reports for Stateflow Charts

- "Summary Report Section" on page 18-50
- "Subsystem and Chart Details Report Sections" on page 18-51
- "State Details Report Section" on page 18-52
- "Transition Details Report Section" on page 18-55

The following sections of a Model Coverage report were generated by simulating the sf_boiler model, which includes the Bang-Bang Controller chart. The coverage metrics for **Decision**, **Condition**, and **MCDC** are enabled for this report.

Summary Report Section

The Summary section shows coverage results for the entire test and appears at the beginning of the Model Coverage report.
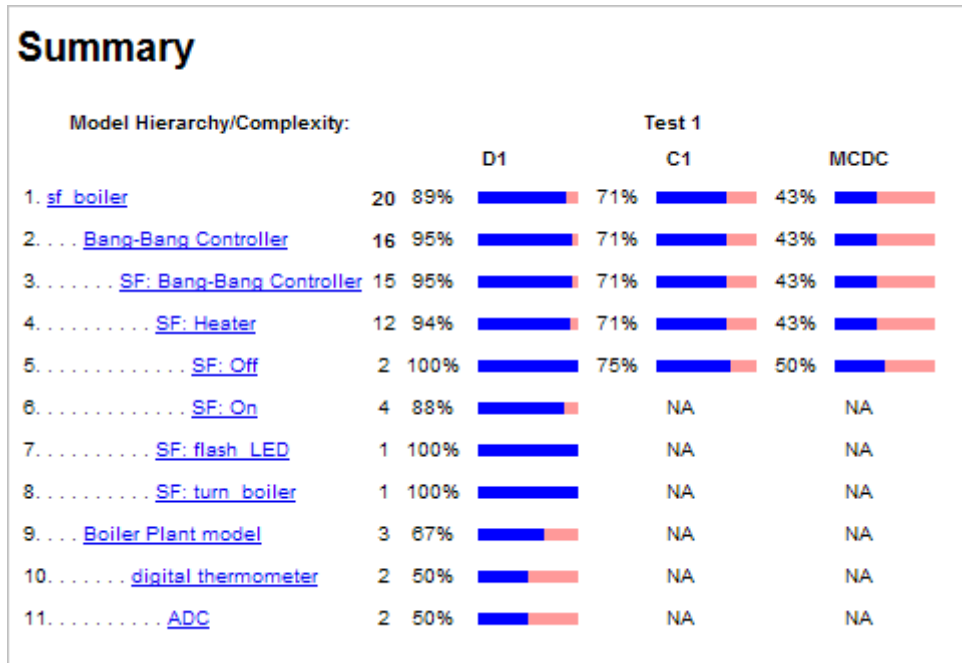
Each line in the hierarchy summarizes the coverage results at that level and the levels below it. You can click a hyperlink to a later section in the report with the same assigned hierarchical order number that details that coverage and the coverage of its children.

The top level, `sf_boiler`, is the Simulink model itself. The second level, Bang-Bang Controller, is the Stateflow chart. The next levels are superstates within the chart, in order of hierarchical containment. Each superstate uses an SF: prefix. The bottom level, Boiler Plant model, is an additional subsystem in the model.

Subsystem and Chart Details Report Sections

When recording coverage for a Stateflow chart, the Simulink Verification and Validation software reports two types of coverage for the chart—Subsystem and Chart.

- **Subsystem** — This section reports coverage for the chart:
  - **Coverage (this object)**: Coverage data for the chart as a container object
  - **Coverage (inc.) descendants**: Coverage data for the chart and the states and transitions in the chart.

  If you click the hyperlink of the subsystem name in the section title, the Bang-Bang Controller block is highlighted in the block diagram.

  Decision coverage is not applicable (NA) because this chart does not have an explicit trigger. Condition coverage and MCDC are not applicable (NA) for a chart, but apply to its descendants.

2. Subsystem "Bang-Bang Controller"

Parent:	/sf_boiler
Child Systems:	Bang-Bang Controller

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	16
Decision (D1)	NA	95% (21/22) decision outcomes
Condition (C1)	NA	71% (10/14) condition outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

- **Chart** — This section reports coverage for the chart:
  - **Coverage (this object)**: Coverage data for the chart and its inputs
  - **Coverage (inc.) descendants**: Coverage data for the chart and the states and transitions in the chart.

  If you click the hyperlink of the chart name in the section title, the chart opens in the Stateflow Editor.

  Decision coverage is listed appears for the chart and its descendants. Condition coverage and MCDC are not applicable (NA) for a chart, but apply to its descendants.

3. Chart "Bang-Bang Controller"

Parent:	sf_boiler/Bang-Bang Controller	
Child Systems:	Heater, flash_LED, turn_boiler	

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	15
Decision (D1)	100% (2/2) decision outcomes	95% (21/22) decision outcomes
Condition (C1)	NA	71% (10/14) condition outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

Decisions analyzed:

Substate executed	100%
State "Off"	1160/1400
State "On"	240/1400

State Details Report Section

For each state in a chart, the coverage report includes a **State** section with details about the coverage recorded for that state.

In the sf_boiler model, the state On resides in the box Heater. On is a superstate that contains:

- Two substates HIGH and NORM

- A history junction

- The function warm
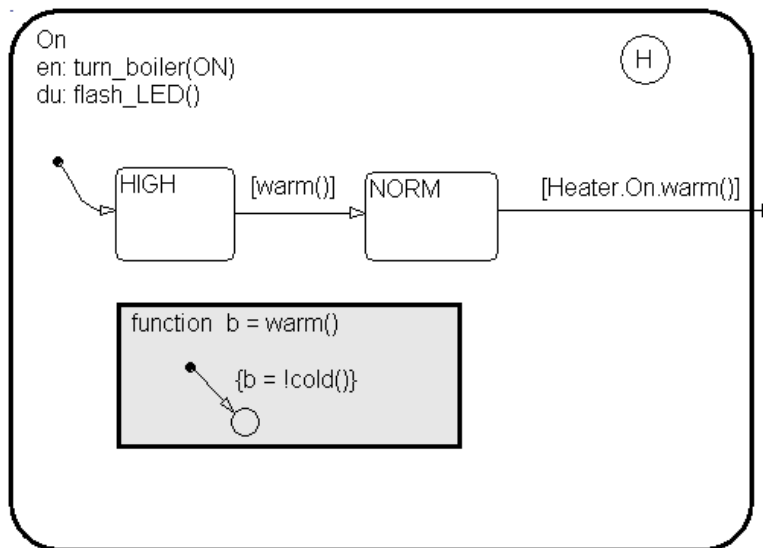
The coverage report includes a **State** section on the state On.

6. State "On"

Parent:	sf_boiler/Bang-Bang Controller.Heater	

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	3	4
Decision (D1)	83% (5/6) decision outcomes	88% (7/8) decision outcomes

Decisions analyzed:

Substate executed	100%
State "HIGH"	150/233
State "NORM"	83/233
Substate exited when parent exits	50%
State "HIGH"	7/7
State "NORM"	0/7
Previously active substate entered due to history	100%
State "HIGH"	7/28
State "NORM"	21/28

The decision coverage for the On state tests the decision of which substate to execute.

The three decisions are listed in the report:

- Under **Substate executed**, which substate to execute when On executes.

- Under **Substate exited when parent exited**, which substate is active when On exits. NORM is listed as never being active when On exits because the coverage tool sees the supertransition from NORM to Off as a transition from On to Off.

- Under **Previously active substate entered due to history**, which substate to reenter when On re-executes. The history junction records the previously active substate.

Because each decision can result in either HIGH or NORM, the total possible outcomes are $3 \times 2 = 6$. The results indicate that five of six possible outcomes were tested during simulation.

Cyclomatic complexity and decision coverage also apply to descendants of the On state. The decision required by the condition [warm()] for the transition from HIGH to NORM brings the total possible decision outcomes to 8. Condition coverage and MCDC are not applicable (NA) for a state.

Note Nodes and edges that make up the cyclomatic complexity calculation have no direct relationship with model objects (states, transitions, and so on). Instead, this calculation requires a graph representation of the equivalent control flow.

Transition Details Report Section

Reports for transitions appear under the report sections of their owning objects. Transitions do not appear in the model hierarchy of the Summary section, since the hierarchy is based on superstates that own other Stateflow objects.

Transition "after(40,sec) [cold()]" from "Off" to "On"

Parent:	sf_boiler/Bang-Bang Controller.Heater
Uncovered Links:	← →

Metric	Coverage
Cyclomatic Complexity	3
Decision (D1)	100% (2/2) decision outcomes
Condition (C1)	67% (4/6) condition outcomes
MCDC (C1)	33% (1/3) conditions reversed the outcome

Decisions analyzed:

Transition trigger expression	100%
false	1131/1160
true	29/1160

Conditions analyzed:

Description:	True	False
Condition 1, "sec"	1160	0
Condition 2, "after(40,sec)"	29	1131
Condition 3, "cold()"	29	0

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

Decision/Condition:	True Out	False Out
Transition trigger expression		
Condition 1, "sec"	TTT	(Fxx)
Condition 2, "after(40,sec)"	TTT	TFx
Condition 3, "cold()"	TTT	(TTF)

The decision for this transition depends on the time delay of 40 seconds and the condition `[cold()]`. If, after a 40 second delay, the environment is cold (`cold() = 1`), the decision to execute this transition and turn the Heater on is made. For other time intervals or environment conditions, the decision is made not to execute.

For decision coverage, both true and false outcomes occurred. Because two of two decision outcomes occurred, coverage was full or 100%.

Condition coverage shows that only 4 of 6 condition outcomes were tested. The temporal logic statement `after(40,sec)` represents two conditions:

the occurrence of `sec` and the time delay `after(40,sec)`. Therefore, three conditions on the transition exist: `sec`, `after(40,sec)`, and `cold()`. Since each of these decisions can be true or false, six possible condition outcomes exist.

The **Conditions analyzed** table shows each condition as a row with the recorded number of occurrences for each outcome (true or false). Decision rows in which a possible outcome did not occur are shaded. For example, the first and the third rows did not record an occurrence of a false outcome.

In the MC/DC report, all sets of occurrences of the transition conditions are scanned for a particular pair of decisions for each condition in which the following are true:

- The condition varies from true to false.
- All other conditions contributing to the decision outcome remain constant.
- The outcome of the decision varies from true to false, or the reverse.

For three conditions related by an implied AND operator, these criteria can be satisfied by the occurrence of these conditions.

Condition Tested	True Outcome	False Outcome
1	**T**TT	**F**xx
2	T**T**T	T**F**x
3	TT**T**	TT**F**

Notice that in each line, the condition tested changes from true to false while the other condition remains constant. Irrelevant contributors are coded with an "x" (discussed below). If both outcomes occur during testing, coverage is complete (100%) for the condition tested.

The preceding report example shows coverage only for condition 2. The false outcomes required for conditions 1 and 3 did not occur, and are indicated by parentheses for both conditions. Therefore, condition rows 1 and 3 are shaded. While condition 2 has been tested, conditions 1 and 3 have not and MCDC is 33%.

For some decisions, the values of some conditions are irrelevant under certain circumstances. For example, in the decision [C1 & C2 & C3 | C4 & C5] the left side of the | is false if any one of the conditions C1, C2, or C3 is false. The same applies to the right side result if either C4 or C5 is false. When searching for matching pairs that change the outcome of the decision by changing one condition, holding some of the remaining conditions constant is irrelevant. In these cases, the MC/DC report marks these conditions with an "x" to indicate their irrelevance as a contributor to the result. These conditions appear as shown.

Transition "[c1&c2&c3 | c4&c5]" . . .

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

Decision/Condition:	#1 True Out	#1 False Out
Transition trigger expression		
Condition 1, "c1"	TTTxx	FxxFx
Condition 2, "c2"	TTTxx	TFxFx
Condition 3, "c3"	TTTxx	TTFFx
Condition 4, "c4"	FxxTT	FxxFx
Condition 5, "c5"	FxxTT	FxxTF

Consider the first matched pair. Since condition 1 is true in the **True** outcome column, it must be false in the matching **False** outcome column. This makes the conditions C2 and C3 irrelevant for the false outcome since C1 & C2 & C3 is always false if C1 is false. Also, since the false outcome is required to evaluate to false, the evaluation of C4 & C5 must also be false. In this case, a match was found with C4 = F, making condition C5 irrelevant.

Model Coverage for Stateflow Atomic Subcharts

In a Stateflow chart, an atomic subchart is a graphical object that allows you to reuse the same state or subchart across multiple charts and models.

When you specify to record coverage data for a model during simulation, the Simulink Verification and Validation software records coverage for any atomic subcharts in your model. The coverage data records the execution of the

chart itself, and the execution of states, transition decisions, and individual conditions that compose each decision in the atomic subchart.

Simulate the `doc_atomic_subcharts_map_iodata` example model and record decision coverage:

1 Open the `doc_atomic_subcharts_map_iodata.mdl` model.

  This model contains two Sine Wave blocks that supply input signals to the Stateflow chart Chart. Chart contains two atomic subcharts—A and B—that are linked from the same library chart, also named A. The library chart contains the following objects:
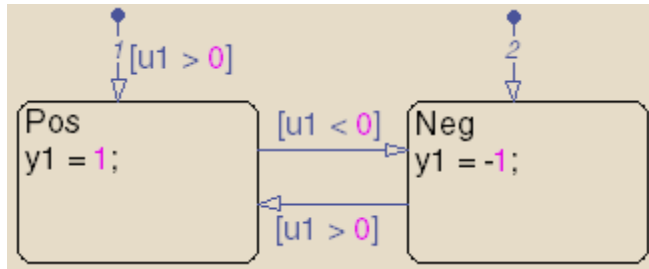
2 In the Model Editor, select **Tools > Coverage Settings**

  The Coverage Settings dialog box appears.

3 On the **Coverage** tab, select **Coverage for this model: doc_atomic_subcharts_map_iodata**.

4 On the **Reporting** tab, select **Generate HTML report**.

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

6 Simulate the `doc_atomic_subcharts_map_iodata` model.

  When the simulation completes, the coverage report opens.

The report provides coverage data for atomic subcharts A and B in the following forms:

- For the atomic subchart instance and its contents. Decision coverage is not applicable (NA) because this chart does not have an explicit trigger.

4. Atomic Subchart "A"

Parent:	doc_atomic_subcharts_map_iodata/Chart
Child Systems:	A

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	4
Decision (D1)	NA	88% (7/8) decision outcomes

- For the library chart A and its contents. The chart itself achieves 100% coverage on the input u1, and 88% coverage on the states and transitions inside the library chart.

5. Chart "A"

Parent:	doc_atomic_subcharts_map_iodata/Chart.A

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	4
Decision (D1)	100% (2/2) decision outcomes	88% (7/8) decision outcomes

Decisions analyzed:

Substate executed	100%
State "Neg"	4/10
State "Pos"	6/10

Atomic subchart B is a copy of the same library chart A. The coverage of the contents of subchart B is identical to the coverage of the contents of subchart A.

Model Coverage for Stateflow Truth Tables

- "Types of Coverage in Stateflow Truth Tables" on page 18-61
- "Analyzing Coverage in Stateflow Truth Tables" on page 18-62

Types of Coverage in Stateflow Truth Tables

Simulink Verification and Validation software reports model coverage for the decisions the objects make in a Stateflow chart during model simulation. The report includes coverage for the decisions the truth table functions make.

For this type of truth table...	The report includes coverage data for...
Stateflow Classic	Conditions only.
MATLAB	Conditions and only those actions that have decision points. **Note** With the MATLAB for code generation action language, you can specify decision points in actions using control flow constructs, such as loops and switch statements.

Note To measure model coverage data for a Stateflow truth table, you must have a Stateflow license.

For detailed information about truth tables and truth table functions in Stateflow, see "Truth Table Functions for Decision-Making Logic".

Analyzing Coverage in Stateflow Truth Tables

If you have a Stateflow license, you can generate a model coverage report for a truth table.

Consider the following model.
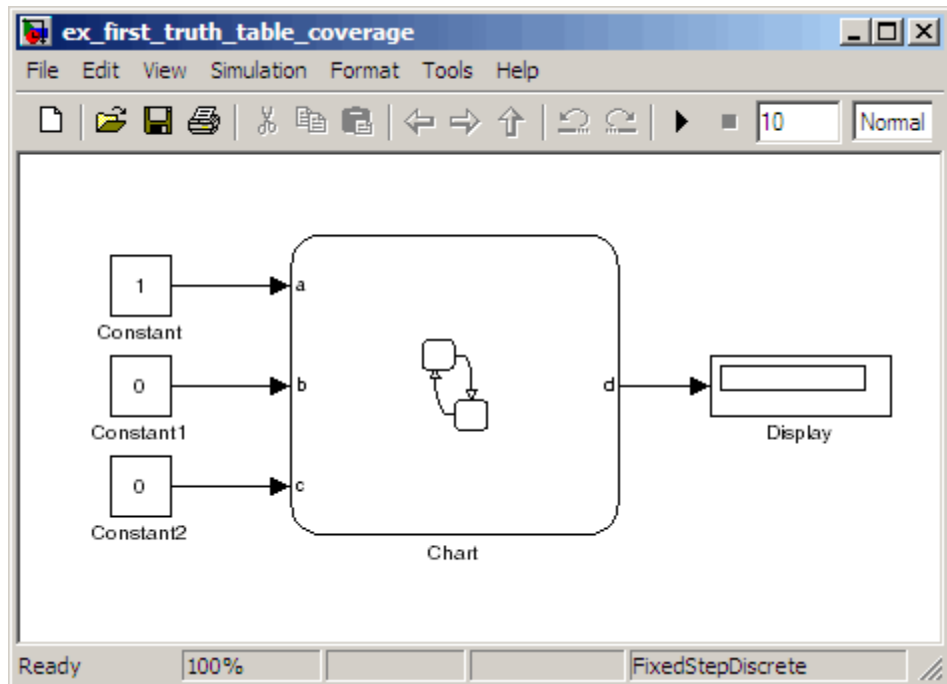
The Stateflow chart Chart contains the following truth table:

Stateflow (truth table) first_truth_table/Chart.ttable

File   Edit   Settings   Add   Help

Condition Table

	Description		Condition	D1	D2	D3	D4	D5	
1	x is equal to 1		XEQ1: x == 1	T	F	F	–	–	
2	y is equal to 1		YEQ1: y == 1	F	T	F	–	–	
3	z is equal to 1		ZEQ1: z == 1	F	F	F	T	–	
			Actions: Specify a row from the Action Table	A1	A2	A3	A4	A5	

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1 Maintain a counter and a circular vector	A1: persistent values counter; cycle = 6;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	set t to 5	A5: t=5;
7	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

When you simulate the model and collect coverage, the model coverage report includes the following data:

4. Truth Table "ttable"

Parent: first_truth_table/Chart

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	10
Decision (D1)	NA	13% (1/8) decision outcomes
Condition (C1)	NA	17% (3/18) condition outcomes
MCDC (C1)	NA	0% (0/9) conditions reversed the outcome

Condition table analysis (missing values are in parentheses)

x is equal to 1	XEQ1: x == 1	T (F)	F (TF)	F (TF)	-	-
y is equal to 1	YEQ1: y == 1	F (T)	T (TF)	F (TF)	-	-
z is equal to 1	ZEQ1: z == 1	F (T)	F (TF)	T (TF)	T (TF)	-
	Actions	A1 (F)	A2 (TF)	A3 (TF)	A4 (TF)	A5

The **Coverage (this object)** column shows no coverage. The reason is that the container object for the truth table function—the Stateflow chart—does not decide whether to execute the `ttable` truth table.

The **Coverage (inc. descendants)** column shows coverage for the graphical function. The graphical function has the decision logic that makes the

transitions for the truth table. The transitions in the graphical function contain the decisions and conditions of the truth table. Coverage for the descendants in the **Coverage (inc. descendants)** column includes coverage for these conditions and decisions. Function calls to the truth table test the model coverage of these conditions and decisions.

Note See "How Stateflow Software Generates Content for Truth Tables" for a description of the graphical function for a truth table.

Coverage for the decisions and their individual conditions in the `ttable` truth table function are as follows.

Coverage	Explanation
No model coverage for the default decision, **D5**	All logic that leads to taking a default decision is based on a false outcome for all preceding decisions. This means that the default decision requires no logic, so there is no model coverage.
13% (1/8) decision coverage	The three constants that are inputs to the truth table (1, 0, 0) cause only decision **D1** to be true. These inputs satisfy only one of the eight decisions (**D1** through **D4**, T or F).
	Because each condition can have an outcome value of T or F, three conditions can have six possible values. However, decision **D4** has only decision coverage, not condition coverage or MCDC coverage, because it represents a decision with a single predicate.
3 of the 18 (17%) condition coverage	Three decisions **D1**, **D2**, and **D3** have condition coverage, because the set of inputs (1, 0, 0) make only decision **D1** true.

Coverage	Explanation
No (0/9) MCDC coverage	MCDC coverage looks for decision reversals that occur because one condition outcome changes from T to F or F to T. The simulation tests only one set of inputs, so the model reverses no decisions.
Missing coverage	The red letters T and F indicate that model coverage is missing for those conditions. For decision **D1**, only the T decision is satisfied. For decisions **D2**, **D3**, and **D4**, none of the conditions are satisfied.

Colored Stateflow Chart Coverage Display

The Model Coverage tool displays model coverage results for individual blocks directly in Simulink diagrams. If you enable this feature, the Model Coverage tool:

- Highlights Stateflow objects that receive model coverage during simulation

- Provides a context-sensitive display of summary model coverage information for each object

Caution The coverage tool changes colors only for open charts at the time coverage information is reported. When you interact with the chart, such as selecting a transition or a state, colors revert to default values.

For details on enabling and selecting this feature in the Simulink window, see "Enabling Coverage Highlighting" on page 18-6 in the Simulink Verification and Validation documentation.

Displaying Model Coverage with Model Coloring

Once you enable display coverage with model coloring, anytime that the model generates a model coverage report, individual chart objects receiving coverage appear highlighted with light green or light red.

1 Open the sf_car model.

2 Select **Tools > Coverage Settings**.

3 In the Coverage Settings dialog box, select **Coverage for this model**.

4 Click **OK**.

5 Simulate the model.

After simulation ends, chart objects with coverage appear highlighted.
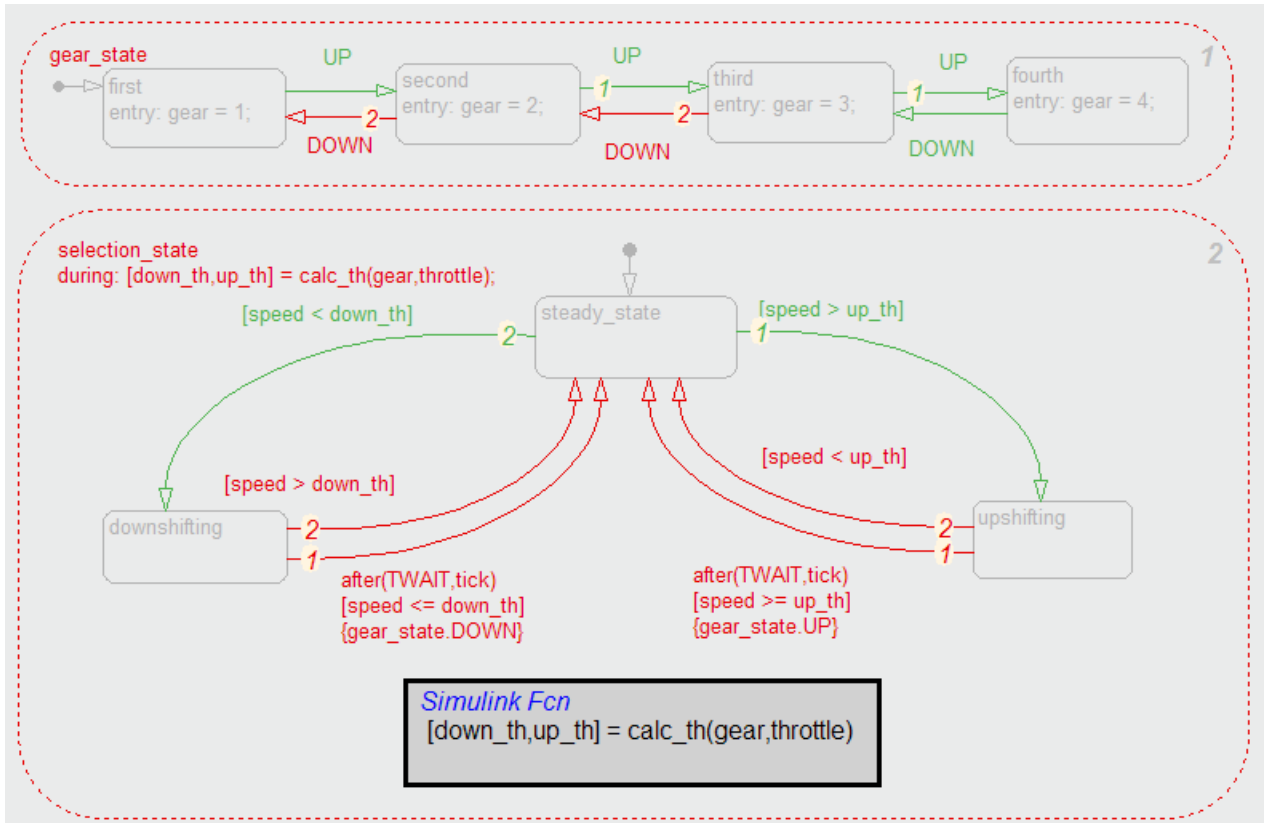
Object highlighting indicates coverage as follows:

- Light green for full coverage
- Light red for partial coverage
- No color for zero coverage

Note To revert the chart to show original colors, select and deselect any objects.

6 Click selection_state in the chart.

The following summary report appears.
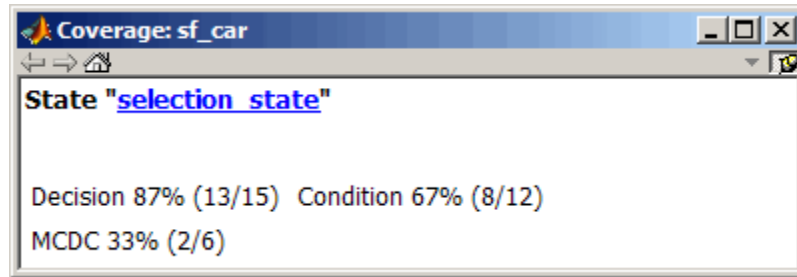
When you click a highlighted Stateflow object, the summarized coverage for that object appears in the Coverage Display Window. Clicking the hyperlink opens the section of the coverage report for this object.

Tip You can set the Coverage Display Window to appear for a block in response to a hovering mouse cursor instead of a mouse click in one of two ways:

- Select the downward arrow on the right side of the Coverage Display Window and select **Focus**.

- Right-click a colored block and select **Coverage > Display details on mouse-over**.

19

Results Review

Types of Coverage Reports

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

Report Type	Description	HTML Report File Name
"Model Coverage Reports" on page 19-3	Provides coverage information for all model elements, including the model itself.	*model_name*_cov.html
"Model Summary Reports" on page 19-38	Provides links to coverage results for all referenced models and external MATLAB files in the model hierarchy. Created when the top-level model includes Model blocks or calls one or more external files.	*model_name* _summary_cov.html
"Model Reference Coverage Reports" on page 19-39	Created for each referenced model in the model hierarchy; has the same format as the model coverage report.	*reference_model_name* _cov.html
"External MATLAB File Coverage Reports" on page 19-40	Provides detailed coverage information about any external MATLAB file that the model calls. There is one report for each external file called from the model.	*MATLAB_file_name*_cov.html
"Subsystem Coverage Reports" on page 19-45	Model coverage report includes only coverage results for the subsystem, if you select one.	*model_name*_cov.html; *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

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, in the Summary subsection, click the subsystem name.

Coverage Report for sldemo_fuelsys

Model Information

Model Version	1.533
Author	The MathWorks Inc.
Last Saved	Wed Sep 21 00:43:33 2011

Simulation Optimization Options

Inline Parameters	on
Block Reduction	forced off
Conditional Branch Optimization	on

Coverage Options

Analyzed model	sldemo_fuelsys/fuel_rate_control/control_logic
Logic block short circuiting	off

Tests

Test 1

Started Execution: 11-Oct-2011 15:47:27
Ended Execution: 11-Oct-2011 15:47:56

Summary

Model Hierarchy/Complexity:		Test 1 D1	
1. control_logic	56	25%	
2. . . . Pressure.map_estimate	1	0%	
3. . . . Speed.speed_estimate	3	0%	
4. . . . Throttle.throttle_estimate	1	0%	
5. . . . SF: control_logic	50	27%	
6. . . . . . . SF: Fail	12	9%	
7. . . . . SF: Multi	6	0%	
8. . . . . . . SF: Fueling_Mode	19	28%	
9. . . . . SF: Fuel_Disabled	4	0%	
10. . . . . SF: Running	10	35%	
11. . . . . SF: Low_Emissions	4	50%	

19-5

Details

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

- "Filtered Objects" on page 19-6
- "Model Details" on page 19-7
- "Subsystem Details" on page 19-8
- "Block Details" on page 19-9
- "Chart Details" on page 19-10
- "Coverage Details for MATLAB Functions and Simulink® Design Verifier™ Functions" on page 19-11

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

1 Right-click a Simulink element.

2 In the context menu, select **Coverage > Report**.

Filtered Objects

The Filtered Objects section lists all the objects in the model that were filtered from coverage recording, and the rationale you specified for filtering those objects. If the filter rule specifies that all blocks of a certain type be filtered, all those blocks are listed here.

In the following graphic, several blocks, subsystems, and transitions were filtered. Two library-linked blocks, protected division and protected division1, were filtered because their block library was filtered.

Filtered Objects

Objects filtered from coverage analysis

Model Object	Rationale
Subsystem "<u>Switchable config</u>"	Subsystem is never executed
Subsystem "<u>protected division</u>"	Protection against division by zero
Subsystem "<u>protected division1</u>"	Protection against division by zero
Saturate block "<u>Saturation</u>"	Input is never
Transition "<u>after(4, tick)</u>" from "<u>Clipped</u>" to "<u>Full</u>"	tick is never false
Transition "<u>[Ion]</u>" from Junction #0 to "<u>off</u>"	This transition is never evaluated

Model Details

The Details section contains a results summary for the model as a whole, followed by a list of elements. 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 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

The **Uncovered Links** element first appears in the Block Details section of the first block in the model hierarchy that does not achieve 100% coverage.

The first **Uncovered Links** element has an arrow that links to the Block Details section in the report of the *next* block that does not achieve 100% coverage.

Subsequent blocks that do not achieve 100% coverage have links to the Block Details sections in the report of the previous and next blocks that do not achieve 100% coverage.

Chart Details

The following graphic shows the coverage results for the Stateflow chart, Chart2, in the mExternalMfile model.

For more information about model coverage reports for Stateflow charts and their objects, see "Model Coverage for Stateflow Charts" on page 18-40 in the Stateflow documentation.

Coverage Details for **MATLAB** Functions and Simulink Design Verifier Functions

By default, the Simulink Verification and Validation software records coverage for all MATLAB functions in a model. MATLAB functions are in MATLAB Function blocks, Stateflow charts, or external MATLAB files.

Note For a detailed example of coverage reports for external MATLAB files, see "External MATLAB File Coverage Reports" on page 19-40.

To record Simulink Design Verifier coverage for sldv.* functions called by MATLAB functions, and any Simulink Design Verifier blocks, in the Coverage Settings dialog box, on the **Coverage** tab, select **Simulink Design Verifier**.

The following example shows coverage details for a MATLAB function, hFcnsInExternalEML, that calls four Simulink Design Verifier functions. In this example, the code for hFcnsInExternalEML resides in an external file.

This example also shows Simulink Design Verifier coverage details for the following functions:

- sldv.assume
- sldv.condition
- sldv.prove
- sldv.test

In the coverage results, code that achieves 100% coverage is green; code that achieves less than 100% coverage is red.

Embedded MATLAB function "**hfcnsinexternaleml**"

Parent: hfcnsinexternaleml
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	4
Decision (D1)	40% (2/5) decision outcomes
Test Objective	50% (1/2) objective outcomes
Proof Objective	0% (0/1) objective outcomes
Test Condition	100% (1/1) objective outcomes
Proof Assumption	0% (0/1) objective outcomes

```
 1   function y = hFcnsInExternalEML(u1, u2)
 2   % use all four functions.
 3   %#eml
 4   sldv.assume(u1 > u2);
 5   sldv.condition(u1 == 0);
 6   switch u1
 7       case 0
 8           y = u2;
 9       case 1
10           y = 3;
11       case 2
12           y = 0;
13       otherwise
14           y = 0;
15           sldv.prove(u2 < u1);
16   end
17   sldv.test(y > u1); sldv.test(y == 4);
18
```

Coverage for the hFcnsInExternalEML function and the sldv.* calls is:

- Line 1, the function declaration for hFcnsInExternalEMLis green because the simulation executes that function at least once. fcn calls hFcnsInExternalEML 11 times during simulation.

#1: function y = hFcnsInExternalEML(u1, u2)

Decisions analyzed:

function y = hFcnsInExternalEML(u1, u2)	100%
executed	11/11

Line 4, sldv.assume(u1 > u2), achieves 0% coverage because u1 > u2 never evaluates to true.

#4: sldv.assume(u1 > u2);

Proof Assumption analyzed:

sldv.assume(u1 > u2)	0/11

- Line 5, sldv.condition(u1 == 0), achieves 100% coverage because u1 == 0 evaluates to true for at least one time step.

#5: sldv.condition(u1 == 0);

Test Condition analyzed:

sldv.condition(u1 == 0)	11/11

- Line 6, switch u1, achieves 25% coverage because only one of the four outcomes in the switch statement (case 0) occurs during simulation.

- Line 17, `sldv.test(y > u1); sldv.test (y == 4)` achieves 50% coverage. The first `sldv.test` call achieves 100% coverage, but the second `sldv.test` call achieves 0% coverage.

For more information about coverage for MATLAB functions, see "Model Coverage for MATLAB Functions" on page 18-20.

For more information about coverage for Simulink Design Verifier functions, see "Simulink Design Verifier Coverage" on page 15-9.

Cyclomatic Complexity

You can specify that the model coverage report include cyclomatic complexity numbers in two locations in the report:

- The Summary section contains the cyclomatic complexity numbers for each object in the model hierarchy. For a subsystem or Stateflow chart, that number includes the cyclomatic complexity numbers for all their descendants.

Summary

Model Hierarchy/Complexity:

1. fuelsys	83
2. . . . EGO sensor	1
3. . . . MAP sensor	1
4. . . . engine speed	1
5. . . . engine gas dynamics	5
6. . . . . . . Mixing & Combustion	1
7. . . . . . . Throttle & Manifold	4
8. . . . . Throttle	2
9. . . . fuel rate controller	72

- The Details sections for each object list the cyclomatic complexity numbers for all individual objects.

7. Subsystem "Throttle & Manifold"

Parent:	fuelsys/engine gas dynamics
Child Systems:	Throttle

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	4
Decision (D1)	NA	63% (5/8) decision outcomes

Saturate block "Limit to Positive"

Parent:	fuelsys/engine gas dynamics/Throttle & Manifold

Metric	Coverage
Cyclomatic Complexity	2
Decision (D1)	50% (2/4) decision outcomes

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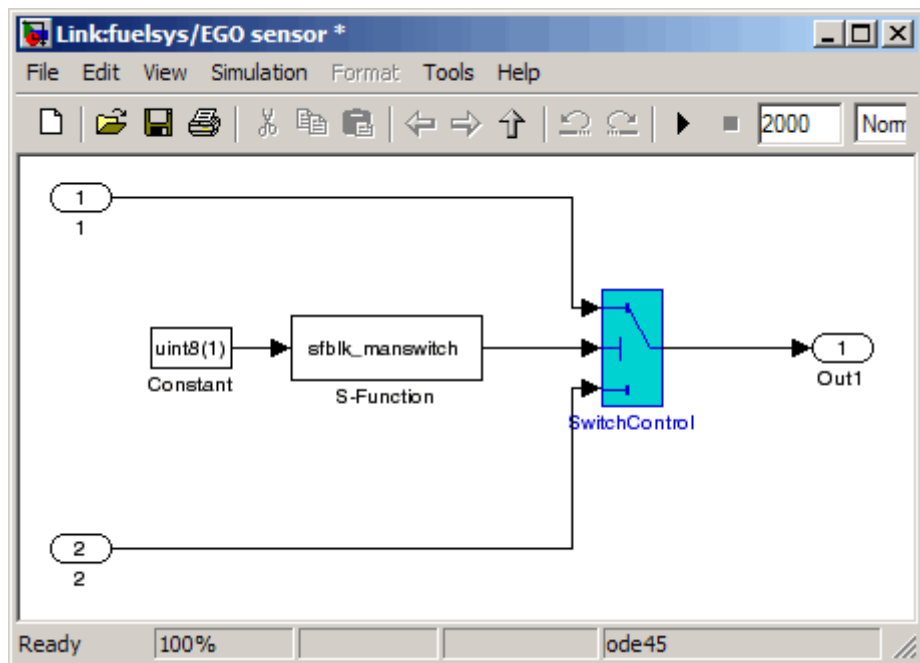
The next graphic shows the Decisions analyzed table for the `lib_em2` function call in `Chart2` of the `MexternalMfile` model.

#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

MCDC Analysis

The MC/DC analysis table lists the MCDC 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. Input n is called the *deciding input* of the

condition case. Changing the value of input n alone changes the value of the block's output.

The MC/DC analysis table shows 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, F**T**F 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, **F**F, to express the values of all the inputs to the block, with the value of the deciding variable 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, MC/DC coverage analysis does not verify whether short-circuited inputs actually occur. The MC/DC analysis table uses an x in a condition expression (for example, T**F**xxx) to indicate short-circuited inputs that were not analyzed by the tool.

If you enable this feature and Logic blocks are short-circuited while collecting model coverage, you may not be able to achieve 100% coverage for that block.

Uncovered Links. The section for each block that did not achieve 100% coverage contains a backward and a forward arrow. Click the forward arrow to go to the next section in the report that describes a block that did not achieve 100% coverage. Click the back arrow to return to the previous section in the report that describes a block that did not achieve 100% coverage.

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 results located in the right-most area 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 coverage added to the cumulative coverage achieved with the simulation just completed. If the previous simulation's cumulative coverage and the current coverage are nonzero, the delta may be 0 if the new coverage does not add to the cumulative coverage.

- Cumulative — The total coverage collected for the model up to, but not including, the simulation just 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	1097/1098	1097/1097	1097/1100
true	1/1098	0/1097	3/1100

The Conditions analyzed table uses column headers **#n T** and **#n F** to indicate results for individual test cases. The table uses **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, "alt_ctrl"	1	1097	0	1097	3	1097
Condition 2, "wow"	0	1	0	0	0	3
Condition 3, "in(GS.Active.Coupled)"	0	1	0	0	0	3

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, "alt_ctrl"	TFF	Fxx	(TFF)	Fxx	TFF	Fxx
Condition 2, "wow"	TFF	(TTx)	(TFF)	(TTx)	TFF	(TTx)
Condition 3, "in(GS.Active.Coupled)"	TFF	(TFT)	(TFF)	(TFT)	TFF	(TFT)

Note You can calculate cumulative coverage for reusable subsystems and Stateflow constructs at the command line. For more information, see "Creating Cumulative Coverage for Reusable Subsystems and Stateflow Constructs" on page 21-9.

N-Dimensional Lookup Table

The following interactive chart summarizes the extent to which elements of a lookup table are accessed. In this example, two Sine Wave blocks generate x and y indices that access a 2-D Lookup Table block of 10-by-10 elements filled with random values.
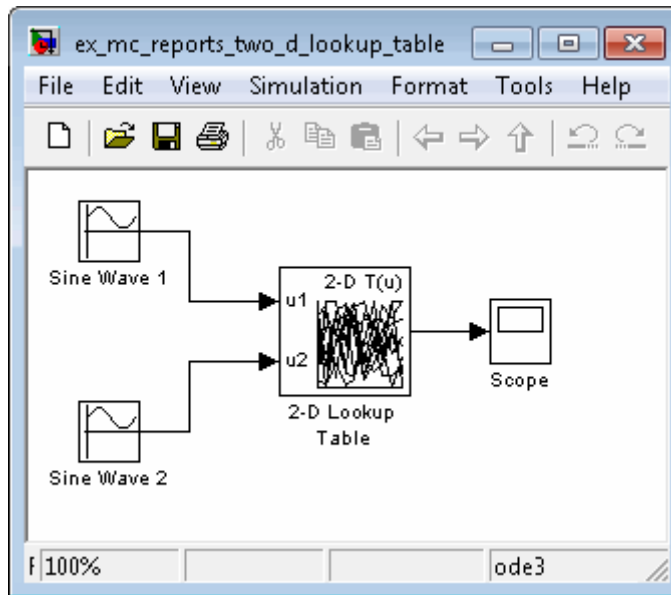
In this model, the lookup table indices are 1, 2,..., 10 in each direction. The Sine Wave 2 block is out of phase with the Sine Wave 1 block by pi/2 radians. This generates x and y numbers for the edge of a circle, which you see when you examine the resulting Lookup Table coverage.
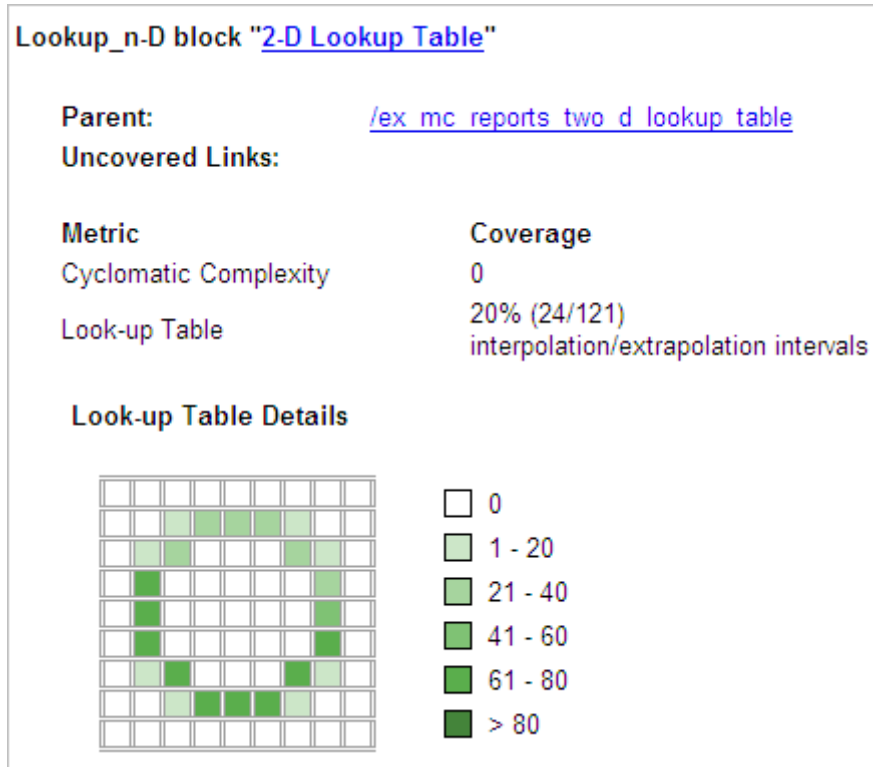
The report contains a two-dimensional table representing the elements of the lookup table. The element indices are represented by the cell border grid lines, which number 10 in each dimension. Areas where the lookup table interpolates between table values are represented by the cell areas. Areas of extrapolation 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 one 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.
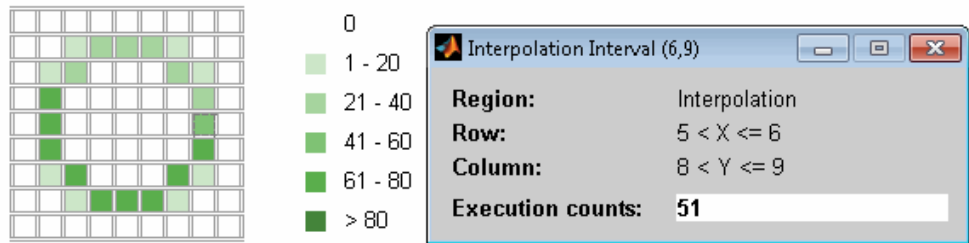
The selected cell is outlined in red. You can also click the extrapolation cells on the edge of the table.
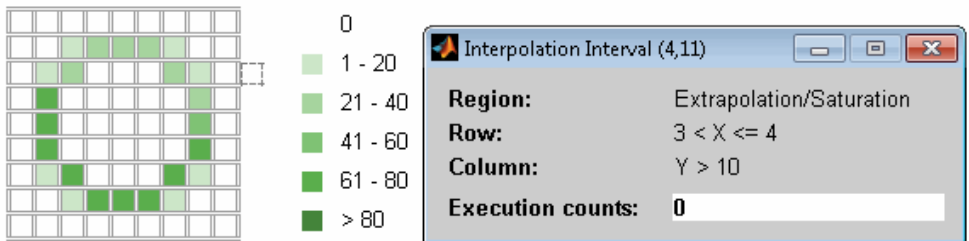
A bold grid line indicates that at least one block input equal to its exact index value occurred during the simulation. Click the border to display the exact number of hits for that index value.
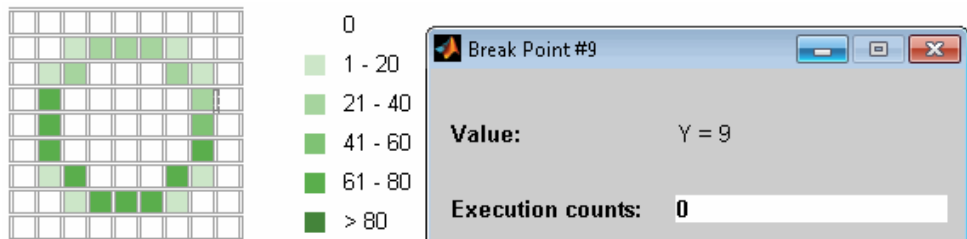
The following example model uses an n-D Lookup Table block of 10-by-10-by-5 elements filled with random values.
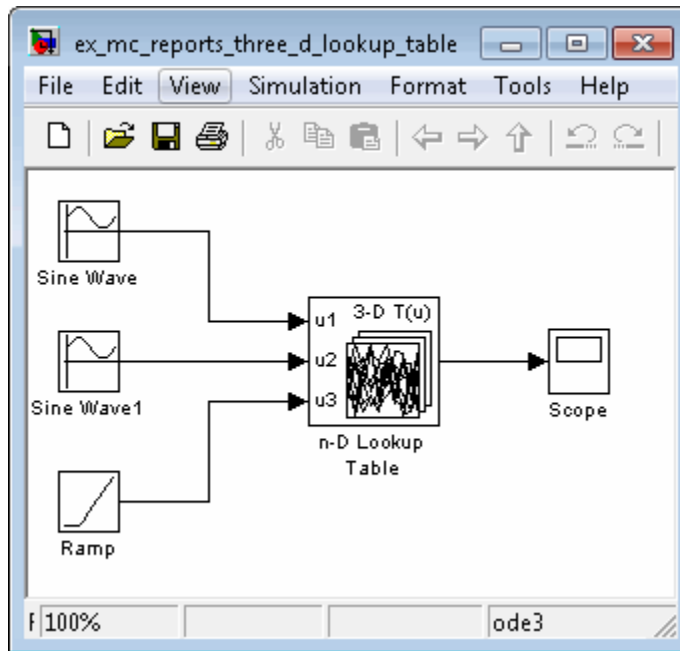
Both the *x* and *y* table axes have the indices 1, 2,..., 10. The *z* axis has the indices 10, 20,..., 50. Lookup table values are accessed with *x* and *y* indices that the two Sine Wave blocks generated, in the preceding example, and a *z* index that a Ramp block generates.

After simulation, you see the following lookup table report.

Lookup_n-D block "n-D Lookup Table"

Parent:	/ex_mc_reports_three_d_lookup_table
Uncovered Links:	

Metric	Coverage
Cyclomatic Complexity	0
Look-up Table	6% (42/726) interpolation/extrapolation intervals

Table map was not generated due to the table size.
Force Map Generation.

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

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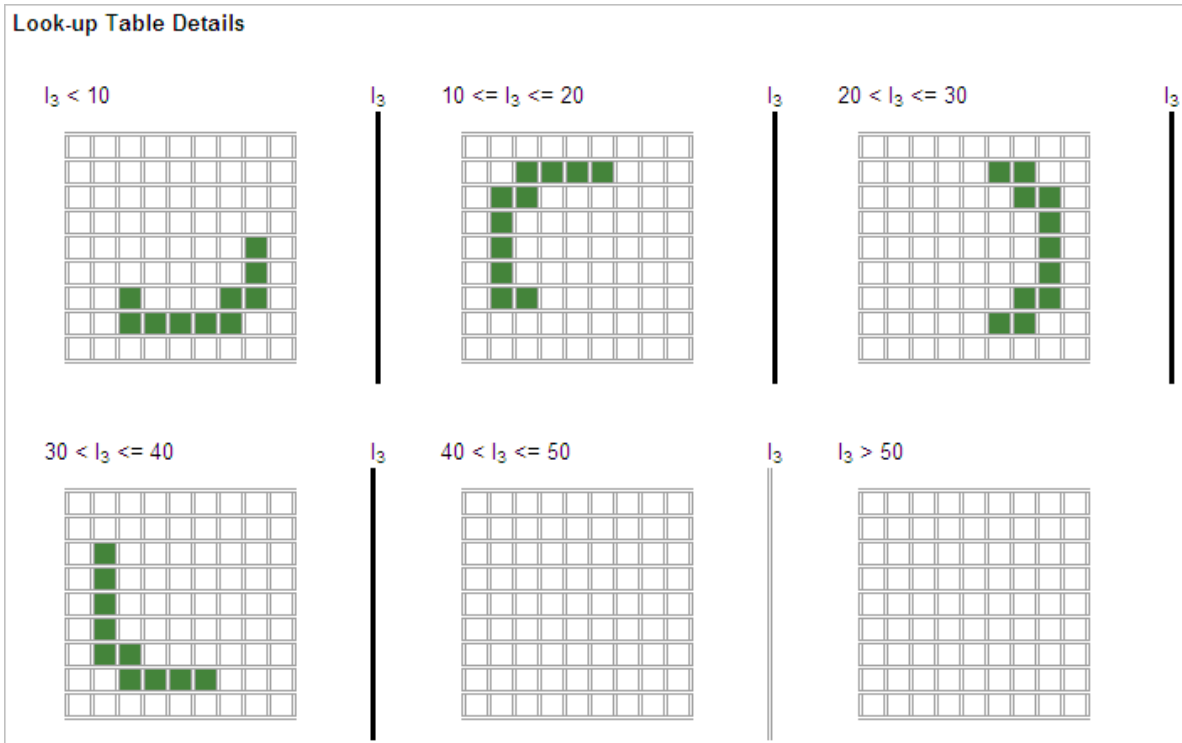
Lookup table coverage for a three-dimensional lookup table block is reported as a set of two-dimensional tables.

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 coverage report for the exact index value that bar 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.

Block Reduction

All model coverage reports indicate the status of the Simulink **Block reduction** parameter at the beginning of the report. In the following example, you set **Force block reduction off**.

Simulation Optimization Options

Inline Parameters	off
Block Reduction	forced off
Conditional Branch Optimization	on

In the next example, you enabled the Simulink **Block reduction** parameter, and you did not set **Force block reduction off**.

Simulation Optimization Options

Inline Parameters	off
Block Reduction	on
Conditional Branch Optimization	on

Consider the following model where the simulation does not execute the MinMax1 block because there is only one input—the constant 3.
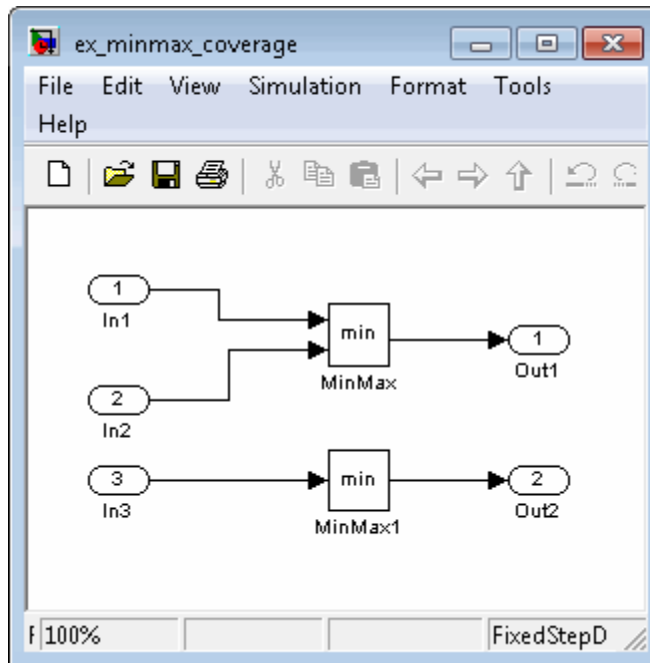
If you set **Force block reduction off**, the report contains no coverage data for this block because the minimum input to the MinMax1 block is always 1.

MinMax block "MinMax1"

Parent:	/ex_minmax_coverage
Uncovered Links:	←

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

Decisions analyzed:

Logic to determine output	0%
input 1 is the minimum	0/0

If you do not set **Force block reduction off**, the report contains no coverage data for reduced blocks.

Reduced Blocks

Blocks eliminated from coverage analysis by block reduction model simulation setting:

. . . ex_minmax_coverage/MinMax1

Saturate on Integer Overflow Analysis

On the **Coverage** tab, if you select the **Saturate on integer overflow** coverage metric, the software creates a Saturation on Overflow analyzed table in the model coverage report for each block with the **Saturate on integer overflow** parameter selected.

The Saturation on Overflow analyzed table lists the number of times a block saturates on integer overflow, indicating a true decision. If the block does not

saturate on integer overflow, the table indicates a false decision. Outcomes that do not occur are in red highlighted table rows.

The following graphic shows the Saturation on Overflow analyzed table for the MinMax block in the `Mixing & Combustion` subsystem of the `fuelsys` model.

MinMax block "MinMax"

Parent:	fuelsys/engine gas dynamics/Mixing & Combustion
Uncovered Links:	← →

Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	50% (1/2) decision outcomes
Saturation on Overflow	50% (1/2) objective outcomes

Decisions analyzed:

Logic to determine output	50%
input 1 is the maximum	204508/204508
input 2 is the maximum	0/204508

Saturation on Overflow analyzed:

Saturate on integer overflow	50%
false	204508/204508
true	0/204508

To display and highlight the block in question, click the block name associated with the Saturation on Overflow analyzed table, as in this example from the `fuelsys` model.
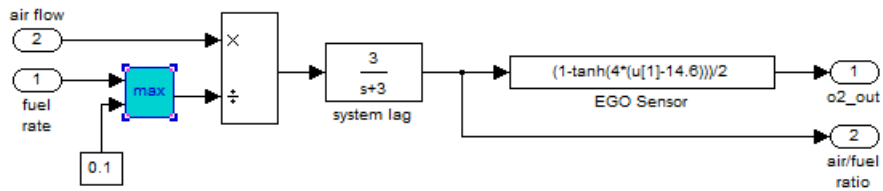
Signal Range Analysis

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

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 appear directly under parent blocks. Each block name in the **Signal Ranges** report is a link.

For example, select the EGO sensor link to display this block highlighted in its native diagram.
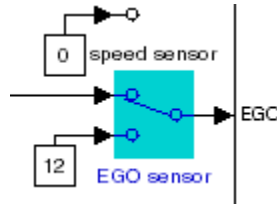
Select the SwitchControl link to display this block in its own subsystem by looking under the mask for EGO sensor.
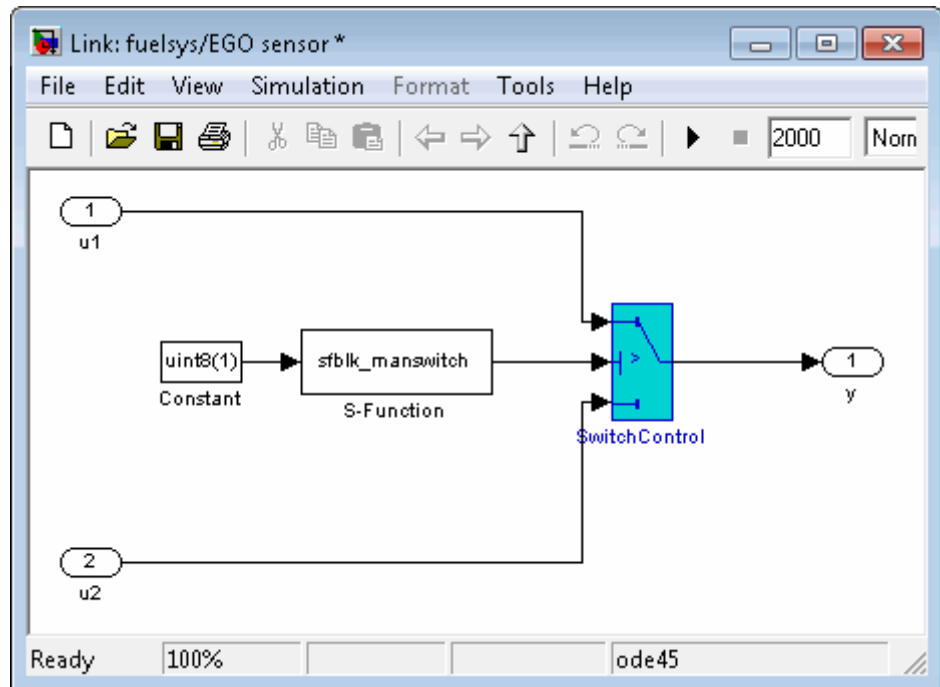
Signal Size Coverage for Variable-Dimension Signals

If you select **Signal Size**, the software creates a Variable Signal Widths section after the Signal Ranges data in the model coverage report. This

section lists the maximum and minimum signal sizes for all output ports
in the model that have variable-size signals. It also lists the memory that
Simulink allocated for that signal, as measured during simulation. This list
does *not* include signals whose size does not vary during simulation.

The following example shows the Variable Signal Widths section in a coverage
report.

Variable Signal Widths:

Hierarchy	Min	Max	Allocated
. . . Abs	2	5	5
. . . Abs1	4	4	5
. . . MinMax1	2	5	5
. . . Switch	2	5	5
. . . Switch1	2	5	5
. . . Selector	4	4	5
. . . c2ri			
. . . . . . out1	4	4	5
. . . . . . out2	4	4	5
. . . Subsystem			
. . . . . . LogicalOperator	1	2	2
. . . . . . Switch1	1	2	2
. . . . . . Switch2	1	2	2

Each block is reported in hierarchical fashion; child blocks appear directly
under parent blocks. Each block name in the Variable Signal Widths list
is a link.

In this example, the Abs block signal size varied from 2 to 5, with an allocation of 5. Click the Abs link in the report. The Model Editor becomes current, with the Abs block highlighted.
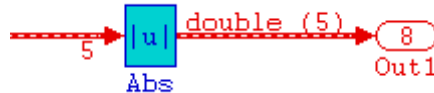
After the analysis, the variable-size signals have a wider line design. `double (5)` in this example indicates the data type and allocation for that signal.

Simulink Design Verifier Coverage

If you select **Simulink Design Verifier**, the analysis collects coverage data for all Simulink Design Verifier blocks in your model.

For an example of how this works, open the `sldvdemo_debounce_testobjblks` model.

This model contains two Test Objective blocks:

• The True block defines a property that the signal have a value of 2.

• The Edge block, inside the Masked Objective subsystem, describes the property where the output of the AND block in the Masked Objective subsystem changes from 2 to 1.

The Simulink Design Verifier software analyzes this model and produces a harness model that contains test cases that achieve certain test objectives. To see if the original model achieves those objectives, simulate the harness model and collect model coverage data. The model coverage tool analyzes any decision points or values within an interval that you specify in the Test Objective block.

In this example, the coverage report shows that you achieved 100% coverage of the True block because the signal value was 2 at least once. The signal value was 2 in 6 out of 14 time steps.

Design Verifier Test Objective block "True"

Parent:	sldvdemo_debounce_testobjblks_harness/Test Unit (copied from sldvdemo_debounce_testobjblks)

Metric	**Coverage**
Test Objective	100% (1/1) objective outcomes

Points/Intervals analyzed:

Point : 2	6/14

The input signal to the Edge block achieved a value of `True` once out of 14 time steps.

Design Verifier Test Objective block "Edge"

Parent:	sldvdemo_debounce_testobjblks_harness/Test Unit (copied from sldvdemo_debounce_testobjblks)/Masked Objective

Metric	**Coverage**
Test Objective	100% (1/1) objective outcomes

Points/Intervals analyzed:

Point : T	1/14

Model Summary Reports

If the top-level model contains Model blocks or calls external 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 files called by MATLAB code in the model. For more information, see "External MATLAB File Coverage Reports" on page 19-40.

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 files called from the model (externalmfile,I externalmfile1, andexternalmfile2).
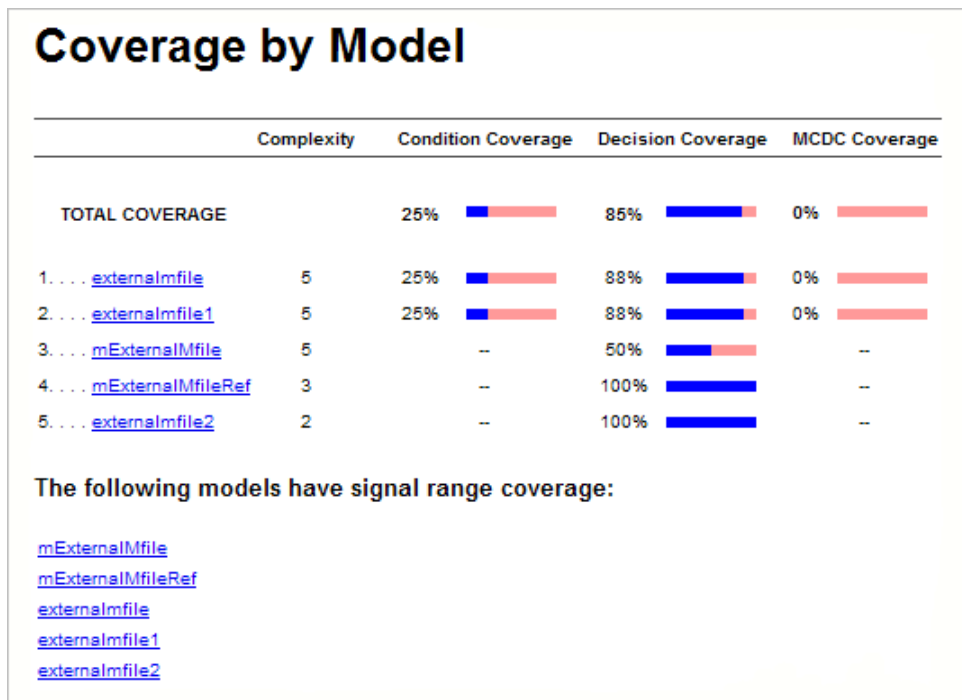
Model Reference Coverage Reports

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 19-3. Coverage results are recorded as if the referenced model was a standalone model; the report gives no indication that the model is referenced in a Model block.

External MATLAB File Coverage Reports

If your top-level model calls any external MATLAB files, select
Coverage for External MATLAB files on the **Coverage** tab of the
Coverage Settings dialog box. The software creates a report, named
*MATLAB_file_name*_cov.html, for each distinct file called from the model.
When there are several calls to a given file from the model, the software
creates only one report for that file, but it accumulates coverage from all the
calls to the file. The external MATLAB file coverage report does not include
information about what parts of the model call the external file.

The first section of the external MATLAB file coverage report contains
summary information about the external file, 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 reports coverage for the external file and the function in that 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 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 files and you select
Coverage for External MATLAB files in the Coverage Settings dialog box,
the results include coverage for all external files called from:

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

If the subsystem parameter **Read/Write Permissions** is set to
NoReadOrWrite, the software does not record coverage for that subsystem.

For example, in the fuelsys model, you click **Select Subsystem**, and select
coverage for the EGO sensor subsystem.
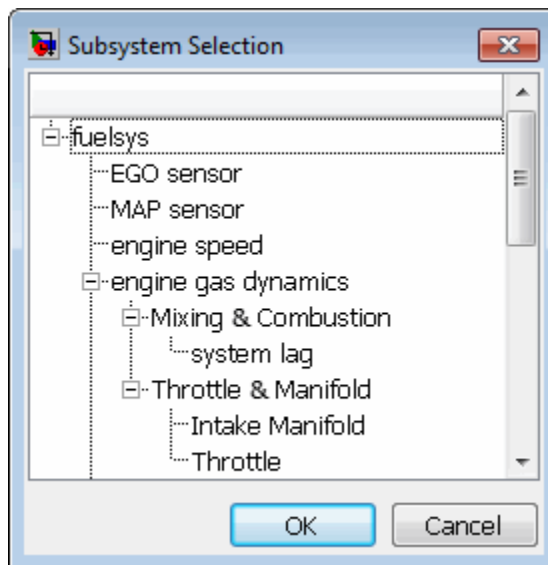
The report is similar to the model coverage report, except that it includes only results for the EGO sensor subsystem and its contents.

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

Excluding Model Objects From Coverage

What Is Coverage Filtering?

Coverage filtering is excluding model objects from model coverage when you simulate your Simulink model. You specify which objects you want to be excluded from coverage recording.

After you specify the objects to exclude, when you simulate your model, Simulink Verification and Validation software does not record coverage for filtered objects in your model.

When to Use Coverage Filtering

You use coverage filtering to facilitate a bottom-up approach to recording model coverage. If you have a large model, there may be design elements that intentionally do not record 100% coverage. You might also have several design elements that do not record 100% coverage that must record 100% coverage. You can temporarily or permanently eliminate these elements from coverage recording to focus on a subset of objects for testing and modification.

This approach allows you to iterate more efficiently—focus on a small problem, fix it, and then move on to resolve the next small problem. Before recording coverage for the entire model, you can resolve missing coverage problems with individual parts of the model.

Coverage Filter Rules and Files

In this section...
"What Is a Coverage Filter Rule?" on page 20-4
"What Is a Coverage Filter File?" on page 20-4

What Is a Coverage Filter Rule?

A *coverage filter rule* is a rule that specifies a model object or set of objects to exclude from coverage recording:

Each coverage filter rule includes the following fields:

- **Name**—Name or path of the object to be filtered from coverage
- **Type**—Whether a specific object is filtered or all objects of a given type are filtered
- **Rationale**—An optional description that explains why this object is filtered from coverage

What Is a Coverage Filter File?

A *coverage filter file* is a collection of coverage filter rules. Each rule specifies one or more objects to exclude from coverage recording.

To apply the coverage filter rules during coverage recording, you must first *attach* a coverage filter file to your model. After you attach the coverage filter file, when you simulate the model, the specified objects are excluded from coverage. You can attach a coverage filter file to several Simulink models. However, a model can have only one attached coverage filter file.

MATLAB saves coverage filter files in the MATLAB Current Folder, unless you specify a different folder. The default name for a coverage filter file is *<model_name>*_covfilter.cvf.

If you use the default file name and the coverage filter file exists on the MATLAB path, each time you open the model, you see the coverage filter rules, unless another coverage filter file is already attached to that model.

Model Objects That You Can Exclude From Coverage

In your model, the objects that you can filter from coverage recording are:

- Simulink blocks that receive coverage, including MATLAB Function blocks
- Subsystems and their contents. When you exclude a subsystem from coverage recording, none of the objects inside the subsystem record coverage.
- Individual library-linked blocks or charts
- All reference blocks linked to a library
- Stateflow charts, subcharts, states, transitions, and temporal events

For a complete list of model objects that receive coverage, see Chapter 16, "Model Objects That Receive Model Coverage".

Managing Coverage Filter Rules for a Simulink Model

In this section...
"Edit the Coverage Filter Rules" on page 20-6
"Save the Coverage Filter to a File" on page 20-9
"Attach a Coverage Filter File to a Model" on page 20-9
"View Coverage Filter Rules in Your Model" on page 20-10
"Remove a Coverage Filter Rule" on page 20-10

Edit the Coverage Filter Rules

- "Create a Coverage Filter Rule" on page 20-6
- "Add a Rationale to a Coverage Filter Rule" on page 20-7
- "Create Additional Coverage Filter Rules" on page 20-8
- "Remove a Coverage Filter Rule" on page 20-9

Create a Coverage Filter Rule

To create a coverage filter rule:

1 In the Coverage Settings dialog box, enable model coverage.

2 In the model window, right-click a model object and select
Coverage > Exclude ....

The following table lists the **Exclude ...** menu options. Depending on which
option you select, the **Type** field is automatically set for the coverage filter
rule you selected; you cannot override the value in the **Type** field.

If you select Coverage > ...	The rule type is ...
Exclude this block	by block path
Exclude all blocks with type <block_type>	by block type

If you select Coverage > ...	The rule type is ...
Exclude all blocks with type MATLAB Function	by block type
Exclude all blocks with type Truth Table	by block type
Exclude subsystem with all dependents	by subsystem
Exclude referenced library: <library_name>	by library reference
Exclude subsystem with all descendants	by subsystem
Exclude chart with all descendants	by chart
Exclude mask type <mask name>	by mask type
Exclude state with all descendants	by state
Exclude this transition	by transition
Exclude temporal event <event_name>	by temporal event

Add a Rationale to a Coverage Filter Rule

Optionally, you can add text that describes why you want to exclude that object or objects from coverage recording, and might be useful to others who review the coverage for your model. When you add a new coverage filter rule, the Coverage Filter Viewer opens. To add the rationale:

1 Double-click the Rationale field for the rule.

2 Delete the existing text.

3 Add the rationale for excluding this object.

The following graphic shows examples of text in the **Rationale** field.
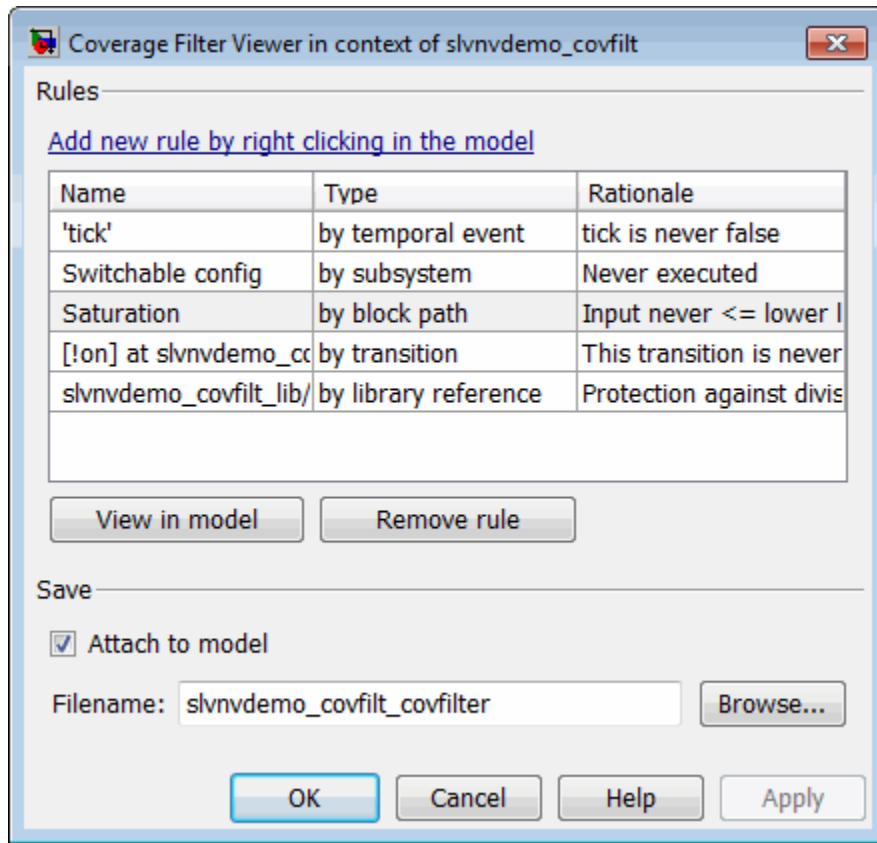
Note The **Rationale** field is the only coverage filter rule field that you can edit in the Coverage Filter Viewer.

Create Additional Coverage Filter Rules

From the Coverage Filter Viewer, you can navigate back to the model to create as many coverage filter rules as you need. To return to the model window, click **Add new rule by right-clicking in the model**.

For each rule that you add, the Coverage Filter Viewer opens so that you can specify a rationale for excluding that object from coverage.

Remove a Coverage Filter Rule

To delete a coverage filter rule:

1 To open the Coverage Filter Viewer, right-click anywhere in the model and select **Coverage > Open Filter Viewer**.

2 Select all the rules that you want to remove.

3 Click **Remove rule**.

Save the Coverage Filter to a File

After you define the coverage filter rules, save the rules to a file so that you can reuse them with this model or with other models. By default, coverage filter files are named *<model_name>*_covfilter.cvf.

In the Coverage Filter Viewer:

1 In the **File name** field, specify a file name for the filter file or accept the default file name.

2 Click **Apply** to save the coverage filter rules to that file.

If you make multiple changes to the coverage filter rules, apply the changes to the coverage filter file each time.

Attach a Coverage Filter File to a Model

Attach a coverage filter file to your model so that each time you open the model, the coverage filter rules apply when you simulate your model.

In the Coverage Filter Viewer:

1 Select **Attach file to model**.

2 Click **Apply**.

Note You can also attach a coverage filter file to your model in the Coverage Settings dialog box, on the **Filter** tab.

You can have only one coverage filter file attached to a model at a time. If you attach a different coverage filter file, the newly attached file replaces the previously attached file.

Two or more models can have the same coverage filter file attached. If a model has an attached filter file that contains coverage filter rules for specific objects in a different model, those rules are ignored during coverage recording.

View Coverage Filter Rules in Your Model

Whenever you define a coverage filter rule or remove an existing coverage filter rule, the Coverage Filter Viewer opens. This dialog box lists all the coverage filter rules for your model. For more information, see "Using the Coverage Filter Viewer" on page 20-11.

To open the Coverage Filter Viewer, right-click anywhere in the model window and select **Coverage > Open Filter Viewer**.

If you are inside a subsystem, you can view any coverage filter rule attached to the subsystem. To open the Coverage Filter Viewer, right-click any object inside the subsystem and select **Coverage > Show filter parent**.

Remove a Coverage Filter Rule

To remove a model object from coverage filtering, right-click the object and select **Coverage > Remove ...**. The Coverage Filter Viewer opens. The coverage filter rule for the select model object is no longer on the list of rules.

To remove additional rules, in the Coverage Filter Viewer, select the rule and click **Remove rule**.

Using the Coverage Filter Viewer

In the Coverage Filter Viewer, you can:

- Review and manage the coverage filter rules for your Simulink model.
- Attach and detach coverage filter files for your model.
- Navigate to the model to create additional coverage filter rules.
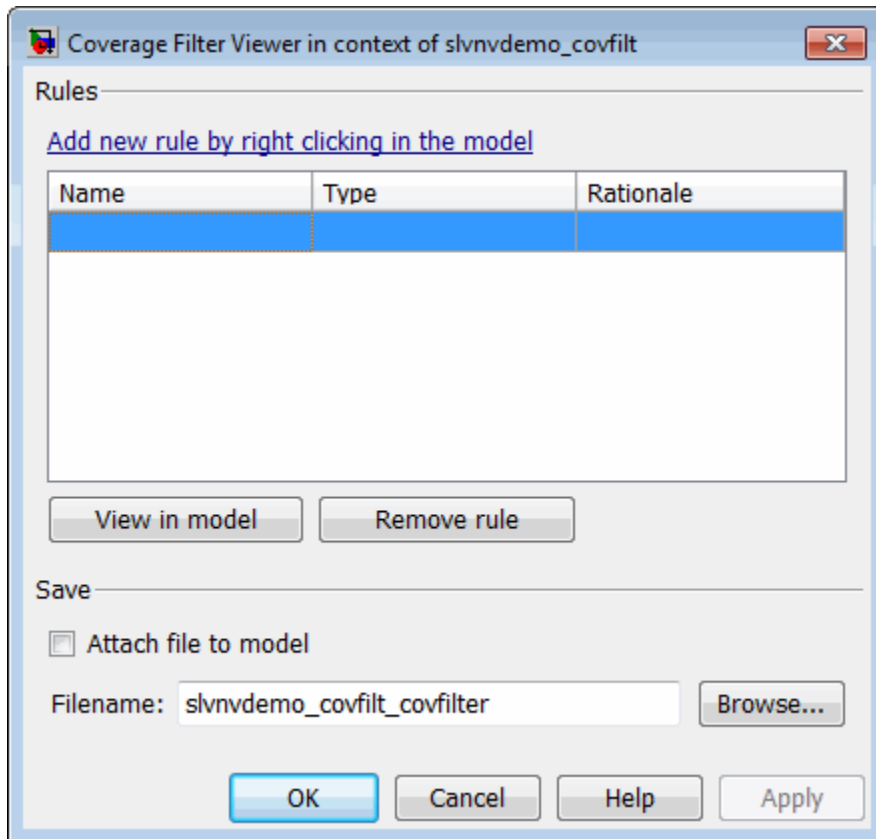
To ...	Do this:
Navigate to the model to create coverage filter rules.	Click **Add new rule by right-clicking in the model**.
Navigate to a model object associated with a rule.	**1** Select the rule. **2** Click **View in model**.
Delete a rule.	**1** Select the rule. **2** Click **Remove rule**.
Save the current rules to a file.	**1** Enter a file name or browse to the file. **2** Click **Apply**.
Attach the current filter file to the model.	**1** Clear the **Attach file to model** check box. **2** Click **Apply**.
Detach the current filter file from the model.	**1** Click **Attach file to model**. **2** Click **Apply**.
Attach a new filter file to the model.	**1** Click **Browse**. **2** Navigate to the desired filter file. **3** Click **Open**. **4** Click **Attach file to model**. **5** Click **Apply**.
Close the Coverage Viewer and save the changes.	Click **OK**.

Creating Coverage Filter Rules for a Simulink Model

About the Example Model

In this example, when you simulate the slvnvdemo_covfilt model, the model does not record 100% coverage. In subsequent steps, you filter certain objects from recording coverage. These steps allow you to focus on specific parts of the model to test for coverage.

The slvnvdemo_covfilt model is configured to record and report coverage during simulation for the following coverage metrics:

- Decision coverage
- Condition coverage
- Modified condition/decision coverage (MCDC)

Simulating the Example Model and Reviewing Coverage

To identify areas of your model that do not record 100% coverage, simulate the model and record coverage.

1 Open the demo model:

   slvnvdemo_covfilt

2 Select **Simulation > Start**.

When the simulation is complete, an HTML coverage report opens. This model does not record 100% coverage.

Summary

Model Hierarchy/Complexity:			Test 1					
			D1		C1		MCDC	
1. slvnvdemo_covfilt	29	36%		67%		50%		
2. . . . Mode Logic	13	55%		75%		50%		
3. . . . . . . SF: Mode Logic	12	55%		75%		50%		
4. . . . . SF: Clipped	6	42%		NA		NA		
5. . . . . SF: Full	2	25%		NA		NA		
6. . . . Normal config (feedthrough)	2	50%		NA		NA		
7. . . . Switchable config	5	17%		NA		NA		
8. . . . protected division	2	0%		NA		NA		
9. . . . protected division1	2	25%		NA		NA		
10. . . . time capture	2	50%		NA		NA		
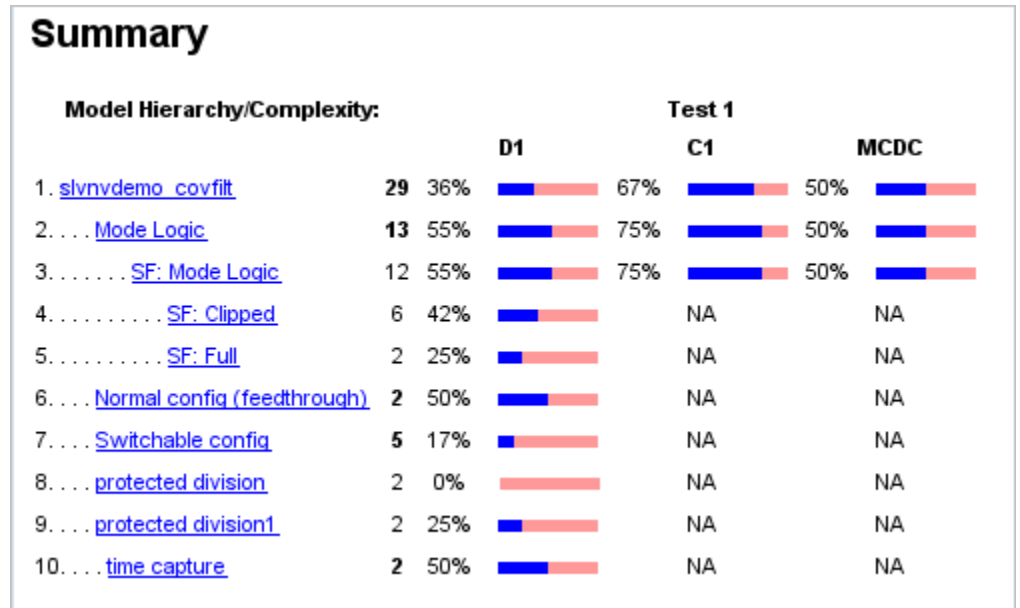
Filtering a Stateflow Transition

In the Mode Logic Stateflow chart, the [!on] transition is never false because it evaluates only when the [on] transition is false. If you do not collect coverage for the [!on] transition, the chart behavior does not change, so you should filter the [!on] transition.

1 Open the Mode Logic chart.
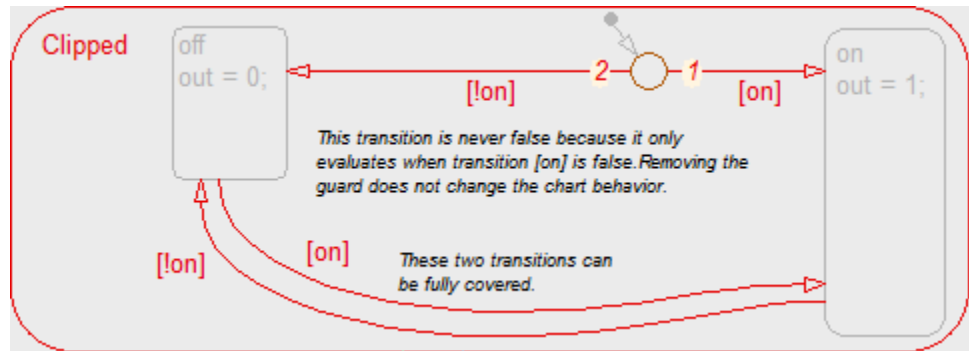
2 Right-click the [!on] transition and select **Coverage > Exclude this transition**.

The Coverage Filter Viewer opens with the new filter rule listed.

3 Click in the **Rationale** field and enter the reason for excluding this transition, for example, This transition is never evaluated.

4 Save this rule to a filter file with the default name, slvnvdemo_covfilt_covfilter.cvf. Click **Apply**.

If you are using this filter for the first time, the file is created in your MATLAB Current Folder. You can specify a different file name and location for your filter file.

5 Attach this filter file to the slvnvdemo_covfilt model. Click **Attach file to model** and click **Apply**.

6 Click **OK** to close the Coverage Filter Viewer.

7 Simulate the model again and review the results in the coverage report.

Under **Filtered Objects**, the report lists the [!on] transition as filtered from the coverage analysis.

If you open the Mode Logic chart and click the transition, the Informer window displays filtering information and the **Rationale** text.
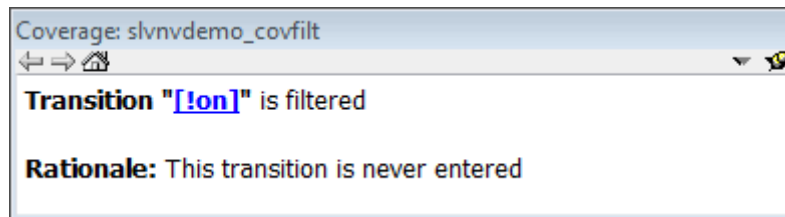
Filtering a Stateflow Temporal Event

Temporal events in Stateflow are a common cause for missing coverage in a chart, because they sometimes form an condition for coverage that can never be satisfied.

For example, in the Mode Logic chart, as you can see from the coverage report, the temporal event tick is never false.

Conditions analyzed:

Description:	True	False
Condition 1, "tick"	40	0
Condition 2, "after(4, tick)"	10	30

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

Decision/Condition:	True Out	False Out
Transition trigger expression		
Condition 1, "tick"	TT	(Fx)
Condition 2, "after(4, tick)"	TT	TF

As a result, you cannot record 100% condition and MCDC coverage for the transition `after(4, tick)`.
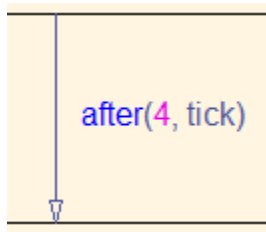
To filter this temporal event, filter the transition `after(4, tick)`.

1 Open the Mode Logic chart.

2 Right-click the `after(4, tick)` transition and select **Coverage > Exclude temporal event 'tick'**.

The Coverage Filter Viewer opens with the new filter rule listed.

3 Click in the **Rationale** field and enter text for excluding this transition, for example, `tick is never false`.

4 Save this rule to the current filter. Click **Apply**.

5 Simulate the model again and review the results.

The **Filtered Blocks** section of the report lists the transition `after(4, tick)`. Condition and MCDC coverage for the Mode Logic chart is not recorded.

Filtering Library Reference Blocks

The `slvnvdemo_covfilt` model contains two instances of a library-linked subsystem in the library `slvnvdemo_covfilt_lib`:

- protected division
- protected division1

The library subsystem is a protection against division by zero and might not be relevant in the coverage report. Exclude it from coverage for this model.

1 In the model window, right-click either of the protected division reference blocks.

When you filter a library block, all instances of that block are filtered from coverage.

2 Select **Coverage > Exclude reference library: slvnvdemo_covfilt_lib/protected division**.

The Coverage Filter Viewer opens with the new filter rule listed.

3 Click in the **Rationale** field for this new rule and enter text for excluding this transition, for example, `Protection against division by zero`.

4 Save this rule to the current filter. Click **Apply**.

5 Simulate the model again and review the results.

The **Filtered Blocks** section of the report lists both protected division reference blocks. No coverage is recorded for the two protected division subsystems.

In the model window, the blocks filtered from coverage are colored grey.
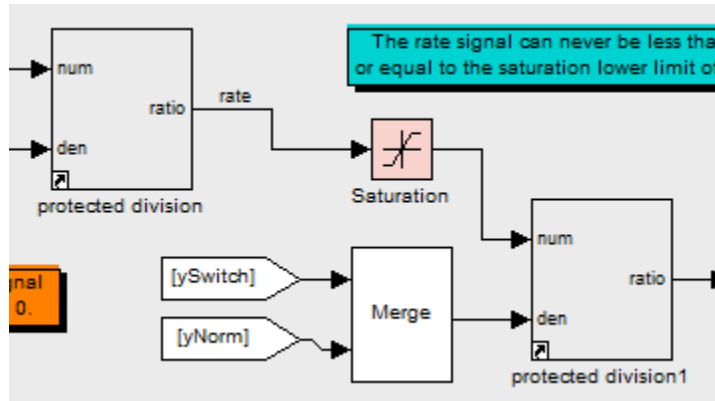
Filtering a Subsystem

The slvnvdemo_covfilt model uses a Constant block to drive the enable port for the Switchable config subsystem. Because the constant is always 0, this subsystem never executes.

Exclude the Switchable config subsystem from coverage.

1 In the model window, right-click the Switchable config subsystem.

2 Select **Coverage > Exclude subsystem with all descendants**.

3 Click in the **Rationale** field for this new rule and enter text for excluding this transition, for example, Never executed.

4 Save this rule to the current filter. Click **Apply**.

5 Simulate the model again and review the results.

The **Filtered Blocks** section of the report lists the Switchable config subsystem. No coverage is recorded for the subsystem.

Filtering a Specific Block

In the slvnvdemo_covfilt model, the rate signal can never be less than or equal to 0, which is the value of the **Lower limit** parameter of the Saturation block. This condition leads to missing coverage.

Exclude the Saturation block from coverage.

1 In the model window, right-click the Saturation block.

2 Select **Coverage > Exclude this block**.

3 Click in the **Rationale** field for this new rule and enter text for excluding this transition, for example, Input never <= lower limit (0).

4 Save this rule to the current filter. Click **Apply**.

5 Simulate the model again and review the results.

The **Filtered Blocks** section of the report lists the Saturation block. Coverage for that block is omitted from the report.

21

Automating Model Coverage Tasks

Commands for Automating Model Coverage Tasks

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, and store and report the results. For a list of the model coverage commands that the Simulink Verification and Validation software provides, see Function Reference.

The following sections 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.

To create a test object with a specified label (used for reporting results):

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

To create a test with a setup command:

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

You execute the setup command in the base MATLAB workspace, just prior to running the instrumented simulation. Use this command 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 for analysis
label	String for reporting results
setupCmd	Command executed prior to simulation

Field	Description
settings.condition	Set to 1 for condition coverage
settings.decision	Set to 1 for decision coverage
settings. designverifier	Set to 1 for coverage for Simulink Design Verifier blocks.
settings.mcdc	Set to 1 for MCDC coverage
settings.overflowsaturation	Set to 1 for saturate on integer overflow coverage
settings.sigrange	Set to 1 for signal range coverage
settings.sigsize	Set to 1 for signal size coverage.
settings.tableExec	Set to 1 for lookup table coverage
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 for only referenced models not listed in the excludedModels subfield
modelRefSettings. excludeTopModel	Set to 1 for excluding coverage for the top model
modelRefSettings. excludedModels	String specifying a comma-separated list of referenced models for which coverage is disabled when modelRefSettings.enable specifies filtered

Field	Description
`emlSettings.` `enableExternal`	Set to 1 to enable coverage for external program files called by MATLAB functions in your model
`options.` `forceBlockReduction`	Set to 1 to override the Simulink **Block reduction** parameter if it is enabled.

Running Tests with cvsim

Use the `cvsim` command to simulate a test specification object.

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

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

This command executes the `cvtest` object `cvto` by simulating the corresponding model. `cvsim` returns the coverage results in the `cvdata` object `cvdo`. 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 setting model parameters for the Simulink `sim` command to apply during simulation:

• The following command executes the test object `cvto` and simulates the model using the default model parameters. The `cvsim` function returns the coverage results in the `cvdata` object `cvdo` and returns the simulation outputs in a `Simulink.SimulationOutput` object `simOut`:

```
[cvdo,simOut] = cvsim(cvto)
```

• The following commands create a structure, `paramStruct`, that specifies the model parameters to use during the simulation. The first command specifies that the simulation collect decision, condition, and MCDC coverage for this model.

```
paramStruct.CovMetricSettings  = 'dcm';
paramStruct.SimulationMode     = 'rapid';
paramStruct.AbsTol             = '1e-5';
paramStruct.SaveState          = 'on';
paramStruct.StateSaveName      = 'xoutNew';
paramStruct.SaveOutput         = 'on';
paramStruct.OutputSaveName     = 'youtNew';
```

Note For a complete list of model parameters, see "Model Parameters" in the Simulink documentation.

The following `cvsim` command executes the test object `cvto` and simulates the model using the model parameter values specified in `paramStruct`:

```
[cvdo,simOut] = cvsim(cvto,paramStruct);
```

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

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

Retrieving Coverage Details from Results

Simulink Verification and Validation provides commands that allow you to retrieve specific coverage information from the `cvtest` object after you have simulated your model and recorded coverage. Use these commands to retrieve the specified coverage information for a block, subsystem, or Stateflow chart in your model or for the model itself:

- `complexityinfo` — Cyclomatic complexity coverage
- `conditioninfo` — Condition coverage
- `decisioninfo` — Decision coverage
- `mcdcinfo` — Modified condition/decision (MCDC) coverage
- `sigrangeinfo` — Signal range coverage
- `sigsizeinfo` — Signal size coverage
- `tableinfo` — Lookup Table block coverage
- `getCoverageinfo` — Coverage for Simulink Design Verifier blocks

The basic format of these functions is:

```
coverage = <coverage_type_prefix>info = (cvdata_object,
object, ignore_descendants)
```

- `coverage` — Multipart vector containing the retrieved coverage results for `object`
- `cvdata_object` — `cvdata` object that you create when you call `cvtest`
- `object` — Handle to a model or object in the model
- `ignore_descendants` — Flag to ignore coverage results in subsystems, referenced models, and Stateflow charts

Creating Cumulative Coverage for Reusable Subsystems and Stateflow Constructs

Simulink Verification and Validation provides cumulative coverage for multiple instances of identically configured:

- Reusable subsystems
- Stateflow constructs

To obtain cumulative coverage, you add the individual coverage results at the command line. You can create cumulative coverage for multiple instances across models and test harnesses by adding the individual coverage results.

This example shows how to create and view cumulative coverage results for a model with a reusable subsystem. At the MATLAB command line:

1 Type `slvnvdemo_cv_mutual_exclusion`. This model has two instances of a reusable subsystem. The instances are named `Subsystem 1` and `Subsystem 2`.

2 Execute the commands for `Subsystem 1` decision coverage:

```
testobj1 = cvtest([model '/Subsystem 1']);
testobj1.settings.decision = 1;
covobj1 = cvsim(testobj1);
```

3 Execute the commands for `Subsystem 2` decision coverage:

```
testobj2 = cvtest([model '/Subsystem 2']);
testobj2.settings.decision = 1;
covobj2 = cvsim(testobj2);
```

4 Execute the command to create cumulative decision coverage for `Subsystem 1` and `Subsystem 2`:

```
covobj3 = covobj1 + covobj2;
```

5 Create an HTML report for Subsystem 1 decision coverage:

```
cvhtml('subsystem1',covobj1)
```

The report indicates that decision coverage is 50% for `Subsystem 1`. The `true` condition for `enable logical value` is not analyzed.

Subsystem "Subsystem 1"

Uncovered Links:

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

Decisions analyzed:

enable logical value	50%
false	51/51
true	0/51

6 Create an HTML report for `Subsystem 2` decision coverage:

```
cvhtml('subsystem2',covobj2)
```

The report indicates that decision coverage is 50% for `Subsystem 2`. The `false` condition for `enable logical value` is not analyzed.

Subsystem "Subsystem 2"

Uncovered Links:

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

Decisions analyzed:

enable logical value	50%
false	0/51
true	51/51

7 Create an HTML report for cumulative decision coverage for `Subsystem 1` and `Subsystem 2`:

```
cvhtml('cum_subsystem',covobj3)
```

Cumulative decision coverage for reusable subsystems `Subsystem 1` and `Subsystem 2` is 100%. Both the `true` and `false` conditions for `enable logical value` are analyzed.

Subsystem "Subsystem 1"

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	2	2
Decision (D1)	100% (2/2) decision outcomes	100% (2/2) decision outcomes

Decisions analyzed:

enable logical value	100%
false	51/102
true	51/102

Creating HTML Reports with cvhtml

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

The following command creates an HTML report of the coverage results in the `cvdata` object `cvdo`. The results are written to the file `file` in the current MATLAB folder.

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

The following command 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:

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

Higher 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 `cvsave` function:

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

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

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

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

You can 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 loaded are returned in `cvtos`, a cell array of `cvtest` objects. The `cvdata` objects that are 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

When using the `cvload` command, be aware of the following considerations:

- When a model with the same name exists in the coverage database, only the compatible results are loaded from the file. They reference the existing model to prevent duplication.

- When 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.

Using Coverage Commands in a Script

The following script demonstrates some common model coverage commands.

This script:

- Creates two data files to load before simulation.

- Creates two `cvtest` objects, `testObj1` and `testObj2`, and simulates them using the default model parameters. Each `cvtest` object uses the `setupCmd` property to load a data file before simulation.

- Enables decision, condition, and MCDC coverage.

- Retrieves the decision coverage results for the Adjustable Rate Limited subsystem.

- Uses `cvhtml` to display the coverage results for the two tests and the cumulative coverage.

- Compute cumulative coverage with the + operator and save the results

```
mdl = 'slvnvdemo_ratelim_harness';
mdl_subsys = 'slvnvdemo_ratelim_harness/Adjustable Rate Limiter';

open_system(mdl);
open_system(mdl_subsys);

t_gain = (0:0.02:2.0)'; u_gain = sin(2*pi*t_gain);
t_pos = [0;2]; u_pos = [1;1]; t_neg = [0;2]; u_neg = [-1;-1];
save('within_lim.mat','t_gain','u_gain','t_pos','u_pos', ...
    't_neg', 'u_neg');
t_gain = [0;2]; u_gain = [0;4]; t_pos = [0;1;1;2];
u_pos = [1;1;5;5]*0.02; t_neg = [0;2]; u_neg = [0;0];
save('rising_gain.mat','t_gain','u_gain','t_pos','u_pos', ...
    't_neg', 'u_neg');

testObj1             = cvtest(mdl_subsys);
testObj1.label       = 'Gain within slew limits';
testObj1.setupCmd    = 'load(''within_lim.mat'');';
testObj1.settings.mcdc = 1;
testObj1.settings.condition = 1;
```

```
testObj1.settings.decision = 1;

testObj2         = cvtest(mdl_subsys);
testObj2.label   = 'Rising gain that temporarily exceeds slew limit';
testObj2.setupCmd = 'load(''rising_gain.mat'');';
testObj2.settings.mcdc = 1;
testObj2.settings.condition = 1;
testObj2.settings.decision = 1;

[dataObj1,simOut1] = cvsim(testObj1);
decision_cov1 = decisioninfo(dataObj1,mdl_subsys);
percent_cov1 = 100 * decision_cov1(1) / decision_cov1(2)
cc_cov2 = complexityinfo(dataObj1, mdl_subsys);

[dataObj2,simOut2] = cvsim(testObj2,[0 2]);
decision_cov2 = decisioninfo(dataObj2,mdl_subsys);
percent_cov2 = 100 * decision_cov2(1) / decision_cov2(2)
cc_cov2 = complexityinfo(dataObj1, mdl_subsys);


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

cvsave('ratelim_testdata',cumulative);

close_system('slvnvdemo_ratelim_harness',0);
```

Using Model Coverage Commands for Referenced Models

Introduction

Using Simulink software, you can include one model in another with Model blocks. Each Model block represents a reference to another model, called a *referenced model* or *submodel*. A referenced model 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*.

Note For more information, see "Referencing a Model" in the *Simulink User's Guide*.

Model coverage supports referenced models that operate in Normal mode. 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. 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 21-20
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 21-20
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 21-21

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

TopModel references **SubModel1**

SubModel1 references **SubModel2**
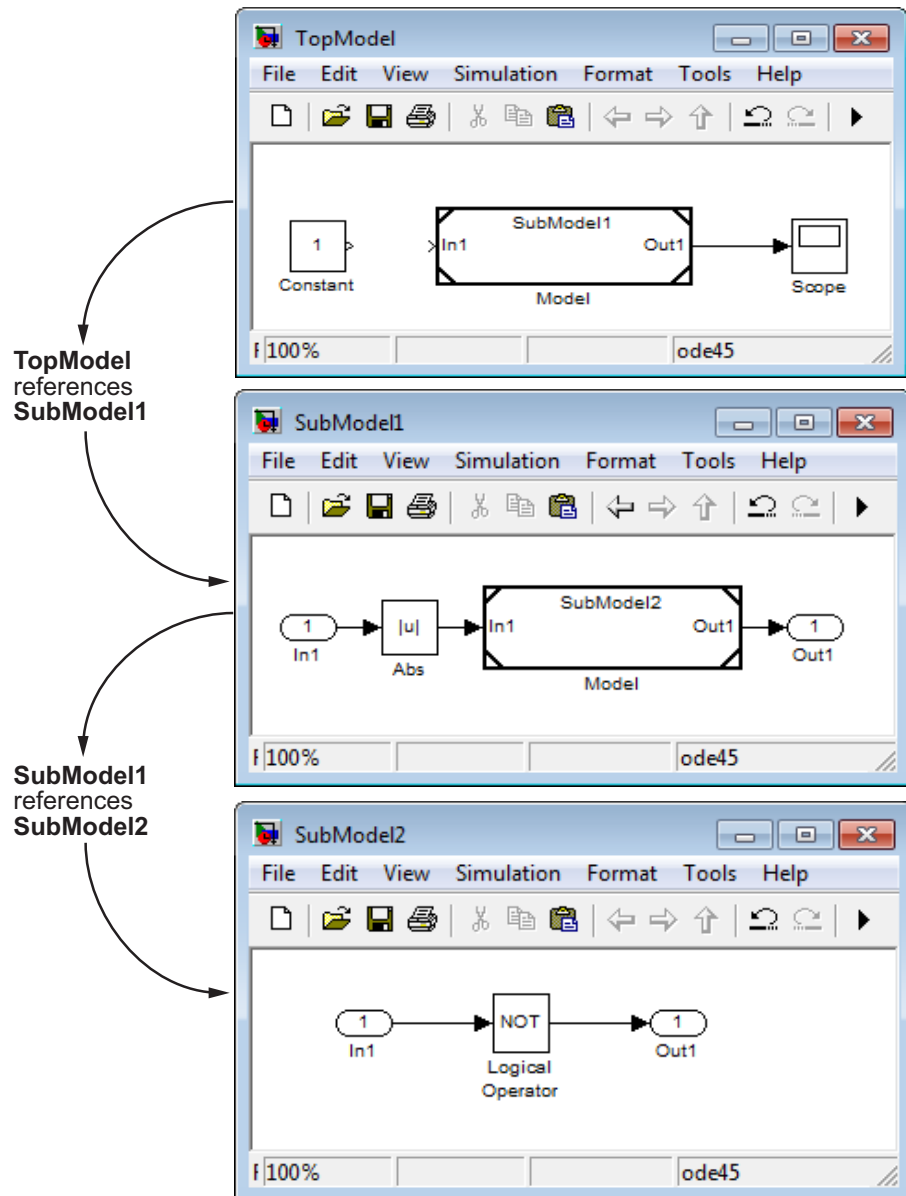
Creating a Test Group with cv.cvtestgroup

The cvtest command creates a test specification object for a Simulink model. 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, so that you can customize the contents of the cv.cvtestgroup object 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`.

You can use parameters from the Simulink `sim` function in a `cvsimref` command to control the simulation:

- To return the simulation time vector `t`, matrix of state values `x`, and matrix of output values `y`:

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

- To override 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 `sim` function reference page.

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`, so that you can 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 use 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 "Creating HTML Reports with cvhtml" on page 21-12).

Checking Systems with the Model Advisor

22

Checking Systems Interactively

About Checking Systems Interactively

Use the Model Advisor to check a Simulink model or subsystem for adherence to modeling guidelines. Using MathWorks checks, you can easily apply guidelines across projects and development teams. For more information, see "Consulting the Model Advisor" in the Simulink documentation.

The Model Advisor includes MathWorks checks that help you define and implement consistent design guidelines. When you run the MathWorks checks, the Model Advisor reviews your model for conditions and configuration settings that cause inaccurate or inefficient simulation and code generation of the system that the model represents. Depending on which products you have installed, the Model Advisor displays different MathWorks checks . For more information, see:

- "Simulink Checks"

- "Embedded Coder Checks"

- "Simulink Verification and Validation Checks"

- "Simulink Control Design Checks"

Software is inherently complex and may not be completely free of errors. Model Advisor checks might contain bugs. MathWorks reports known bugs brought to its attention on its Bug Report system at http://www.mathworks.com/support/bugreports/. The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

While applying Model Advisor checks to your model will increase the likelihood that your model does not violate certain modeling standards or guidelines, it is ultimately your responsibility to verify, using multiple methods, that the system being developed provides its intended functionality and does not include any unintended functionality.

Limiting the Scope of a Model Advisor Analysis

In this section...
"What Is a Model Advisor Exclusion?" on page 22-3
"Model Advisor Exclusion Files" on page 22-3
"Creating Model Advisor Exclusions" on page 22-4
"Reviewing Model Advisor Exclusions" on page 22-6
"Managing Exclusions" on page 22-8

What Is a Model Advisor Exclusion?

To save time during model development and verification, you might decide to limit the scope of a Model Advisor analysis of your model. You can do this by excluding individual blocks from checks. You create a *Model Advisor exclusion* to exclude blocks in the model from selected checks. You can exclude all or selected checks from:

- Simulink blocks
- Stateflow charts

After you specify the blocks to exclude, when you use the Model Advisor to analyze your model, the software uses the exclusion information to exclude blocks from specified checks during the analysis. A Model Advisor exclusion file stores the block and check exclusion information.

You can also exclude blocks from checks that you write by using Model Advisor APIs. For more information, see "Excluding Blocks From Custom Checks" on page 25-55.

Model Advisor Exclusion Files

A *Model Advisor exclusion file* specifies a collection of blocks to exclude from specified checks during a Model Advisor analysis.

To exclude blocks from specified checks during an analysis of your model, you first create exclusions and save them in an exclusion file. You can also use an existing Model Advisor exclusion file.

When you analyze a model with Model Advisor exclusions, the blocks in the exclusion file are excluded from the specified checks during the analysis. You can use an exclusion file with several models. However, a model can have only one exclusion file.

Unless you specify a different folder, the Model Advisor saves exclusion files in the current folder. The default name for an exclusion file is `<model_name>_exclusions.xml`.

If you create an exclusion file and save your model, you attach the exclusion file to your model. Each time that you open the model, the blocks and checks specified in the exclusion file are excluded from the analysis.

In the Model Advisor Exclusion Editor dialog box, you can view each exclusion. Each exclusion includes the information listed in the following table.

Field	Description
Rationale	A description of why this object is excluded from Model Advisor checks.
Type	Whether a specific block is excluded or all blocks of a given type are excluded.
Value	Name of block or blocks that are excluded.
Check ID (s)	Names of checks for which the block exclusion applies.

Creating Model Advisor Exclusions

You might want to exclude model blocks and checks from a Model Advisor analysis to save time during model development and verification. To create a Model Advisor exclusion:

1 In the model window, right-click a block and select **Model Advisor**. The following table lists the possible menu options. The following table lists what you can exclude and the corresponding menu options.

To ...	Select Model Advisor > ...
Exclude the block from all checks.	**Exclude block only > All Checks**
Exclude all blocks of this type from all checks.	**Exclude all blocks with type <block_type> > All Checks**
Exclude the block from selected checks.	• **Exclude block only > Select Checks**. • In the Check Selector dialog box, select the checks. Click **OK**.
Exclude all blocks of this type from selected checks.	• **Exclude all blocks with type <block_type> > Select Checks**. • In the Check Selector dialog box, select the checks. Click **OK**.
Exclude the block from all failed checks. This option is available after a Model Advisor analysis.	**Exclude block only > Only failed checks**
Exclude all blocks of this type from all failed checks. This option is available after a Model Advisor analysis.	**Exclude all blocks with type <block_type> > Only failed checks**
Exclude the block from a failed check. This option is available after a Model Advisor analysis.	**Exclude block only > <name of failed check>**
Exclude all blocks of this type from a failed check. This option is available after a Model Advisor analysis.	**Exclude all blocks with type <block_type> > <name of failed check>**

2 In the Model Advisor Exclusion Editor dialog box, click **OK** or **Apply** to create the exclusion and save the information to an exclusion file. If this exclusion is the first one, a Save Exclusion File as dialog box opens. In this dialog box, click **Save** to create a exclusion file with the default name `<model_name>_exclusions.xml` in the current folder. Optionally, in the Save Exclusion File dialog box, you can select a different file name or location.

3 If you want to change the exclusion file name or location:

a In the Model Advisor Exclusion Editor dialog box, select **Change**.

b In the Change Exclusion File dialog box, select **Save as**.

c In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.

d In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information to an exclusion file.

You can create as many Model Advisor exclusions as you want by right-clicking model blocks and selecting **Model Advisor**. Each time that you create an exclusion, the Model Advisor Exclusion Editor dialog box opens. In this dialog box, in the **Rationale** field, you can specify a reason for excluding blocks or checks from the Model Advisor analysis. The rationale might be useful to others who review your model.

If you create an exclusion file and save your model, you attach the exclusion file to your model. Each time that you open the model, the blocks and checks specified in the exclusion file are excluded from the analysis.

Reviewing Model Advisor Exclusions

You can review the exclusions associated with your model. After opening your model, you might want to view the exclusions defined for your model by an attached exclusion file.

To view exclusions information, either before or after an Model Advisor analysis:

- Right-click in the model window or right-click a block and select **Model Advisor > Open Model Advisor Exclusion Editor**. The Model Advisor Exclusion Editor dialog box lists all the exclusions for your model.

- In the model window, select **Tools > Model Advisor** to launch the Model Advisor.

  1 On the Model Advisor window toolbar, select **Highlighting > Highlight Exclusions**. By default, this menu option is selected.

  2 In the Model Advisor window, click the **Enable Model Advisor highlighting** toggle button ( ).

  3 In the left pane of the Model Advisor window, select a check. The blocks excluded from the check appear in the model window, highlighted in gray with a black border.

After the Model Advisor completes an analysis, you can view exclusion information for individual checks in the:

- HTML report. In the Model Advisor window, make sure select the **Show report after run** check box before the analysis.

- Model Advisor window. In the left pane of the Model Advisor window, select **By Product > Simulink > < name of check >**

If the check ...	The HTML report and Model Advisor window ...
Has no exclusions rules applied.	State that no exclusions were applied to this check. This might happen if you exclude a block from a check that passes without an exclusion.
Does not support exclusions.	State that the check does not support exclusions. This might happen if you exclude a block from a check that does not support exclusions.
Is excluded from a block.	Lists the check exclusion rules.

Managing Exclusions

- "Saving Exclusions To a File" on page 22-8
- "Loading an Exclusion File" on page 22-9
- "Detaching an Exclusion File" on page 22-9
- "Removing an Exclusion" on page 22-10
- "Adding a Rational to an Exclusion" on page 22-10

To open the Model Advisor Exclusion Editor dialog box, either right-click in the model window or right-click a block and select **Model Advisor > Open Model Advisor Exclusion Editor**.
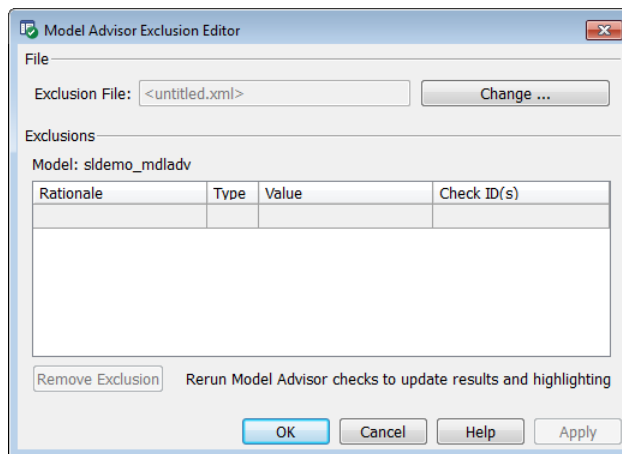
Note The **Rationale** field is the only field that you can edit in the Model Advisor Exclusion Editor.

Saving Exclusions To a File

To save an exclusion to a file:

1 In the Model Advisor Exclusion Editor dialog box, click **OK** or **Apply**. If this exclusion is the first one, a Save Exclusion File as dialog box opens.

In this dialog box, click **Save** to create an exclusion file with the default name <model_name>_exclusions.xml in the current folder. Optionally, in the Save Exclusion File dialog box, you can select a different file name or location.

2 If you want to change the exclusion file name or location:

   a In the Model Advisor Exclusion Editor dialog box, select **Change**.

   b In the Change Exclusion File dialog box, select **Save as**.

   c In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.

   d In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information to an exclusion file.

Loading an Exclusion File

To load an existing exclusion file for use with your model:

1 In the Model Advisor Exclusion Editor dialog box, click **Change**.

2 In the Change Exclusion File Dialog box, click **Load**.

3 Navigate to the exclusion file that you want to use with your model. Select **Open**.

4 In the Model Advisor Exclusion Editor dialog box, click **OK** to associate the exclusion file with your model.

Detaching an Exclusion File

To detach an exclusion file associated with your model:

1 In the Model Advisor Exclusion Editor dialog box, click **Change**.

2 In the Change Exclusion File Dialog box, click **Detach**.

3 In the Model Advisor Exclusion Editor dialog box, click **OK** to detach the exclusion file from your model.

Removing an Exclusion

To remove an exclusion:

1 In the Model Advisor Exclusion Editor dialog box, double-click the **Rationale** field for the exclusion.

2 Delete the existing text.

3 Add the rationale for excluding this object.

Adding a Rational to an Exclusion

You can add text that describes why you excluded a particular block or blocks from selected checks during Model Advisor analysis. A description might be useful to others who review your model.

1 In the Model Advisor Exclusion Editor dialog box, select the exclusions that you want to remove.

2 Click **Remove Exclusion**.

Limiting the Scope of a Model Advisor Analysis By Excluding Gain and Outport Blocks

This example shows how to exclude a gain block and all outport blocks from a Model Advisor check during a Model Advisor analysis. By excluding individual blocks from checks, you limit the scope of the analysis and might save time during model development and verification.

1 At the MATLAB command line, type sldemo_mdladv.

2 From the model window, select **Tools > Model Advisor** to open the Model Advisor.

3 A System Selector — Model Advisor dialog box opens. Click **OK**.

4 In the left pane of the Model Advisor window, expand **By Product > Simulink**. Select the **Show report after run** check box to see an HTML report of check results after you run the checks.

5 Run the selected checks by clicking the **Run Selected Checks** button. After the Model Advisor runs the checks, an HTML report displays the check results in a browser window. The check **Identify unconnected lines, input ports, and output ports** triggers a warning.

6 In the left pane of the Model Advisor window, select the check **By Product > Simulink > Identify unconnected lines, input ports, and output ports**.

7 In the Model Advisor window, click the **Enable Model Advisor highlighting** button ().

   • The model window opens. The blocks causing the **Identify unconnected lines, input ports, and output ports** check warning are highlighted in yellow.
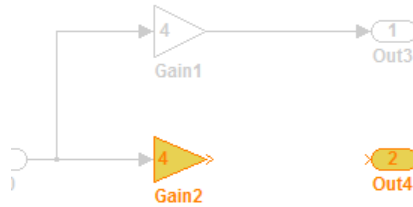
- The Model Advisor Highlighting window opens with a link to the Model Advisor window. In the Model Advisor window, you can find more information about the check results and how to fix the warning condition.
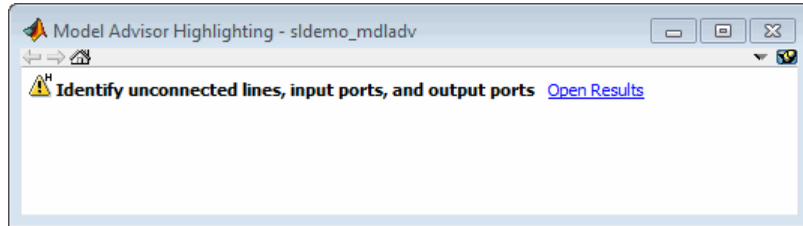
8 After reviewing the check results, exclude the Gain2 block from all Model Advisor checks:

   a In the model window, right-click the Gain2 block and select **Model Advisor > Exclude block only > All Checks.**
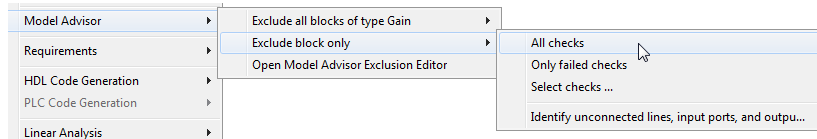
   b In the Model Advisor Exclusion Editor dialog box, double-click in the first row of the **Rationale** field, and enter `Exclude gain block`.
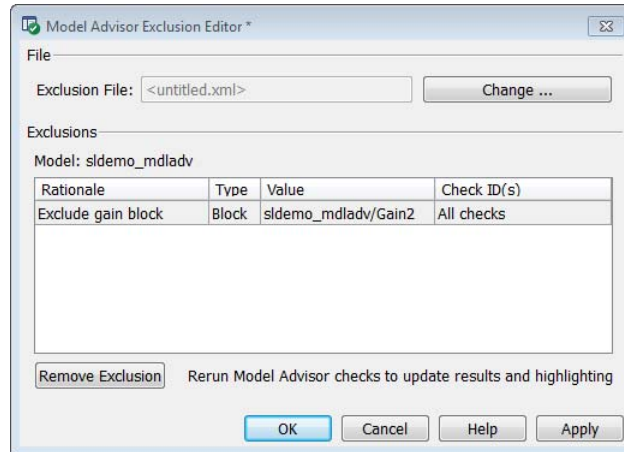
c Click **OK** to create the exclusion file.

d In the Save Exclusion File as dialog box, click **Save** to create a exclusion file with the default name sldemo_mdladv_exclusions.

9 After reviewing the check results, exclude all Outport blocks from the Identify unconnected lines, input ports, and output ports check:

a In the model window, right-click the Out4 block and select **Model Advisor > Exclude all blocks of type Outport > Select checks**.
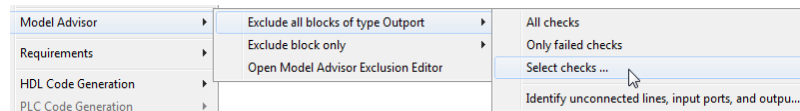
b In the Check Selector dialog box, scroll down and select Identify unconnected lines, input ports, and output ports. Click **OK**.
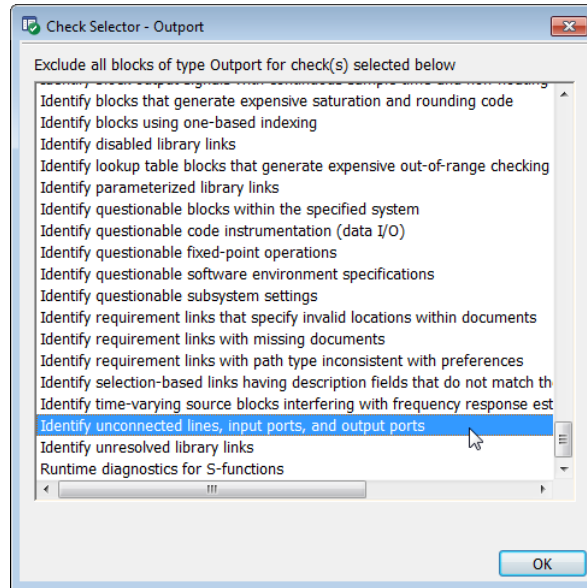
Alternately, to exclude all Outport blocks from the Identify unconnected lines, input ports, and output ports check, right-click the Out4 block and select **Model Advisor > Exclude all blocks of type Outport > Identify unconnected lines, input ports, and output ports**.

c In the Model Advisor Exclusion Editor dialog box, click **OK** to also exclude all Outport blocks from the Identify unconnected lines, input ports, and output ports check. The sldemo_mdladv_exclusions file is updated with the Outport block exclusions.

10 In the left pane of the Model Advisor window, select **By Product > Simulink** and then:

- Select the **Show report after run** check box.

- Select **Run Selected Checks** to run a Model Advisor analysis.

11 After the Model Advisor completes the analysis, you can view exclusion information for the Identify unconnected lines, input ports, and output ports check in the:

- HTML report:

✅ **Identify unconnected lines, input ports, and output ports**

Identify unconnected lines, input ports, and output ports in the model

Passed
There are no unconnected lines, input ports, and output ports in this model.

Check Exclusions Rules

Rationale	Exclusion Usage Count
Exclusion for all blocks of type Outport	1
Exclude gain block	1

- Model Advisor window. In the left pane of the Model Advisor window, select **By Product > Simulink > Identify unconnected lines, input ports, and output ports**.

Identify unconnected lines, input ports, and output ports in the model

Passed
There are no unconnected lines, input ports, and output ports in this model.

Check Exclusions Rules

Rationale	Exclusion Usage Count
Exclusion for all blocks of type Outport	1
Exclude gain block	1

- Model window. In the left pane of the Model Advisor window, select **By Product > Simulink > Identify unconnected lines, input ports, and output ports.** Then click the **Enable Model Advisor highlighting** button (⧉).
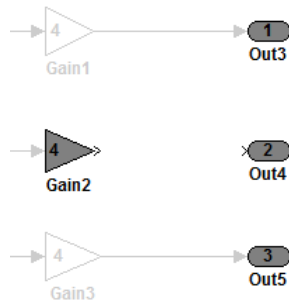
12 Close sldemo_mdladv.

Related Examples

- "Running Model Advisor Checks"

More About

- "Consulting the Model Advisor"
- "Model Advisor Highlighting"

23

Checking Systems Programmatically

Overview

The Simulink Verification and Validation product includes a programmable interface for scripting and for command-line interaction with the Model Advisor. Using this interface, you can:

- Create scripts and functions for distribution that check one or more systems using the Model Advisor.

- Run the Model Advisor on multiple systems in parallel on multicore machines (requires a Parallel Computing Toolbox™ license).

- Check one or more systems using the Model Advisor from the command line.

- Archive results for reviewing at a later time.

Workflow for Checking Systems Programmatically

To define the workflow for running multiple checks on systems:

1 Specify a list of checks to run. Do one of the following:

   • Create a Model Advisor configuration file that includes only the checks that you want to run. For more information, see "Organizing Checks and Folders Using the Model Advisor Configuration Editor" on page 26-3.

   • Create a list of check IDs. For more information on finding check IDs, see "Finding Check IDs" on page 23-4.

2 Specify a list of systems to check.

3 Run the Model Advisor checks on the list of systems using the `ModelAdvisor.run` function.

4 Archive and review the results of the run. For details, see "Archiving and Viewing Results" on page 23-10.

Finding Check IDs

An *ID* is a unique string that identifies a Model Advisor check. You find check IDs in the Model Advisor, using check context menus.

To Find...	Do This...
A check ID	1 In the left pane of the Model Advisor, select the check. 2 Right-click the check name and select **Send Check ID to Workspace**. The ID is displayed in the Command Window and sent to the base workspace.
Check IDs for selected checks in a folder	1 In the left pane of the Model Advisor, select the checks for which you want IDs. Clear the other checks in the folder. 2 Right-click the folder and select **Send Check ID to Workspace**. An array of the selected check IDs are sent to the base workspace.

If you know a check ID from a previous release, you can find the current check ID using the ModelAdvisor.lookupCheckID function. For example, the check ID for **By Product > Simulink Verification and Validation > Modeling Standards > DO-178B Checks > Check safety-related optimization settings** prior to Release 2010b was DO178B:OptionSet. Using the ModelAdvisor.lookupCheckID function returns:

```
>> NewID = ModelAdvisor.lookupCheckID('DO178B:OptionSet')

NewID =

mathworks.do178.OptionSet
```

Creating a Function for Checking Multiple Systems

The following tutorial guides you through creating and testing a function to run multiple checks on any model. The function returns the number of failures and warnings.

1 In the MATLAB window, select **File > New > Function**.

2 Save the function as run_configuration.m.

3 In the MATLAB Editor, specify [output_args] as [fail, warn].

4 Rename the function run_configuration.

5 Specify input_args to SysList.

6 Inside the function, specify the list of checks to run using the demo Model Advisor configuration file:

```
fileName = 'slvnvdemo_mdladv_config.mat';
```

7 Call the ModelAdvisor.run function:

```
SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName);
```

8 Determine the number of checks that return warnings and failures:

```
fail=0;
warn=0;

for i=1:length(SysResultObjArray)
    fail = fail + SysResultObjArray{i}.numFail;
    warn = warn + SysResultObjArray{i}.numWarn;
end
```

The function should now look like this:

```
function [fail, warn] = run_configuration( SysList)
%RUN_CONFIGURATION Check systems with Model Advsior
%   Check systems given as input and return number of warnings and
%   failures.
```

```
fileName = 'slvnvdemo_mdladv_config.mat';
fail=0;
warn=0;

SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName);

for i=1:length(SysResultObjArray)
    fail = fail + SysResultObjArray{i}.numFail;
    warn = warn + SysResultObjArray{i}.numWarn;
end
end
```

9 Save the function.

10 Test the function. In the MATLAB Command Window, run
run_configuration.m on the sldemo_auto_climatecontrol/Heater
Control subsystem:

```
[failures, warnings] = run_configuration(...
    'sldemo_auto_climatecontrol/Heater Control');
```

11 Review the results. Click the Summary Report link to open the Model
Advisor Command-Line Summary report.

Checking Multiple Systems in Parallel

Checking multiple systems in parallel reduces the processing time required by the Model Advisor to check multiple systems. If you have a Parallel Computing Toolbox license, you can check multiple systems in parallel on a multicore host machine.

Creating a Function for Checking Multiple Systems in Parallel

If you have a Parallel Computing Toolbox license and a multicore host machine, you can create the following function to check multiple systems in parallel:

1 Create the `run_configuration` function as described in "Creating a Function for Checking Multiple Systems" on page 23-5.

2 Save the function as `run_fast_configuration.m`.

3 In the Editor, change the name of the function to `run_fast_configuration`.

4 Add another input to the `run_fast_configuration` function so that the inputs are now:

```
SysList, numParallel
```

5 In the `run_fast_configuration` function, before calling the `ModelAdvisor.run` function, add a call to the `matlabpool` function that evaluates the number of cores to use:

```
eval(['matlabpool open ' num2str(numParallel)]);
```

6 At the end of the `run_fast_configuration` function, add a call to close the `matlabpool` function:

```
matlabpool close;
```

The function should now look like this:

```
function [fail, warn] = run_fast_configuration(SysList, numParallel)
%RUN_FAST_CONFIGURATION Check systems in parallel with Model Advisor
%   Check systems given as input in parallel on the number of cores
%   specified as input. Return number of warnings and failures.
fileName = 'slvnvdemo_mdladv_config.mat';
fail=0;
warn=0;

eval(['matlabpool open ' num2str(numParallel)]);
```

```
SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName);

for i=1:length(SysResultObjArray)
    fail = fail + SysResultObjArray{i}.numFail;
    warn = warn + SysResultObjArray{i}.numWarn;
end

matlabpool close
end
```

7 Save the function.

8 Test the function. In the MATLAB Command Window, create a list of systems:

```
SysList={'sldemo_auto_climatecontrol/Heater Control',...
    'sldemo_auto_climatecontrol/AC Control', 'rtwdemo_iec61508'};
```

9 Run `run_fast_configuration` on the list of systems, specifying `numParallel` to be the number of cores in your system. For example, the following command specifies two cores:

```
% Run on 2 cores
[failures, warnings] = run_fast_configuration(SysList, 2);
```

10 Review the results. Click the Summary Report link to open the Model Advisor Command-Line Summary report.

Archiving and Viewing Results

You can archive and view the results of running the Model Advisor programmatically as described in the following sections:

In this section...
"Archiving Results" on page 23-10
"Viewing Results in the Command Window" on page 23-10
"Viewing Results in the Model Advisor Command-Line Summary Report" on page 23-12
"Viewing Results in the Model Advisor GUI" on page 23-13
"Viewing the Model Advisor Report" on page 23-13

Archiving Results

After you run the Model Advisor programmatically, you can archive the results for use at another time. The ModelAdvisor.run function returns a cell array of ModelAdvisor.SystemResult objects, one for each system run. If you save the objects, you can use them to view the results at a later time without rerunning the Model Advisor. For details, see "Saving and Loading Objects".

For an example of archiving results, see "Archiving and Viewing Results Example" on page 23-14.

Viewing Results in the Command Window

When you run the Model Advisor programmatically, the system-level results of the run are displayed in the Command Window. For example, when you run the function that you created in "Creating a Function for Checking Multiple Systems" on page 23-5, the following results are displayed:

```
Systems passed: 0 of 1
Systems with warnings: 1 of 1
Systems failed: 0 of 1
Summary Report
```

The Summary Report link provides access to the Model Advisor Command-Line Summary report (see "Viewing Results in the Model Advisor Command-Line Summary Report" on page 23-12).

You can review additional results in the Command Window by calling the `DisplayResults` parameter when you run the Model Advisor. For example, run the Model Advisor as follows:

```
SysResultObjArray = ModelAdvisor.run('sldemo_auto_climatecontrol/Heater Control',...
    'Configuration','slvnvdemo_mdladv_config.mat','DisplayResults','Details');
```

The results displayed in the Command Window are:

```
Running Model Advisor
Running Model Advisor on sldemo_auto_climatecontrol/Heater Control
============================================================
Model Advisor run: 09-Feb-2011 15:29:50
Configuration: slvnvdemo_mdladv_config.mat
System: sldemo_auto_climatecontrol/Heater Control
System version: 7.7
Created by: The MathWorks Inc.
============================================================
(1) Warning: Check model diagnostic parameters [check ID: mathworks.maab.jc_0021]
------------------------------------------------------------
(2) Warning: Check for fully defined interface [check ID: mathworks.iec61508.RootLevelInports]
------------------------------------------------------------
(3) Pass: Check for unconnected objects [check ID: mathworks.iec61508.UnconnectedObjects]
 ------------------------------------------------------------
(4) Pass: Check for questionable constructs [check ID: mathworks.iec61508.QuestionableBlks]
------------------------------------------------------------
 Summary:    Pass     Warning     Fail     Not Run
             2         2           0         0
============================================================


Systems passed: 0 of 1
Systems with warnings: 1 of 1
Systems failed: 0 of 1
Summary Report
```

To display the results in the Command Window after loading an object, use the `viewReport` function.

Viewing Results in the Model Advisor Command-Line Summary Report

When you run the Model Advisor programmatically, a Summary Report link is displayed in the Command Window. Clicking this link opens the Model Advisor Command-Line Summary report. The following graphic is the report that the Model Advisor generates for `run_configuration`.

To view the Model Advisor Command-Line Summary report after loading an object, use the `ModelAdvisor.summaryReport` function.

Viewing Results in the Model Advisor GUI

In the Model Advisor window, you can view the results of running the Model Advisor programmatically using the `viewReport` function. In the Model Advisor window, you can review results, run checks, fix warnings and failures, and view and save Model Advisor reports. For more information, see "Consulting the Model Advisor".

Tip To fix warnings and failures, you must rerun the check in the Model Advisor window.

Viewing the Model Advisor Report

For a single system or check, you can view the same Model Advisor report that you access from the Model Advisor GUI.

To view the Model Advisor report for a system:

- Open the Model Advisor Command-Line Summary report. In the Systems Run table, click the link for the Model Advisor report.

- Use the `viewReport` function.

To view individual check results:

- In the Command Window, generate a detailed report using the `viewReport` function with the `DisplayResults` parameter set to `Details`, and then click the Pass, Warning, or Fail link for the check. The Model Advisor report for the check opens.

- Use the `view` function.

Archiving and Viewing Results Example

The following tutorial guides you through archiving the results of running checks so that you can review them at a later time. To simulate archiving and reviewing, the steps in the tutorial detail how to save the results, clear out the MATLAB workspace (simulates shutting down MATLAB), and then load and review the results.

1 Call the `ModelAdvisor.run` function:

```
SysResultObjArray = ModelAdvisor.run({'sldemo_auto_climatecontrol/Heater Control'},...
    'Configuration','slvnvdemo_mdladv_config.mat');
```

2 Save the `SystResulObj` for use at a later time:

```
save my_model_advisor_run SysResultObjArray
```

3 Clear the workspace to simulate viewing the results at a different time:

```
clear
```

4 Load the results of the Model Advisor run:

```
load my_model_advisor_run SysResultObjArray
```

5 View the results in the Model Advisor:

```
viewReport(SysResultObjArray{1},'MA')
```

Customizing the Model Advisor

Overview of Customizing the Model Advisor

- "About Customizing the Model Advisor" on page 24-2
- "Workflows for Creating Custom Model Advisor Configurations" on page 24-4
- "Requirements for Customizing the Model Advisor" on page 24-6

About Customizing the Model Advisor

Using Model Advisor APIs and the Model Advisor Configuration Editor, you can:

- Define your own custom checks and write your own callback functions.

- Create custom configurations by organizing checks and folders.

- Specify the order in which you make changes to your model by creating a procedural-based configuration.

- Create multiple custom configurations for different projects or modeling guidelines, and switch between these configurations in the Model Advisor.

- Deploy the custom configurations to your users.

For more information about the Model Advisor, see "Consulting the Model Advisor".

If you want to...	Follow workflow...	For more information, see...
Author custom checks.	"Authoring Checks Workflow" on page 25-2	Chapter 25, "Authoring Custom Checks"
Create custom configurations to organize the checks MathWorks provides.	"Creating Configurations by Organizing Checks and Folders Workflow" on page 24-4	Chapter 26, "Creating Custom Configurations by Organizing Checks and Folders"
Use custom checks to create a custom configuration.	"Creating Configurations by Organizing Checks and Folders Workflow" on page 24-4	Chapter 26, "Creating Custom Configurations by Organizing Checks and Folders"
Specify the order in which you make changes to your model.	"Creating Procedural-Based Configurations Workflow" on page 24-5	Chapter 27, "Creating Procedural-Based Model Advisor Configurations"

If you want to...	Follow workflow...	For more information, see...
Deploy the custom configurations to your users.	"Deploying Custom Configurations Workflow" on page 28-2	Chapter 28, "Deploying Custom Configurations"
Verify that models comply with modeling guidelines.		"Consulting the Model Advisor"

Workflows for Creating Custom Model Advisor Configurations

In this section...
"Creating Configurations by Organizing Checks and Folders Workflow" on page 24-4
"Creating Procedural-Based Configurations Workflow" on page 24-5

Creating Configurations by Organizing Checks and Folders Workflow

To customize the Model Advisor with MathWorks and custom checks, perform the following tasks:

1 Review the information in "Requirements for Customizing the Model Advisor" on page 24-6.

2 Optionally, author custom checks in a customization file. See Chapter 25, "Authoring Custom Checks".

3 Organize the checks into new and existing folders to create custom configurations. See Chapter 26, "Creating Custom Configurations by Organizing Checks and Folders".

   a Identify which checks you want to include in your custom Model Advisor configuration. You can use MathWorks checks and/or custom checks.

   b Create the custom configurations using either of the following:

     • Model Advisor Configuration Editor - "Organizing Checks and Folders Using the Model Advisor Configuration Editor" on page 26-3.

     • A customization file - "Organizing Checks and Folders Within a Customization File" on page 26-12.
    For information on whether to use the Model Advisor configuration editor or a customization file, see "Using the Model Advisor Configuration Editor Versus Customization File" on page 26-2.

   c Verify the custom configuration. See "Verifying and Using Custom Configurations" on page 26-22.

4 Optionally, deploy the custom configurations to your users. See Chapter 28, "Deploying Custom Configurations".

5 Verify that models comply with modeling guidelines. See "Consulting the Model Advisor".

Creating Procedural-Based Configurations Workflow

To create a procedural-based configuration, perform the following tasks:

1 Review the information in "Requirements for Customizing the Model Advisor" on page 24-6.

2 Decide on order of changes to your model.

3 Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.

4 Optionally, author custom checks in a customization file. See Chapter 25, "Authoring Custom Checks".

5 Organize the checks into procedures for a procedural-based configuration. See Chapter 27, "Creating Procedural-Based Model Advisor Configurations".

   a Create procedures using the procedure API. For detailed information, see "Creating Procedures Using the Procedures API" on page 27-3.

   b Create the custom configuration by using a customization file. See "Organizing Checks and Folders Within a Customization File" on page 26-12.

   c Verify the custom configuration as described in "Verifying and Using Custom Configurations" on page 26-22.

6 Optionally, deploy the custom configurations to your users. For detailed information, see Chapter 28, "Deploying Custom Configurations".

7 Verify that models comply with modeling guidelines. For detailed information, see "Consulting the Model Advisor".

Requirements for Customizing the Model Advisor

Before customizing the Model Advisor:

- If you want to create custom checks, know how to create a MATLAB script. For more information, see "Scripts" in the MATLAB documentation.

- 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 25-35.

25

Authoring Custom Checks

Authoring Checks Workflow

1 On your MATLAB path, create a *customization file* named `sl_customization.m`. In this file, create a `sl_customization()` function to register the custom checks that you create and optional process callbacks with the Model Advisor. For detailed information, see "Registering Checks and Process Callbacks" on page 25-18.

2 Define custom checks and where they appear in the Model Advisor. For detailed information, see "Defining Custom Checks" on page 25-23.

3 Specify what actions you want the Model Advisor to take for the custom checks by creating a check callback function for each custom check. For detailed information, see "Creating Callback Functions and Results" on page 25-34.

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 25-49.

5 Optionally, specify startup and post-execution actions by creating a process callback function. For detailed information, see "Defining Startup and Post-Execution Actions Using Process Callback Functions" on page 25-20.

Customization File Overview

A *customization file* is a MATLAB file that you create and name
`sl_customization.m`. 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 "Registering Checks and Process Callbacks" on page 25-18).	Required for all customizations to the Model Advisor.
One or more check definitions	Defines all custom checks (see "Defining Custom Checks" on page 25-23).	Required for custom checks and to add custom checks to the **By Product** folder.
Check callback functions	Defines the actions of the custom checks (see "Creating Callback Functions and Results" on page 25-34).	Required for custom checks. You must write one callback function for each custom check.
One or more calls to check input parameters	Specifies input parameters to custom checks (see "Defining Check Input Parameters" on page 25-28).	Optional.
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 25-30).	Optional.

Function	Description	When Required
One or more calls to check actions	Specifies actions the software performs for custom checks (see "Defining Check Actions" on page 25-31 and "Action Callback Function" on page 25-49).	Optional.
One process callback function	Specifies actions to be performed at startup and post-execution time (see "Defining Startup and Post-Execution Actions Using Process Callback Functions" on page 25-20).	Optional.

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

Quick Start Examples

Adding a Customized Check to the By Product Folder

The following example shows how to add a customized check to a Model Advisor **By Product > Demo** subfolder. In this example, the customized check does not check model elements.

1 In your working directory, create the sl_customization.m file, as shown below. This file registers and creates the check registration function defineModelAdvisorChecks, which in turn registers the check callback function SimpleCallback. The function defineModelAdvisorChecks uses a ModelAdvisor.Root object to define the check interface.

```
function sl_customization(cm)

% --- register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Example of a customized check';
rec.TitleTips = 'Added customized check to Product Folder';
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne');
mdladvRoot.publish(rec, 'Demo');

% --- creates SimpleCallback function
function result = SimpleCallback(system);
result={};
```

2 Close the Model Advisor and your model if either are open.

3 In the MATLAB Command Window, enter:

```
sl_refresh_customizations
```

4 From the MATLAB window, select **File > New > Model** to open a new Simulink model window.

5 From the model window, select **Tools > Model Advisor** to open the Model Advisor.

6 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.

7 In the left pane, expand the **By Product** folder to display the subfolders.

The customized check **Example of a customized check** appears in the **By Product > Demo** subfolder.

See Also

- "Registering Checks and Process Callbacks" on page 25-18

Creating a Customized Pass/Fail Check

The following example shows how to create a Model Advisor pass/fail check. In this example, the Model Advisor checks Constant blocks. If a Constant blocks value is numeric, the check fails.

1 In your working directory, update the sl_customization.m file, as shown below. This file registers and creates the check registration function defineModelAdvisorChecks, which also registers the check callback function SimpleCallback. The function SimpleCallback creates a check that finds Constant blocks that have numeric values. SimpleCallback uses the Model Advisor format template.

```
function sl_customization(cm)

% --- register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne')

mdladvRoot.publish(rec, 'Demo');

% --- SimpleCallback function that checks constant blocks
function result = SimpleCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
result   = {};

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');

ft = ModelAdvisor.FormatTemplate('ListTemplate');
ft.setInformation(['This check looks for constant blocks that'...
    'use numeric values']);
```

```
if ~isempty(blk_with_value)
    ft.setSubResultStatusText(['Check has failed. The following '...
        'Constant blocks have numeric values:']);
    ft.setListObj(blk_with_value);
    ft.setSubResultStatus('warn');
    ft.setRecAction('Parameterize the constant block');
    mdladvObj.setCheckResultStatus(false);
else
    ft.setSubResultStatusText(['Check has passed. No constant blocks'...
        ' with numeric values were found.']);
    ft.setSubResultStatus('pass');
    mdladvObj.setCheckResultStatus(true);
end
ft.setSubBar(0);
result{end+1} = ft;
```

2 Close the Model Advisor and your model if either are open.

3 It the MATLAB Command Window, enter:

  `sl_refresh_customizations`

4 From the MATLAB window, select **File > New > Model** to open a new Simulink model window.

5 In the Simulink model window, create two Constant blocks named Const_One and Const_1:

  • Right-click the Const_One block, choose **Constant Parameters**, and assign a **Constant value** of one.

  • Right-click the Const_1 block, choose **Constant Parameters**, and assign a **Constant value** of 1.

  • Save your model as `example2_qs.mdl`.
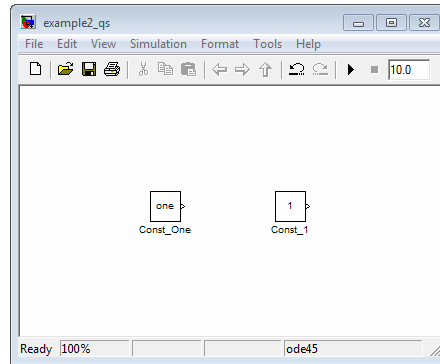
6 From the model window, select **Tools > Model Advisor** to open the Model Advisor.

7 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.

8 In the left pane, click **By Product > Demo > Check Constant block usage**.

9 Click **Run This Check**. The Model Advisor check fails for the Const_1 block and displays a **Recommended Action** to parametrize the constant block.
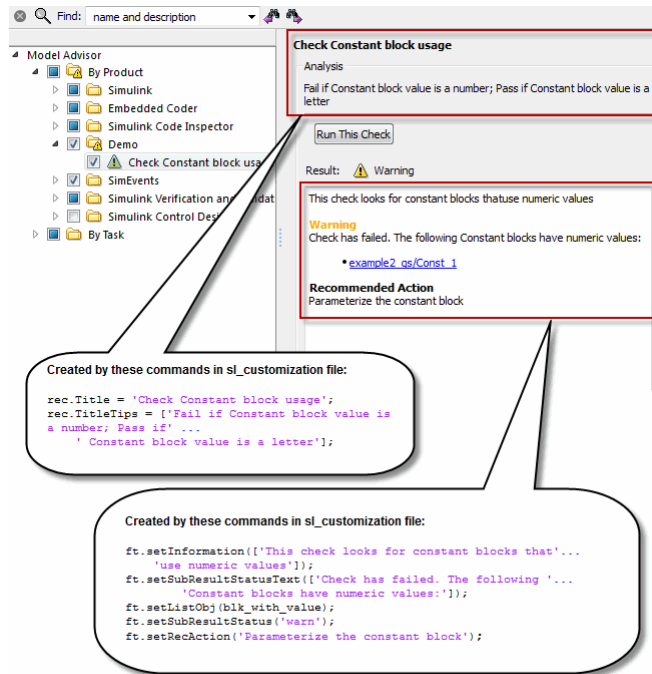
10 Follow the **Recommended Action** to fix the failed Constant block. In the Model Advisor dialog box:

- Double-click the `example2_qs/Const_1` hyperlink.

- Change **Constant Parameters > Constant value** to two, or a nonnumeric value.

- Rerun the Model Advisor check. Both Constant blocks now pass the check.

See Also

- "Registering Checks and Process Callbacks" on page 25-18

- ModelAdvisor.FormatTemplate

Creating a Customized Pass/Fail Check with Fix Action

The following example shows how to create a Model Advisor pass/fail check with a fix action. In this example, the Model Advisor checks Constant blocks. If a Constant block value is numeric, the check fails. The Model Advisor is also customized to create a fix action for the failed checks.

1 In your working directory, update the `sl_customization.m` file, as shown below. This file contains three functions, each of which use the Model Advisor format template:

- `defineModelAdvisorChecks` — Defines the check, creates input parameters, and defines the fix action.

- `simpleCallback` — Creates the check that finds Constant blocks with numeric values.

- `simpleActionCallback` — Creates the fix for Constant blocks that fail the check.

```
function sl_customization(cm)

% --- register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if '...
    'Constant block value is a letter'];
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne')

% --- input parameters
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Text entry example';
inputParam1.Value='VarNm';
inputParam1.Type='String';
inputParam1.Description='sample tooltip';
```

```matlab
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec.setInputParameters({inputParam1});

% -- set fix operation
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@simpleActionCallback);
myAction.Name='Fix Constant blocks';
myAction.Description=['Click the button to update all blocks with'...
    'Text entry example'];
rec.setAction(myAction);

mdladvRoot.publish(rec, 'Demo');

% --- SimpleCallback function that checks constant blocks
function result = SimpleCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
result    = {};

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');

ft = ModelAdvisor.FormatTemplate('ListTemplate');
ft.setInformation(['This check looks for constant blocks that'...
    ' use numeric values']);
if ~isempty(blk_with_value)
    ft.setSubResultStatusText(['Check has failed. The following '...
        'Constant blocks have numeric values:']);
    ft.setListObj(blk_with_value);
    ft.setSubResultStatus('warn');
    ft.setRecAction('Parameterize the constant block');
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
else
    ft.setSubResultStatusText(['Check has passed. No constant blocks'...
        'with numeric values were found.']);
    ft.setSubResultStatus('pass');
    mdladvObj.setCheckResultStatus(true);
end
```

25-13

```
ft.setSubBar(0);
result{end+1} = ft;

% --- creates SimpleActionCallback function that fixes failed check
function result = simpleActionCallback(taskobj)
mdladvObj = taskobj.MAObj;
result    = {};

system = getfullname(mdladvObj.System);

% Get the string from the input parameter box.
inputParams = mdladvObj.getInputParameters;
textEntryEx = inputParams{1}.Value;

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');
ft = ModelAdvisor.FormatTemplate('TableTemplate');
% Define table col titles
ft.setColTitles({'Block','Old Value','New Value'})
for inx=1:size(blk_with_value)
   oldVal = get_param(blk_with_value{inx},'Value');
   ft.addRow({blk_with_value{inx},oldVal,textEntryEx});
   set_param(blk_with_value{inx},'Value',textEntryEx);
end

ft.setSubBar(0);
result = ft;
mdladvObj.setActionEnable(false);
```

2 Close the Model Advisor and your model if either are open.

3 At the MATLAB command line, enter:

   `sl_refresh_customizations`

4 From the MATLAB Command Window, select **File > New > Model** to open a new model.

5 In the Simulink model window, create two Constant blocks named Const_One and Const_1:

- Right-click the Const_One block, choose **Constant Parameters**, and assign a **Constant value** of one.

- Right-click the Const_1 block, choose **Constant Parameters**, and assign a **Constant value** of 1.

- Save your model as example3_qs.mdl.

6 From the model window, select **Tools > Model Advisor** to open the Model Advisor.

7 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.

8 In the left pane, click **By Product > Demo > Check Constant block usage**.

9 Click **Run This Check**. The Model Advisor check fails for the Const_1 block. The Model Advisor box has a **Fix Constant blocks** button in the **Action** section of the Model Advisor dialog box.
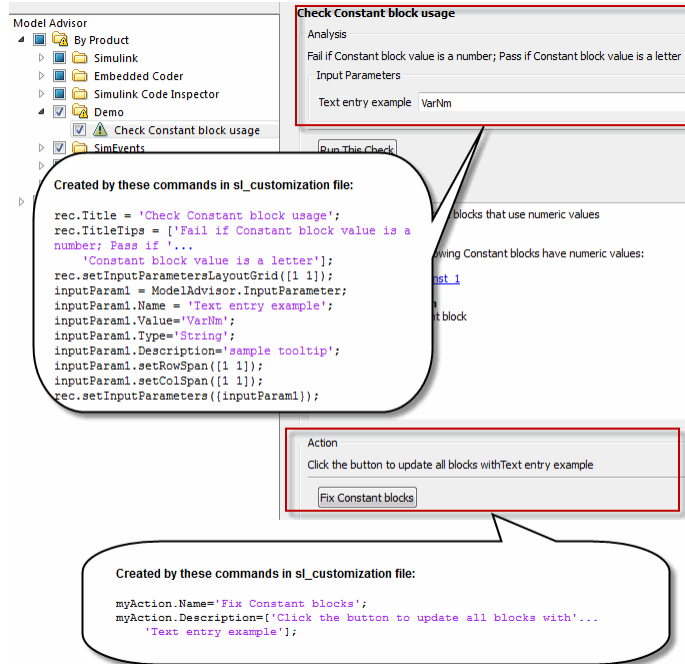
10 In the Model Advisor Dialog box, enter a nonnumeric value in the **Text entry example** parameter field in the **Analysis** section of the Model Advisor dialog box. In this example, the value is VarNm.

11 Click **Fix Constant blocks**. The Const_1 **Constant block value** changes from 1 to the nonnumeric value that you entered in step 10. The **Result** section of the dialog box lists the Old Value and New Value of the Const_1 block.
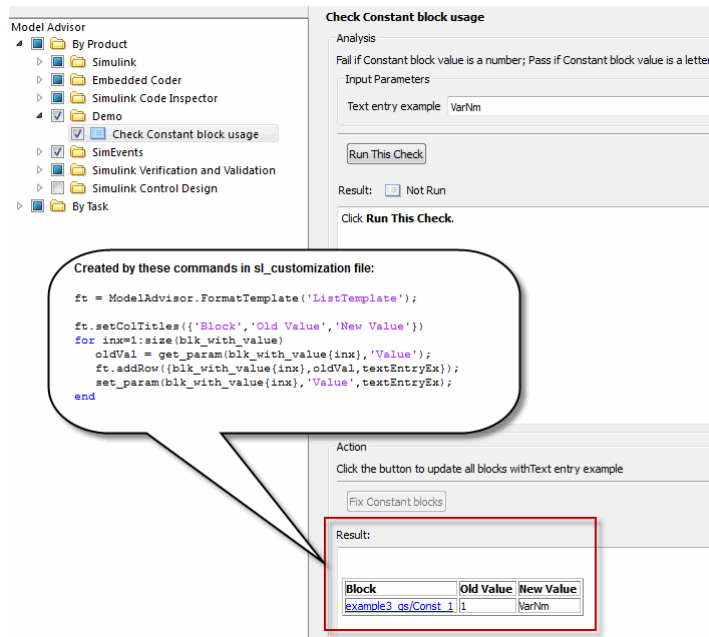
12 In the Model Advisor dialog box, click **Run This Check**. Both constant blocks now pass the check.

See Also

- "Registering Checks and Process Callbacks" on page 25-18

- ModelAdvisor.FormatTemplate

- "Defining Check Input Parameters" on page 25-28 to add input parameters to Model Advisor checks

- ModelAdvisor.Action to add fix actions to Model Advisor checks

Registering Checks and Process Callbacks

In this section...
"Create sl_customization Function" on page 25-18
"Registering Checks and Process Callbacks" on page 25-18
"Defining Startup and Post-Execution Actions Using Process Callback Functions" on page 25-20

Create sl_customization Function

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

Tip

- You can have more than one `sl_customization.m` file on your MATLAB path.

- Do not place an `sl_customization.m` file that customizes checks and folders in the Model Advisor in your root MATLAB folder or any of its subfolders, except for the *matlabroot*/work folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

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

Registering Checks and Process Callbacks

To register custom checks and process callbacks, the customization manager includes the following methods:

- addModelAdvisorCheckFcn (@*checkDefinitionFcn*)

  Registers the checks that 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 that you want to add to the Model Advisor as instances of the ModelAdvisor.Check class (see "Defining Custom Checks" on page 25-23).

- addModelAdvisorProcessFcn (@*modelAdvisorProcessFcn*)

  Registers the process callback function for the Model Advisor checks (see "Defining Startup and Post-Execution Actions Using Process Callback Functions" on page 25-20).

Caution The Model Advisor registers only one process callback function. If you have more than one sl_customization.m file on your MATLAB path, the Model Advisor registers the process callback function from the sl_customization.m file that has the highest priority.

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

Model Advisor Code Example: Registering Custom Checks and Process Callbacks

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

```
function sl_customization(cm)

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

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

> **Note** If you add custom tasks and folders within the sl_customization.m
> file, include methods for registering the tasks and folders in the
> sl_customization function. For more information, see "Registering Tasks
> and Folders" on page 26-14.

Defining Startup and Post-Execution Actions Using Process Callback Functions

The *process callback function* is an optional function that you use to configure
the Model Advisor 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 "Registering Checks and Process Callbacks" on page 25-18. The following
sections 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 that the Model Advisor analyzes.
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.

Model Advisor Code Example: Process Callback Function

The following code is an example of a process callback function that specifies actions in the configure stage, to make only custom checks visible. In the process_results stage, this function displays 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 window 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
```

Defining Custom Checks

In this section...

About Custom Checks

You can create a custom check to use in the Model Advisor. Creating custom checks provides you with the ability to specify which conditions and configuration settings the Model Advisor reviews.

You define custom checks in one or more functions that specify the properties of each instance of the ModelAdvisor.Check class. 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 "Registering Checks and Process Callbacks" on page 25-18.

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 26-16.

The following sections describe how to define custom checks.

Contents of Check Definitions

When you define a Model Advisor check, it contains the information listed in the following table.

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 a 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, see `ModelAdvisor.Check` and `ModelAdvisor.Task`.
	Tip When you add checks to the Model Advisor as tasks, 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.
Input Parameters (optional)	Adds input parameters that request input from the user. The Model Advisor uses the input to perform the check.

Contents	Description
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. You can define when to display a check, or whether a user can 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 ModelAdvisor.Check and ModelAdvisor.Task, the ModelAdvisor.Task properties take precedence, except for the Visible and LicenseName properties. For more information, see ModelAdvisor.Task.

Modify the behavior of the Visible, Enable, and Value properties in a process callback function (see "Defining Startup and Post-Execution Actions Using Process Callback Functions" on page 25-20). The following chart illustrates how these properties interact.
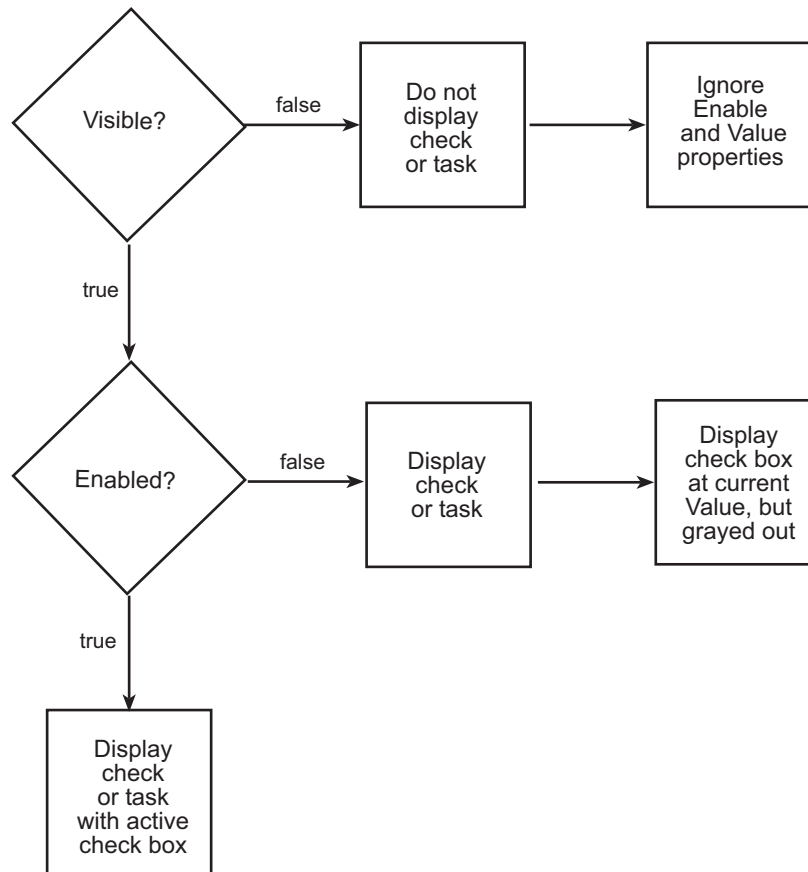
Defining Where Custom Checks Appear

Specify where the Model Advisor places custom checks using the following guidelines:

- To place a check in a new folder in the **Model Advisor** root, use the `ModelAdvisor.Group` class. See "Defining Custom Tasks" on page 26-15.

- To place a check in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See "Defining Custom Tasks" on page 26-15.

- To place a check in the **By Product** folder, use the
  `ModelAdvisor.Root.publish` method.

Model Advisor 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 25-34. The check definition function returns a cell array of custom checks to be added to the Model Advisor.

The check definitions in the example use the tasks described in "Defining Custom Tasks" on page 26-15.

```
% 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');
rec.CallbackContext = 'PostCompile';
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

With input parameters, the check author can request input from the user for a Model Advisor check. Define input parameters using the ModelAdvisor.InputParameter class inside a custom check function (see "Defining Custom Checks" on page 25-23). 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

Specify the layout of input parameters 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 documentation.

Model Advisor Code Example: Input Parameter Definition

The following is an example of defining input parameters that you add to a custom check. You must include input parameter definitions inside a custom check definition (see "Model Advisor Code Example: Check Definition Function" on page 25-27). 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.
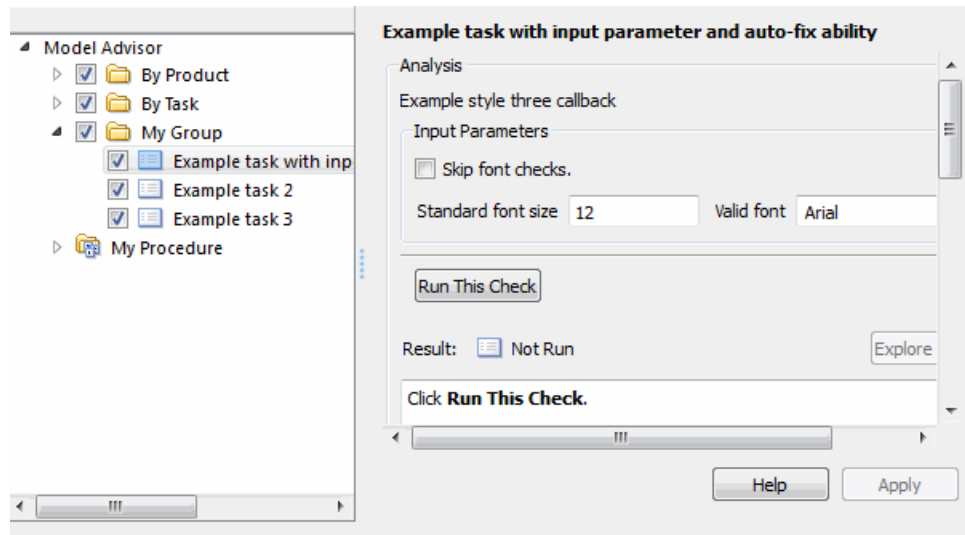
Defining Model Advisor Result Explorer Views

A *list view* provides a way for users to fix check warnings and failures using the Model Advisor Result Explorer. Creating a list view allows you to :

• Add the **Explore Result** button to the custom check in the Model Advisor window.

• Provide the information to populate the Model Advisor Result Explorer.

For information on using the Model Advisor Results Explorer, see "Batch-Fixing Warnings or Failures" in the Simulink documentation.

Define list views using the `ModelAdvisor.ListViewParameter` class inside a custom check function (see "Defining Custom Checks" on page 25-23). You must define one instance of this class for each list view that you want to add to a Model Advisor Result Explorer window.

Note You do not have to create list views for every custom check.

Model Advisor 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 25-43).

The following code, when included in a check definition function, adds the **Explore Result** button to the check in the Model Advisor.

```
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.

```
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

An *action* provides a way for you 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 25-23). 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 25-49).

Note

- Each check can contain only one action.

- You do not have to create actions for every custom check.

Model Advisor 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 "Model Advisor Code Example: Check Definition Function" on page 25-27).

The following code, when included in a check definition function, provides the information to populate the **Action** box in the Model Advisor.

```
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.

Creating Callback Functions and Results

About Callback Functions

A *callback function* specifies the actions that 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 a user runs the check. There are several types of callback functions:

- "Simple Check Callback Function" on page 25-35
- "Detailed Check Callback Function" on page 25-43
- "Check Callback Function with Hyperlinked Results" on page 25-45
- "Action Callback Function" on page 25-49

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. Use the Model Advisor Result Template API to format check results, as described in "Formatting Model Advisor Results" on page 25-50. 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 utilities and when to use them. In the Utility column, click the link 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: • 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.
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 recommend fixing an issue. The keyword for this callback function is StyleOne. The check definition requires this keyword (see "Defining Custom Checks" on page 25-23).

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.

Model Advisor Code Example: Informational Check Callback Function

The following code 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;
```

Model Advisor 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 actions 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);
```

Model Advisor 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 so you can associate text descriptions with one or more paragraphs of detailed information. The keyword for the detailed callback function is StyleTwo. The check definition requires this keyword (see "Defining Custom Checks" on page 25-23).

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. The check definition requires this keyword (see "Defining Custom Checks" on page 25-23).

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.
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 described 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 for a uniform appearance in the 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 for a uniform appearance in the 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);
```

In the Model Advisor, if you run **Example task with input parameter
and auto-fix ability** for the slvnvdemo_mdladv model, 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.

Model Advisor Code Example: Action Callback Function

The following is an example of an action callback function that fixes the optimization settings that the Model Advisor reviews as defined in "Model Advisor Code Example: Check With Subchecks and Actions" on page 25-41.

```
% 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
```

For 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 "Model Advisor Code Example: Input Parameter Definition" on page 25-29, review the customization source code in slvnvdemo_mdladv.

Formatting Model Advisor Results

- "Overview of Displaying Results" on page 25-50

- "Formatting Model Advisor Results" on page 25-51

- "Formatting Text" on page 25-51

- "Formatting Lists" on page 25-52

- "Formatting Tables" on page 25-52

- "Formatting Paragraphs" on page 25-53

- "Model Advisor Code Example: Formatted Output" on page 25-53

Overview of 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 25-35.

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

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, in the Constructor column, click the link.

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.

```
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(' for a uniform appearance in the model.');
```

```
result = [t1, t2, t3, t4, t5];
```

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.

```
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

Change table formatting using the `ModelAdvisor.Table` constructor. The following example code creates a subtable within a table.

```
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.

Model Advisor Code Example: Formatted Output

The following is the example from "Simple Check Callback Function" on page 25-35, 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 for 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
```

Excluding Blocks From Custom Checks

To save time during model development and verification, you might decide to exclude individual blocks from custom checks in a Model Advisor analysis. By modifying the sl_customization.m file to include the ModelAdvisor.Check.supportExclusion and Simulink.ModelAdvisor.filterResultWithExclusion functions, you can exclude your custom checks from:

- Simulink blocks
- Stateflow charts

This example shows how to exclude blocks from a custom check.

1 At the MATLAB command line, type slvnvdemo_mdladv.

2 In the model window, double-click **View demo sl_customization.m**.

3 To exclude the custom check **Check Simulink block font** from blocks during Model Advisor analysis, make three modifications to the sl_customization.m file.

   a Enable the **Check Simulink block font** check to support check exclusions by using the ModelAdvisor.Check.supportExclusion property. You can now exclude the check from model blocks. After rec.setInputParametersLayoutGrid([3 2]);, add rec.supportExclusion = true;. The check 1 section of the function defineModelAdvisorChecks now looks like:

```
% --- 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]);
rec.supportExclusion = true;
```

   b Use the Simulink.ModelAdvisor.filterResultWithExclusion function to filter model objects causing a check warning or failure with checks that have exclusions enabled. To do this, there are two locations in the

sl_customization.m file to modify, both in the [ResultDescription, ResultDetails] = SampleStyleThreeCallback(system) function:

- After both instances of searchResult = mdladvObj.filterResultWithExclusion(searchResult);, add searchResult = setdiff(allBlks, regularBlks);.

- In the first location, the function now looks like:

```
% find regular font name blocks
regularBlks = find_system(allBlks,'FontName',regularFontName);

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
if ~isempty(searchResult)
```

- In the second location, the function now looks like:

```
% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
if ~isempty(searchResult)
```

4 Save the sl_customization.m file. If you are asked if it is ok to overwrite the file, click **OK**.

5 In the model window, double-click **Launch Model Advisor**.

6 In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check fails.

7 In the Model Advisor window, click the **Enable Model Advisor highlighting** button (▥). The blocks causing the **Check Simulink block font** check failure are highlighted in yellow.

8 In the model window, right-click the X block and select **Model Advisor > Exclude block only > Check Simulink block font**.

9 In the Model Advisor Exclusion Editor, click **OK** to create the exclusion file. Additionally, in the Save Exclusion File as dialog box, click **Save** to create an exclusion file with the default name slvnvdemo_mdladv_exclusions.xml.

10 In the model window, right-click the Input block and select **Model Advisor > Exclude block only > Check Simulink block font**.

11 In the Model Advisor Exclusion Editor, click **OK** to update the exclusion file.

12 In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check now passes. In the right-pane of the Model Advisor window, you can see the **Check Exclusion Rules** that the Model Advisor during the analysis.

13 Close slvnvdemo_mdladv.

Related Examples
- "Running Model Advisor Checks"
- "Limiting the Scope of a Model Advisor Analysis By Excluding Gain and Outport Blocks" on page 22-11

More About
- "Limiting the Scope of a Model Advisor Analysis" on page 22-3
- "Consulting the Model Advisor"
- "Model Advisor Highlighting"
- Simulink.ModelAdvisor

Creating Custom Configurations by Organizing Checks and Folders

Overview of Creating Custom Configurations

About Creating Custom Configurations

The Simulink Verification and Validation product allows you to extend the capabilities of the Model Advisor. Using Model Advisor APIs and the Model Advisor Configuration Editor, you can:

- Define your own custom checks and write your own callback functions.

- Create custom configurations by organizing checks and folders.

- Create multiple custom configurations that you use for different projects or modeling guidelines, and switch between these configurations in the Model Advisor.

Creating Custom Configurations Workflow

To create a custom configuration with custom checks and checks MathWorks provides, see "Creating Configurations by Organizing Checks and Folders Workflow" on page 24-4.

Using the Model Advisor Configuration Editor Versus Customization File

The Model Advisor Configuration Editor is a GUI that expedites creating and deploying custom configurations. While you can organize Model Advisor configurations in a customization file, use the Model Advisor Configuration Editor to create custom configurations. For more details, see "Organizing Checks and Folders Using the Model Advisor Configuration Editor" on page 26-3.

Organizing Checks and Folders Using the Model Advisor Configuration Editor

In this section...
"Overview of the Model Advisor Configuration Editor" on page 26-3
"Starting the Model Advisor Configuration Editor" on page 26-9
"How To Organize Checks and Folders Using the Model Advisor Configuration Editor" on page 26-10

Overview of the Model Advisor Configuration Editor

When you start the Model Advisor Configuration Editor, two windows open; the Model Advisor Configuration Editor and the Model Advisor Check Browser. The Configuration Editor window consists of two panes: the Model Advisor Configuration Editor hierarchy and the Workflow. The Model Advisor Configuration Editor hierarchy lists the checks and folders in the current configuration. The Workflow on the right shows the common workflow you use to create a custom configuration.
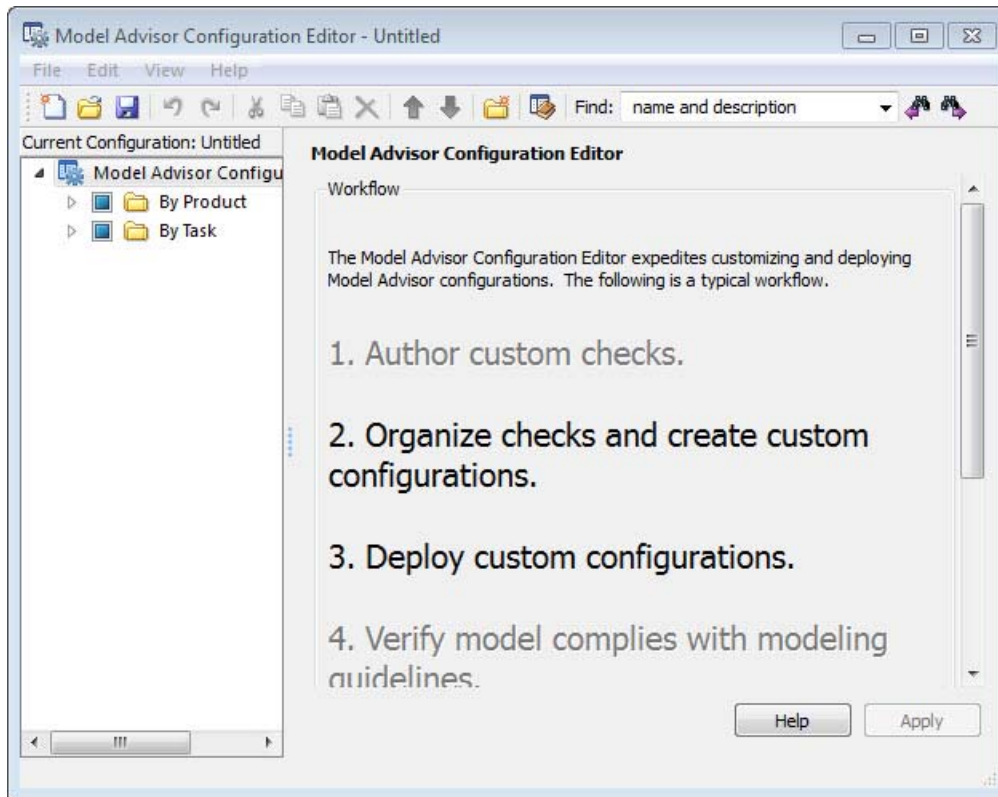
Model Advisor Configuration Editor

When you select a folder or check in the Model Advisor Configuration Editor hierarchy, the Workflow pane changes to display information about the check or folder. You can change the display name of the check or folder in this pane.
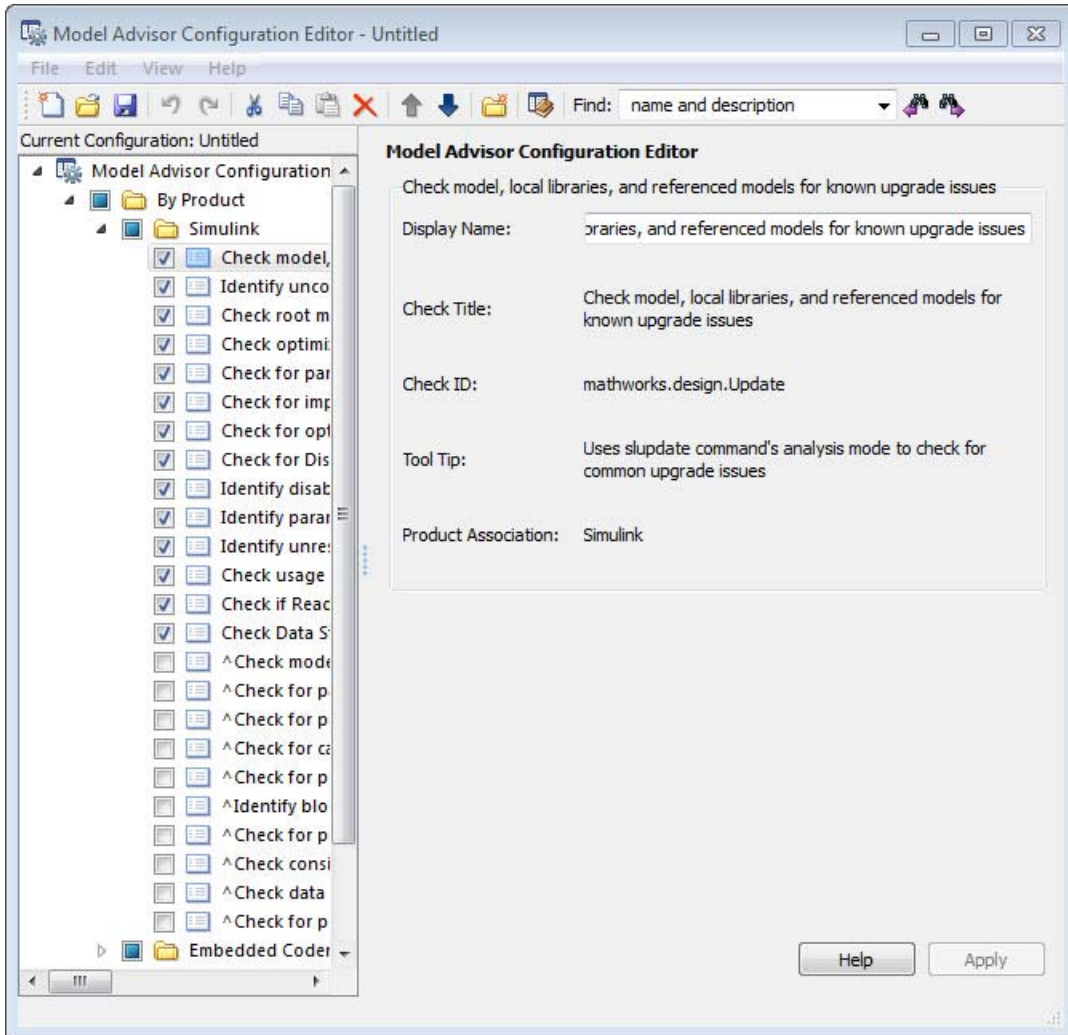
The Model Advisor Check Browser window includes a read-only list of available checks. If you delete a check in the Model Advisor Configuration Editor, you can retrieve a copy of it from the Model Advisor Check Browser.

Tip If you use a process callback function in a `sl_customization` file to hide checks and folders, the Model Advisor Configuration Editor and Model Advisor Check Browser do not display the hidden checks and folders. For a complete list of checks and folders, remove process callback functions and update the Simulink environment (see "Updating the Environment to Include Your sl_customization File" on page 26-22).
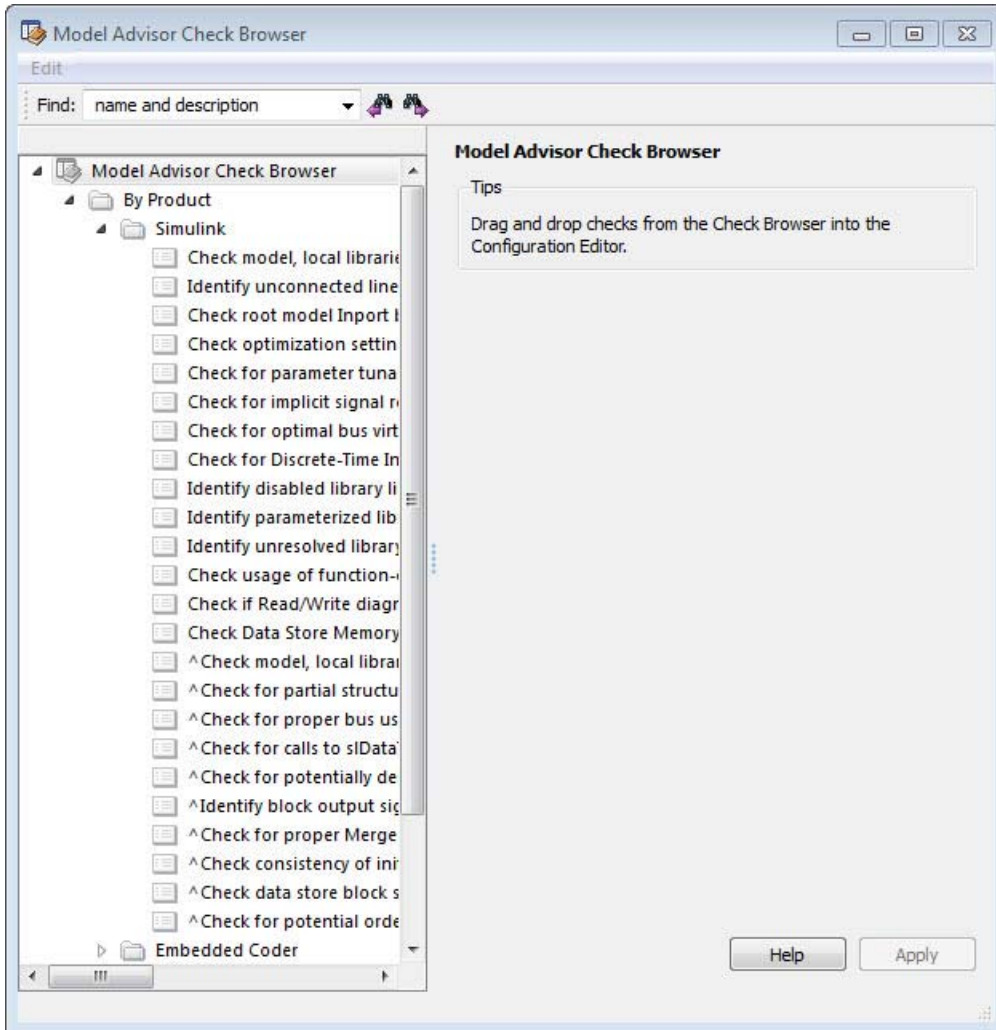
Model Advisor Check Browser

Using the Model Advisor Configuration Editor, you can perform the following actions.

To...	Select...
Create new configurations	**File > New**
Find checks and folders in the Model Advisor Check Browser	**View > Check Browser**
Add checks and folders to the configuration	**Edit > Copy** **Edit > Paste** **Edit > New folder** The check or folder and drag and drop
Remove checks and folders from the configuration	**Edit > Delete** **Edit > Cut**
Reorder checks and folders	**Edit > Move up** **Edit > Move down** The check or folder and drag and drop
Rename checks and folders **Note** The MathWorks folder display names are restricted. When you rename a folder, you cannot use the restricted display names.	The check or folder and edit **Display Name** in right pane.
Allow or gray out the check box control for checks and folders **Tip** This capability is equivalent to enabling checks, described in "Displaying and Enabling Checks" on page 25-25.	**Edit > Enable** **Edit > Disable**
Save the configuration as a MAT file for use and distribution	**File > Save** **File > Save As**
Set the configuration so it opens by default in the Model Advisor	**File > Set Current Configuration as Default**
Restore the MathWorks default configuration	**File > Restore Default Configuration**
Load and edit saved configurations	**File > Open**

Starting the Model Advisor Configuration Editor

Note

- Before starting the Model Advisor Configuration Editor, verify that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.

- The Model Advisor Configuration Editor uses the Simulink project (`slprj`) folder (for details about storing reports and other relevant information, see "Model Reference Simulation Targets") in the current folder. If this folder does not exist in the current folder, the Model Advisor Configuration Editor creates it.

---

1 To include custom checks in the new Model Advisor configuration, update the Simulink environment to include your `sl_customization.m` file. For more information, see "Updating the Environment to Include Your sl_customization File" on page 26-22.

2 Start the Model Advisor Configuration Editor.

To start the Model Advisor Configuration Editor...	Do this:
Programmatically	At the MATLAB command line, enter `Simulink.ModelAdvisor.openConfigUI`. For more information, see the `Simulink.ModelAdvisor` function reference page.
From the Model Advisor	**1** Start the Model Advisor. **2** Select **File > Open Configuration Editor**.

The Model Advisor Configuration Editor and Model Advisor Check Browser windows open.

3 Optionally, to edit an existing configuration in the Model Advisor Configuration Editor window:

  a Select **File > Open**.

  b In the Open dialog box, navigate to the configuration file that you want to edit.

  c Click **Open**.

How To Organize Checks and Folders Using the Model Advisor Configuration Editor

The following tutorial steps you through creating a custom configuration.

1 Open the Model Advisor Configuration Editor at the MATLAB command line by entering `Simulink.ModelAdvisor.openConfigUI` . For more options, see "Starting the Model Advisor Configuration Editor" on page 26-9.

2 In the Model Advisor Configuration Editor, in the left pane, delete the **By Product** and **By Task** folders, to start with a blank configuration.

3 Select the root node which is labeled Model Advisor Configuration Editor.

4 In the toolbar, click the **New Folder** button to create a folder.

5 In the left pane, select the new folder.

6 In the right pane, edit **Display Name** to rename the folder. For the purposes of this tutorial, rename the folder to **Review Optimizations**.

7 In the Model Advisor Check Browser window, in the **Find** field, enter `optimization` to find **Simulink > Check optimization settings**.

8 Drag and drop **Check optimization settings** into **Review Optimizations**.

9 In the Model Advisor Check Browser window, find **Simulink Verification and Validation > Modeling Standards > DO-178B Checks > Check safety-related optimization settings**.

10 Drag and drop **Check safety-related optimization settings** into **Review Optimizations**.

11 In the Model Advisor Configuration Editor window, expand **Review Optimizations**.

12 Rename **Check optimization settings** to **Check Simulink optimization settings**.

13 Select **File > Save As** to save the configuration.

14 Name the configuration optimization_configuration.mat.

15 Close the Model Advisor Configuration Editor window.

Tip To move a check to the first position in a folder:

1 Drag the check to the second position.

2 Right-click the check and select **Move up**.

Organizing Checks and Folders Within a Customization File

In this section...
"Customization File Overview" on page 26-12
"Register Tasks and Folders" on page 26-13
"Defining Custom Tasks" on page 26-15
"Defining Custom Folders" on page 26-18
"Demo and Code Example" on page 26-20

Note While you can organize checks and folders within a customization file, use the Model Advisor Configuration Editor. For more information, see "Using the Model Advisor Configuration Editor Versus Customization File" on page 26-2.

Customization File Overview

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	Required or Optional
sl_customization()	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at startup (see "Registering Checks and Process Callbacks" on page 25-18).	Required for all customizations to the Model Advisor.
One or more check definitions	Defines all custom checks (see "Defining Custom Checks" on page 25-23).	Required for custom checks and to add custom checks to the **By Product** folder.

Function	Description	Required or Optional
One or more task definitions	Defines all custom tasks (see "Defining Custom Tasks" on page 26-15).	Required to add custom checks to the Model Advisor, except when adding the checks to the **By Product** folder. Write one task for each check that you add to the Model Advisor.
One or more groups	Defines all custom groups (see "Defining Custom Tasks" on page 26-15).	Required to add custom tasks to new folders in the Model Advisor, except when adding a new subfolder to the **By Product** folder. Write one group definition for each new folder.
One process callback function	Specifies actions that Simulink performs at startup and post-execution time (see "Defining Startup and Post-Execution Actions Using Process Callback Functions" on page 25-20).	Optional.

Register Tasks and Folders

- "Create sl_customization Function" on page 26-13

- "Registering Tasks and Folders" on page 26-14

Create sl_customization Function

To add tasks and folders to the Model Advisor, create the `sl_customization.m` file on your MATLAB path. Then create the `sl_customization()` function in the `sl_customization.m` file on your MATLAB path.

Tip

- 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 folder or any of its subfolders, except for the *matlabroot*/work folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.

The `sl_customization` function accepts one argument, a customization manager object, 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 Tasks and Folders

The customization manager provides the following methods for registering custom tasks and folders:

- addModelAdvisorTaskFcn (@*factorygroupDefinitionFcn*)

  Registers the tasks that 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" on page 26-15).

- addModelAdvisorTaskAdvisorFcn (@*taskDefinitionFcn*)

  Registers the tasks and folders that you define in *taskDefinitionFcn* to the folder in the Model Advisor 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. Simulink adds the checks and folders to the Model Advisor as instances of the `ModelAdvisor.Task` or `ModelAdvisor.Group` classes (see "Defining Custom Tasks" on page 26-15).

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

Model Advisor Code Example: Registering Custom Tasks and Folders.
The following code example registers custom tasks and folders:

```
function sl_customization(cm)

% register custom factory group
cm.addModelAdvisorTaskFcn(@defineModelAdvisorTasks);

% register custom tasks.
cm.addModelAdvisorTaskAdvisorFcn(@defineTaskAdvisor);
```

Note If you add custom checks and process callbacks within the `sl_customization.m` file, include methods for registering the checks and process callbacks in the `sl_customization` function. For more information, see "Registering Checks and Process Callbacks" on page 25-18.

Defining Custom Tasks

- "Adding a Check to Custom or Multiple Folders Using Tasks" on page 26-16
- "Creating Custom Tasks Using MathWorks Checks" on page 26-16
- "Displaying and Enabling Tasks" on page 26-17
- "Defining Where Tasks Appear" on page 26-17
- "Model Advisor Code Example: Task Definition Function" on page 26-17

Adding a Check to Custom or Multiple Folders Using Tasks

You can use custom tasks for adding checks to the Model Advisor, 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. Define one instance of this class for each custom task that you want to add to the Model Advisor. Then register the custom task, as described in "Register Tasks and Folders" on page 26-13. The following sections 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 25-23.

2 Register a task wrapper for the check, as described in "Register Tasks and Folders" on page 26-13.

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. When you add the checks as custom tasks, you identify checks by the check ID.

To find MathWorks check IDs:

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 they do for checks (see "Displaying and Enabling Checks" on page 25-25).

Defining Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor 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 Folders" on page 26-18.

- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See "Defining Custom Folders" on page 26-18.

Model Advisor Code Example: Task Definition Function

The following is an example of a task definition function. This function defines three tasks. The tasks are derived from the checks defined in "Model Advisor Code Example: Check Definition Function" on page 25-27.

For an example of placing these tasks into a custom group, see "Model Advisor Code Example: Group Definition" on page 26-19.

```
% 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 the Model Advisor under 'Model Advisor' (root)
mdladvRoot.publish(MAG);
```

Defining Custom Folders

- "About Custom Folders" on page 26-18
- "Adding Custom Folders" on page 26-19
- "Defining Where Custom Folders Appear" on page 26-19
- "Model Advisor Code Example: Group Definition" on page 26-19

About Custom Folders

Use folders 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 26-16.

Define one instance of the group classes for each folder that you want to add to the Model Advisor. Then register the custom folder, as described in "Register Tasks and Folders" on page 26-13. The following sections 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, as described in "Defining Custom Folders" on page 26-18.

Defining Where Custom Folders Appear

You can specify the location of custom folders within the Model Advisor 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, use the `ModelAdvisor.Root.publish` method within a custom check. For more information, see "Defining Where Custom Checks Appear" on page 25-26.

Model Advisor Code Example: Group Definition

The following is an example of a group definition. The definition places the tasks defined in "Model Advisor Code Example: Task Definition Function" on

page 26-17 inside a folder called **My Group** under the **Model Advisor** root. The task definition function includes this group definition.

```
% 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 the Model Advisor under 'Model Advisor' (root)
mdladvRoot.publish(MAG);
```

The following is an example of a factory group definition function. The definition places the checks defined in "Model Advisor Code Example: Check Definition Function" on page 25-27 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
```

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

- Custom folders for grouping the checks

- Custom procedures

- 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.

Verifying and Using Custom Configurations

In this section...

"Updating the Environment to Include Your sl_customization File" on page 26-22

"Verifying Custom Configurations" on page 26-22

Updating the Environment to Include Your sl_customization File

When you start Simulink, it reads customization (`sl_customization.m`) files. If you change the contents of your customization file, update your environment by performing these tasks:

1 If you previously started the Model Advisor:

   a Close the model from which you started the Model Advisor

   b Clear the data associated with the previous Model Advisor session by removing the `slprj` folder from your working folder.

2 At the MATLAB command line, enter:

```
sl_refresh_customizations
```

3 Open your model.

4 Start the Model Advisor.

Verifying Custom Configurations

To verify a custom configuration:

1 If you created custom checks, or created the custom configuration using the `sl_customization` method, update the Simulink environment. For more information, see "Updating the Environment to Include Your sl_customization File" on page 26-22.

2 Open a model.

3 From the model window, start the Model Advisor.

4 Select **File > Load Configuration**. If you see a warning that the Model Advisor report corresponds to a different configuration, click **Load** to continue.

5 In the Open dialog box, navigate to and select your custom configuration. For example, if you created the custom configuration in "How To Organize Checks and Folders Using the Model Advisor Configuration Editor" on page 26-10, select `optimization_configuration.mat`.

6 When the Model Advisor reopens, verify that the configuration contains the new folders and checks. For example, the **Review Optimizations** folder and the **Check Simulink optimization settings** and **Check safety-related optimization settings** checks.

7 Optionally, run the checks.

Creating Procedural-Based Model Advisor Configurations

Overview of Procedural-Based Model Advisor Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

Creating Procedural-Based Model Advisor Configurations Workflow

For the workflow to create procedural-based configurations, see "Creating Procedural-Based Configurations Workflow" on page 24-5.

Creating Procedures

In this section...
"What Is a Procedure?" on page 27-3
"Creating Procedures Using the Procedures API" on page 27-3
"Defining Procedures" on page 27-3

What Is a Procedure?

A procedure is a series of checks. The checks in a procedure depend on passing the previous checks. If Check A is the first check in a procedure and Check B follows, the Model Advisor does not run Check B until Check A passes. Checks A and B can be either custom or provided by MathWorks.

You create procedures with the `ModelAdvisor.Procedure` class API. You first add the checks to tasks, which are wrappers for the checks. The tasks are added to procedures. See "Creating Procedures Using the Procedures API" on page 27-3.

When creating procedural checks, be aware of potential conflicts with the checks. Verify that it is possible to pass both checks.

Creating Procedures Using the Procedures API

You use the `ModelAdvisor.Procedure` class to create procedural checks.

1 Add each check to a task using the `ModelAdvisor.Task.setCheck` method. The task is a wrapper for the check. You cannot add checks directly to procedures. For more information, see "Defining Custom Tasks" on page 26-15.

2 Add each task to a procedure using the `ModelAdvisor.Procedure.addTask` method.

Defining Procedures

You define procedures in a procedure definition function that specifies the properties of each instance of the `ModelAdvisor.Procedure` class.

Define one instance of the procedure class for each procedure that you want to add to the Model Advisor. Then register the procedure using the `ModelAdvisor.Root.register` method.

Adding Subprocedures and Tasks to Procedures

You can add subprocedures or a tasks to a procedure. The tasks are wrappers for checks.

- Use the `ModelAdvisor.Procedure.addProcedure` method to add a subprocedure to a procedure.

- Use the `ModelAdvisor.Procedure.addTask` method to add a task to a procedure.

Defining Where Procedures Appear

You can specify where the Model Advisor places a procedure using the `ModelAdvisor.Group.addProcedure` method.

Model Advisor Code Example: Procedure Definition

The following code example adds procedures to a group:

```
%Create three procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure3');

%Create a group
MAG = ModelAdvisor.Group('com.mathworks.sample.myGroup');

%Add the three procedures to the group
addProcedure(MAG, MAP1);
addProcedure(MAG, MAP2);
addProcedure(MAG, MAP3);

%register the group and and procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAG);
mdladvRoot.register(MAP1);
```

```
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

The following code example adds subprocedures to a procedure:

```
%Create a procedures
MAP = ModelAdvisor.Procedure('com.mathworks.example.Procedure');

%Create 3 sub procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub1');
MAP2=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub2');
MAP3=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub3');

%Add sub procedures to procedure
addProcedure(MAP, MAP1);
addProcedure(MAP, MAP2);
addProcedure(MAP, MAP3);

%register the procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAP);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

The following code example adds tasks to a procedure:

```
%Create three tasks
MAT1=ModelAdvisor.Task('com.mathworks.tasksample.myTask1');
MAT2=ModelAdvisor.Task('com.mathworks.tasksample.myTask2');
MAT3=ModelAdvisor.Task('com.mathworks.tasksample.myTask3');

%Create a procedure
MAP = ModelAdvisor.Procedure('com.mathworks.tasksample.myProcedure');

%Add the three tasks to the procedure
addTask(MAP, MAT1);
addTask(MAP, MAT2);
addTask(MAP, MAT3);

%register the procedure and tasks
```

```
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAP);
mdladvRoot.register(MAT1);
mdladvRoot.register(MAT2);
mdladvRoot.register(MAT3);
```

Creating a Procedural-Based Model Advisor Configuration

In this example, you examine a procedural-based configuration.

1 At the MATLAB command line, type slvnvdemo_mdladv.

2 In the Simulink model window, select **View demo sl_customization.m**. The sl_customization.m file opens in the MATLAB Editor window.

The file contains three checks created in the function defineModelAdvisorChecks:

- ModelAdvisor.Check('com.mathworks.sample.Check1') - Checks Simulink block fonts.

- ModelAdvisor.Check('com.mathworks.sample.Check2') - Checks Simulink window screen color.

- ModelAdvisor.Check('com.mathworks.sample.Check3') - Checks model optimization settings.

Each check has a set of fix actions.

3 In the sl_customization.m file, examine the function defineTaskAdvisor.

- The ModelAdvisor.Procedure class API creates procedures My Procedure and My sub_Procedure:

```
% Define procedures
MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');
MAP.DisplayName='My Procedure';

MAP_sub = ModelAdvisor.Procedure('com.mathworks.sample.sub_ProcedureSample');
MAP_sub.DisplayName='My sub_Procedure';
```

- The ModelAdvisor.Task class API creates tasks MAT4, MAT5, and MAT6. The ModelAdvisor.Task.setCheck method adds the checks to the tasks:

```
% Define tasks
MAT4 = ModelAdvisor.Task('com.mathworks.sample.TaskSample4');
MAT4.DisplayName='Check Simulink block font';
MAT4.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT4);
```

```
MAT5 = ModelAdvisor.Task('com.mathworks.sample.TaskSample5');
MAT5.DisplayName='Check Simulink window screen color';
MAT5.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT5);

MAT6 = ModelAdvisor.Task('com.mathworks.sample.TaskSample6');
MAT6.DisplayName='Check model optimization settings';
MAT6.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT6);
```

- The `ModelAdvisor.Procedure.addTask` method adds task MAT4
  to `My Procedure` and tasks MAT5 and MAT6 to `My sub_Procedure`.
  The `ModelAdvisor.Procedure.addProcedure` method adds `My
  sub_Procedure` to `My Procedure`:

```
% Add tasks to procedures:
% Add Task4 to MAP
MAP.addTask(MAT4);
% Now Add Task5 and Task6 to MAP_sub
MAP_sub.addTask(MAT5);
MAP_sub.addTask(MAT6);
% Include the Sub-Procedure in the Procedure
MAP.addProcedure(MAP_sub);
```

4 From the model window, select **Tools > Model Advisor** to open the Model
Advisor.

5 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The
Model Advisor window opens. It might take a few minutes to open.

6 In the left pane, expand **My Procedure > My sub_Procedure**. The
Check Simulink block font check is in the My Procedure folder. My
sub_Procedure contains Check Simulink window screen color and Check
model optimization settings.
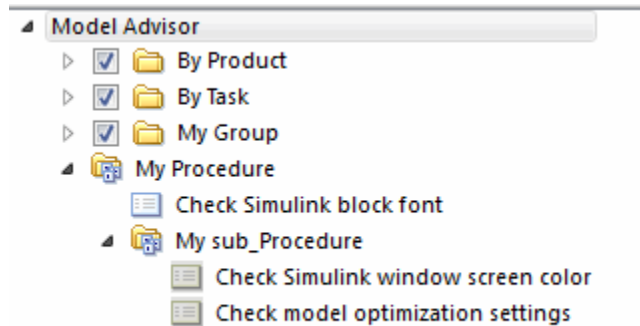
7 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run to Failure**. The Model Advisor Check Simulink block font check fails. The Model Advisor does not check the remaining two checks in the My sub_Procedure folder. Running the checks in the My sub_Procedure folder is dependent on passing the Check Simulink block font check.

8 In the **Action** section of the Model Advisor dialog box, click **Fix block fonts**.

9 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run to Failure**. The Check Simulink block font check passes. The Model Advisor runs the Check Simulink window screen color check. This check fails and the Model Advisor stops checking.

10 In the **Action** section of the Model Advisor dialog box, click **Fix window screen color**.

11 In the left pane of the Model Advisor, select My sub_Procedure. In the right pane of the Model Advisor, click **Run to Failure**. The Check Simulink window screen color check passes. The Model Advisor runs the Check model optimization settings check. This check warns.

12 In the **Action** section of the Model Advisor dialog box, click **Fix model optimization settings**.

13 In the left pane of the Model Advisor, select Check model optimization settings. In the right pane of the Model Advisor, click **Run This Task**. The Check model optimization settings check passes.

28
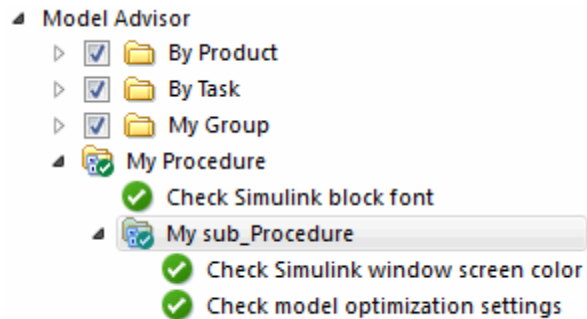
Deploying Custom Configurations

Overview of Deploying Custom Configurations

In this section...
"About Deploying Custom Configurations" on page 28-2
"Deploying Custom Configurations Workflow" on page 28-2

About Deploying Custom Configurations

When you create a custom configuration, often you *deploy* the custom configuration to your development group. Deploying the custom configuration allows your development group to review models using the same checks.

After you create a custom configuration, you can use it in the Model Advisor, or deploy the configuration to your users. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

Deploying Custom Configurations Workflow

When you deploy custom configurations, you:

1 Optionally author custom checks, as described in Chapter 25, "Authoring Custom Checks".

2 Organize checks and folders to create custom configurations, as described in Chapter 26, "Creating Custom Configurations by Organizing Checks and Folders".

3 Deploy the custom configuration to your users, as described in "How to Deploy Custom Configurations" on page 28-3.

How to Deploy Custom Configurations

To deploy a custom configuration:

1 Determine which files to distribute. You might need to distribute more than one file.

If You...	Using the...	Distribute...
Created custom checks	Customization file	• `sl_customization.m` • Files containing check and action callback functions (if separate)
Organized checks and folders	Model Advisor Configuration Editor	Configuration MAT file
	Customization file	`sl_customization.m`

2 Distribute the files and tell the user to include these files on the MATLAB path.

3 Instruct the user to load the custom configuration. For more details, see "Manually Loading and Setting the Default Configuration" on page 28-4 or "Automatically Loading and Setting the Default Configuration" on page 28-5.

Manually Loading and Setting the Default Configuration

When you use the Model Advisor, you can load any configuration. Once you load a configuration, you can set it so that the Model Advisor use that configuration every time you open the Model Advisor.

1 Open the Model Advisor.

2 Select **File > Load Configuration**.

3 In the Open dialog box, navigate to and select the configuration file that you want to edit.

4 Click **Open**.

Simulink reloads the Model Advisor using the new configuration.

5 Optionally, when the Model Advisor opens, set the current configuration as the default by selecting **File > Set Current Configuration as Default**.

Automatically Loading and Setting the Default Configuration

When you use the Model Advisor, you can automatically set the default configuration by modifying an `sl_customization.m` file. For more information on creating the `sl_customization.m` file, see "Registering Checks and Process Callbacks" on page 25-18.

1 Place a configuration MAT file on your MATLAB path. For more information on MAT files, see "Organizing Checks and Folders Using the Model Advisor Configuration Editor" on page 26-3

2 Modify your `sl_customization.m` file by adding the function:

```
function [checkCellArray taskCellArray] = ModelAdvisorProcessFunction ...
 (stage, system, checkCellArray, taskCellArray)
    switch stage
        case 'configure'
    ModelAdvisor.setConfiguration('qeAPIConfigFilePath.mat');
end
```

In the function, replace *qeAPIConfigFilePath.mat* with the name of the configuration MAT file in step 1.

3 The `sl_customization.m` file is loaded every time you start the Model Advisor, using *qeAPIConfigFilePath.mat* as the default configuration. For more information, see "Updating the Environment to Include Your sl_customization File" on page 26-22.

Tip You can restore the MathWorks default configuration by selecting **File > Restore Default Configuration**.

Examples

Use this list to find examples in the documentation.

Requirements Management Interface

Requirements Management Interface (DOORS Version)

Component Verification

Verification Manager

Model Coverage

Model Advisor Check

Model Advisor Organization

Index