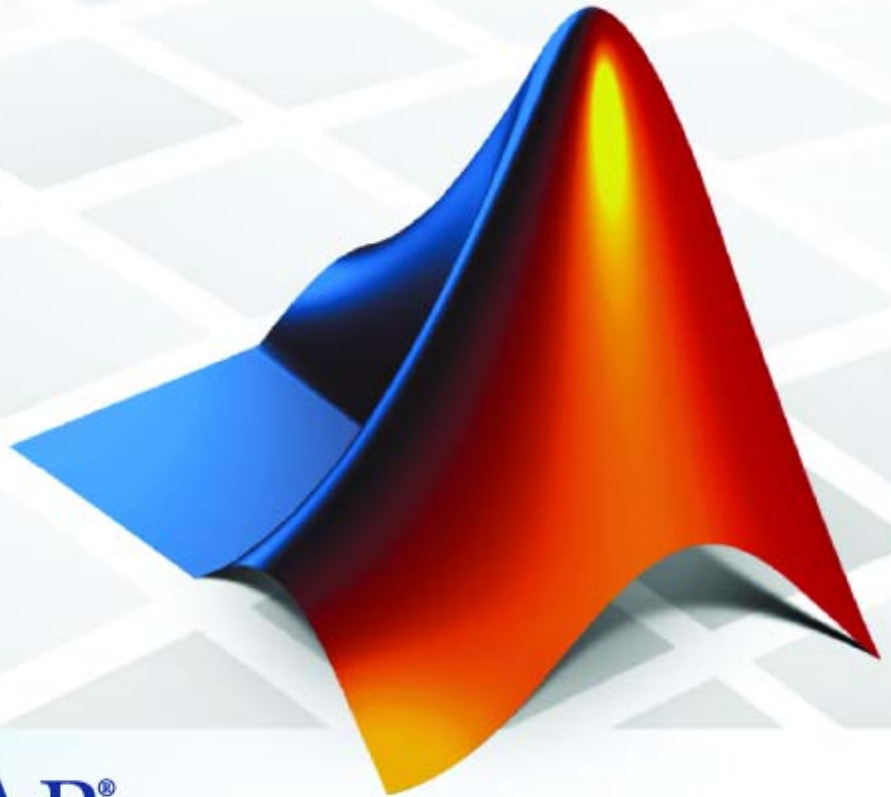


Simulink® Verification and Validation 2

User's Guide



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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 2004–2007 by The MathWorks, Inc.

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

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

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

DOORS is a registered trademark of Telelogic AB.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

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

Getting Started

1

What Is Simulink Verification and Validation?	1-2
System Requirements	1-3
Operating System Requirements	1-3
Product Requirements	1-3
Organization of This User's Guide	1-4

Managing Model Requirements

2

What Is the Requirements Management Interface? ...	2-3
Configuring the Requirements Management Interface	2-4
Adding and Viewing Requirement Links	2-5
Object and Document Types	2-5
Adding Requirement Links to an Object	2-8
Viewing Requirements Documents	2-13
Resolving the Document Path	2-15
Adding Requirement Links to Multiple Objects	
Simultaneously	2-17
Selection-Based Linking	2-20
Linking to Custom Types of Requirements	
Documents	2-28
Why Create a Custom Link Type?	2-28
Custom Link Type Registration	2-29
Built-In Link Types	2-29

Link Properties	2-30
Link Type Properties	2-30
Creating a Custom Link Requirement Type	2-32
Navigating to Simulink from External Documents	2-42
Viewing Objects with Requirement Links	2-45
Generating a Requirements Report	2-48
Displaying the System Requirements in a Diagram ...	2-50
About the System Requirements Block	2-50
Adding the System Requirements Block	2-50
Renaming the System Requirements Block	2-53
Changing Fonts for the System Requirements Block	2-54
Including Requirements with Generated Code	2-56

Managing Model Requirements with DOORS

3

What Is the Requirements Management Interface for DOORS?	3-2
Configuring the Requirements Management Interface for DOORS	3-3
Before You Begin	3-3
Installing DOORS Before RMI	3-3
Installing DOORS After RMI	3-3
Upgrading DOORS	3-4
Manual Installation for DOORS	3-4
Starting the Requirements Management Interface for DOORS	3-6
Linking Objects to DOORS Requirements	3-8
About Linkages Between a Simulink Model and DOORS ..	3-8
Creating a DOORS Requirement Object	3-8

Linking a Simulink or Stateflow Object to a DOORS Requirement	3-10
Synchronizing DOORS with the Simulink Model	3-13
About Module Synchronization	3-13
Synchronizing a Model with DOORS	3-14
Customizing the Level of Synchronization Detail	3-16
Customizing the DOORS Synchronization Settings	3-21
Linking Requirements to the DOORS Synchronized Module	3-23
Navigating Between Model Objects and DOORS	3-25
Viewing Model Elements with Requirements	3-25
Navigating from Simulink to DOORS	3-27
Navigating from DOORS to Simulink	3-29

Managing Model Verification Blocks

4

Using Model Verification Blocks	4-2
Using the Verification Manager	4-7
What Is the Verification Manager?	4-7
Opening the Verification Manager	4-7
Enabling and Disabling Model Verification Blocks with the Verification Manager	4-15
Using Enabling and Disabling Tools in the Verification Manager	4-20
Managing Verification Requirements	4-24

Using Model Coverage

5

Introduction to Model Coverage	5-3
---	------------

What Is Model Coverage?	5-3
How Model Coverage Works	5-3
Types of Model Coverage	5-3
Blocks That Receive Model Coverage	5-5
Using Model Coverage	5-8
Basic Workflow for Using Model Coverage	5-8
Creating and Running Test Cases	5-8
Specifying Model Coverage Reporting Options	5-12
The Coverage Settings Dialog Box	5-12
Coverage Tab	5-13
Results Tab	5-14
Report Tab	5-15
Options Tab	5-19
Understanding Model Coverage Reports	5-21
About Model Coverage Reports	5-21
Summary Report Section	5-21
Details Report Section	5-22
Decisions Analyzed Table	5-24
Conditions Analyzed Table	5-24
MC/DC Analysis Table	5-25
N-Dimensional Lookup Table Report	5-27
Signal Range Analysis Report	5-33
Colored Simulink Diagram Coverage Display	5-36
How Model Coverage Highlighting Works	5-36
Enabling the Colored Diagram Display	5-36
Displaying Model Coverage with Model Coloring	5-37
Accessing Coverage Information for Colored Blocks	5-39
Using Model Coverage Commands	5-41
About Model Coverage Commands	5-41
Creating Tests with cvtest	5-41
Running Tests with cvsim	5-43
Producing HTML Reports with cvhtml	5-44
Saving Test Runs to a File with cvsave	5-44
Loading Stored Coverage Test Results with cvload	5-45

Coverage Script Example	5-46
Model Coverage for Referenced Models	5-47
Introduction	5-47
Creating a Test Group with cv.cvtestgroup	5-50
Running Tests with cvsimref	5-50
Extracting Results from cv.cvdatagroup	5-51
Model Coverage for Embedded MATLAB Function	
Blocks	5-52
Types of Model Coverage in Embedded MATLAB Function	
Blocks	5-52
Creating a Model with Embedded MATLAB Function Block	
Decisions	5-53
Understanding Embedded MATLAB Function Block Model	
Coverage	5-56

Customizing Model Advisor

6

The Customization Process	6-3
Demo and Code Example	6-4
Creating Callback Functions for Checks	6-5
About Check Callback Functions	6-5
Simple Check Callback Function	6-5
Detailed Check Callback Function	6-6
Check Callback Function with Hyperlinked Results	6-8
Defining Custom Checks	6-13
About Custom Checks	6-13
Properties of Model Advisor Checks	6-13
How Visible, Enable, and Value Properties Interact	6-17
Code Example: Check Definition Function	6-17
Defining Custom Tasks	6-19
About Custom Tasks	6-19

Properties of Model Advisor Tasks	6-19
How Visible, Enable, and Value Properties Interact for Tasks	6-20
Code Example: Task Definition Function	6-21
Defining a Process Callback Function	6-22
About Process Callback Functions	6-22
Process Callback Function Arguments	6-22
Code Example: Process Callback Function	6-23
Formatting Model Advisor Outputs	6-25
What Is the Model Advisor Formatting API?	6-25
Formatting Text	6-25
Formatting Lists	6-26
Formatting Tables	6-27
Formatting Paragraphs	6-27
Code Example: Model Advisor Formatted Output	6-28
Registering Custom Checks and Tasks	6-29
About Custom Checks and Tasks Registration	6-29
Methods for Registering Custom Checks and Tasks	6-30
Code Example: Methods for Registering Custom Checks and Tasks	6-30

Functions — By Category

7

Requirements Management Interface	7-2
Model Coverage	7-3
Model Advisor Formatting API	7-4

Functions — Alphabetical List

8

Blocks — Alphabetical List

9

Model Advisor Checks

10

Simulink Verification and Validation Checks	10-2
Modeling Standards Overview	10-3
DO-178B Checks Overview	10-3
Check safety-related optimization settings	10-3
Check safety-related diagnostic settings for solvers	10-7
Check safety-related diagnostic settings for sample time ..	10-9
Check safety-related diagnostic settings for signal data ..	10-11
Check safety-related diagnostic settings for parameters ..	10-14
Check safety-related diagnostic settings for data used for debugging	10-16
Check safety-related diagnostic settings for data store memory	10-17
Check safety-related diagnostic settings for type conversions	10-19
Check safety-related diagnostic settings for signal connectivity	10-20
Check safety-related diagnostic settings for bus connectivity	10-22
Check safety-related diagnostic settings that apply to function-call connectivity	10-24
Check safety-related diagnostic settings for compatibility	10-25
Check safety-related diagnostic setting for model referencing	10-26
Check safety-related model referencing settings	10-29
Check safety-related code generation settings	10-31
Check for proper usage of For Iterator blocks	10-37
Check for proper usage of While Iterator blocks	10-37
Display model version information	10-38

Check for proper usage of blocks that compute absolute values	10-39
Check for proper usage of Relational Operator blocks	10-40

Examples

A

Requirements Management Interface	A-2
Requirements Management Interface (DOORS Version)	A-2
Verification Manager	A-2
Model Coverage	A-2

Index

Getting Started

Simulink® Verification and Validation uses component tools that contribute to the work of certifying the correct design, implementation, and testing of Simulink models. Use the following topics to introduce yourself to Simulink Verification and Validation.

What Is Simulink Verification and Validation? (p. 1-2)	Tells you why you want to use Simulink Verification and Validation in your models
System Requirements (p. 1-3)	Lists system requirements necessary to qualify for installation
Organization of This User's Guide (p. 1-4)	Describes the expected background for using Simulink Verification and Validation and the organization of this guide

What Is Simulink Verification and Validation?

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

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

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

System Requirements

In this section...
“Operating System Requirements” on page 1-3
“Product Requirements” on page 1-3

Operating System Requirements

Simulink Verification and Validation works with the following operating systems:

- Microsoft Windows 2000 and Windows XP systems
- UNIX systems where MATLAB® supports Java (for HTML-based requirements documents only)

Product Requirements

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

- MATLAB
- Simulink

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

- Stateflow

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

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

Organization of This User's Guide

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

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

or code generation. You can write custom checks, tasks, and callback functions, as described in the following chapter:

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

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

- Chapter 7, “Functions — By Category” — Provides a categorical list of functions used in executing and managing model coverage tests and reports from MATLAB. Automate your model coverage tests with scripts of MATLAB commands calling these functions.
- Chapter 8, “Functions — Alphabetical List” — Provides an alphabetical reference of functions used in executing and managing model coverage tests and reports from MATLAB.
- Chapter 9, “Blocks — Alphabetical List” — Provides reference information for the Simulink Verification and Validation library, which currently contains only one block, System Requirements. This block lets you list all the requirements for a model or subsystem on its Simulink diagram.

Managing Model Requirements

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

What Is the Requirements Management Interface? (p. 2-3)

Introduces you to the Requirements Management Interface for linking requirements documents to model elements

Configuring the Requirements Management Interface (p. 2-4)

Describes what you need to do before using the Requirements Management Interface

Adding and Viewing Requirement Links (p. 2-5)

Shows you how to link model elements with requirements documents, and modify both links and documents

Linking to Custom Types of Requirements Documents (p. 2-28)

Shows you how to register additional types of requirements documents and create links to them

Viewing Objects with Requirement Links (p. 2-45)

Shows you how to provide meaningful displays of model elements that are linked to requirements documents

Generating a Requirements Report (p. 2-48)

Shows you how to generate a report on all the requirements associated with a model and its blocks

Displaying the System Requirements in a Diagram (p. 2-50)	Shows you how to display the system requirements directly in your Simulink diagram
Including Requirements with Generated Code (p. 2-56)	Shows you how to specify that requirements descriptions are included in generated code for your model

What Is the Requirements Management Interface?

The Requirements Management Interface (RMI) allows you to associate requirements with Simulink models and Stateflow diagrams. Simulink and Stateflow requirements have the following parts:

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

Use the Requirements Management Interface to

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

Configuring the Requirements Management Interface

Before you start using the Requirements Management Interface, in the MATLAB Command Window type

```
rmi setup
```

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

Adding and Viewing Requirement Links

In this section...

“Object and Document Types” on page 2-5

“Adding Requirement Links to an Object” on page 2-8

“Viewing Requirements Documents” on page 2-13

“Resolving the Document Path” on page 2-15

“Adding Requirement Links to Multiple Objects Simultaneously” on page 2-17

“Selection-Based Linking” on page 2-20

Object and Document Types

You can add requirements to the following types of objects:

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

Note You can add requirements to top-level reference blocks but not to their contents. For example, if you copy a subsystem consisting of multiple blocks from a library, you will be able to add requirements to the subsystem block in your model, but not to its component blocks.

The Requirements Management Interface supports the following built-in types of requirements documents:

- Text
- HTML
- Microsoft Word
- Microsoft Excel

- PDF

You can also link to a DOORS item (see “Linking Objects to DOORS Requirements” on page 3-8), or register your own custom type of documents to link to (see “Linking to Custom Types of Requirements Documents” on page 2-28).

Location Types

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

Document Type	Location Options
Text	<ul style="list-style-type: none">• Search text — Type a string in the Location text field. The Requirements Management Interface searches for the first occurrence of the text string within the document.• Line number — Type a line number in the Location text field. The Requirements Management Interface makes a link to the specified line.
HTML	<p>You can only link to a named anchor.</p> <p>For example, if you define the anchor</p> <pre> ...contents... </pre> <p>in your HTML requirements document, you can enter <code>valve_timing</code> in the Location text field or click the Document Index tab to select <code>valve_timing</code> from an automatically generated list of anchors in the document.</p>

Document Type	Location Options
Microsoft Word	<ul style="list-style-type: none"> • Search text — Type a string in the Location text field. The Requirements Management Interface searches for the first occurrence of the text string within the document. • Named item — Link to a bookmark within the document. The Requirements Management Interface automatically generates a document index based on its headings and bookmarks, or you can type the name in the Location text field. • Page/item number — Type a page number in the Location text field. The Requirements Management Interface makes a link to the top of the page.
Microsoft Excel	<ul style="list-style-type: none"> • Search text — Type a string in the Location text field. The Requirements Management Interface searches for the first occurrence of the text string within the document. • Named item — Link to a named item within the document (defined in Excel using Insert → Name). Type the name in the Location text field. • Sheet range — Type a cell number or a range of cells (such as C5:D7) in the Location text field. The Requirements Management Interface makes a link to the specified cell or cells.
PDF	<ul style="list-style-type: none"> • Named item — Link to a bookmark within the document. The Requirements Management Interface automatically generates a document index based on its headings and bookmarks, or you can type the bookmark name in the Location text field. • Page/item number — Type a page number in the Location text field. The Requirements Management Interface makes a link to the top of the page.
Web Browser URL	<p>You can link to a URL location only. Type the URL location string in the Document text field. When you follow the link, the document opens in the system Web browser.</p>

Adding Requirement Links to an Object

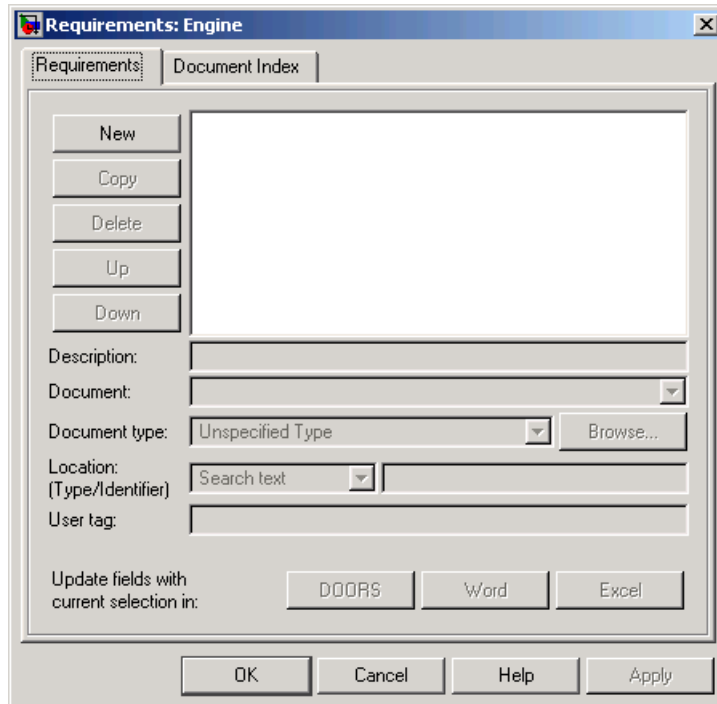
You use the Requirements dialog box to associate a requirements document with a requirements object. You can link a particular location in an existing Microsoft Word or HTML document to a block in a Simulink diagram or a Stateflow object in a Stateflow diagram. In the following procedure, you add three requirement links to a Simulink block in the demo model `sf_car`. In later topics, you modify both the links and the documents they point to.

- 1 Create and save the Microsoft Word document `requirements.doc` with the following format:

```
Primary Requirements
<filler text - 10 lines>
Secondary Requirements
<filler text - 10 lines>
Tertiary Requirements
<filler text - 10 lines>
```

- 2 Open the demo model `sf_car` by typing `sf_car` at the MATLAB prompt.
- 3 Right-click the Engine block and, from the resulting pop-up menu, select **Requirements > Edit/Add Links**.

The Requirements dialog box for the Engine block appears, as shown.

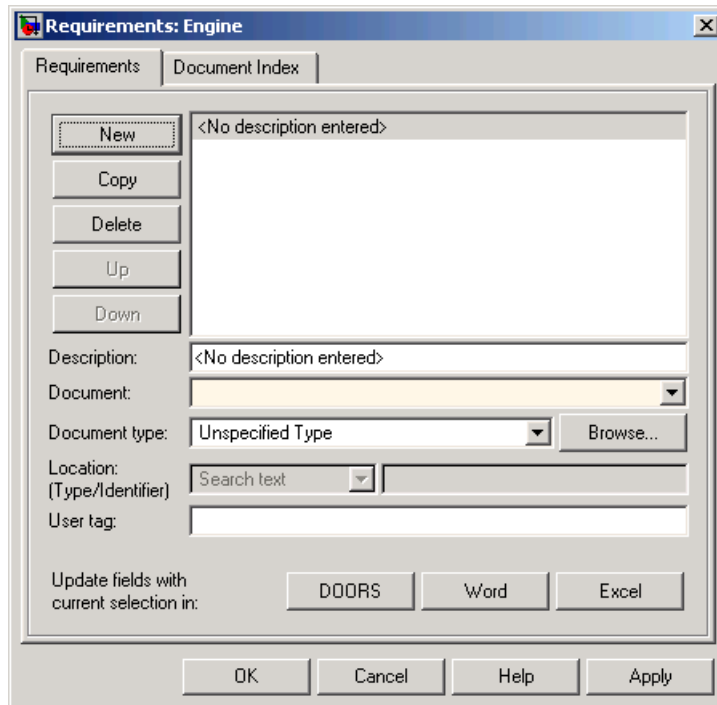


You can access the Requirements dialog box to add requirements in the following right-click contexts.

Model Element	Context
Simulink model	Simulink diagram — Empty diagram space
Simulink block	Simulink diagram Model Explorer Contents pane
Stateflow chart	Simulink diagram — Stateflow block Stateflow diagram — Empty diagram space
Stateflow object	Stateflow diagram Model Explorer Contents pane

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

Fields and tools of the Requirements dialog box are now enabled for a default unspecified requirement, as shown.

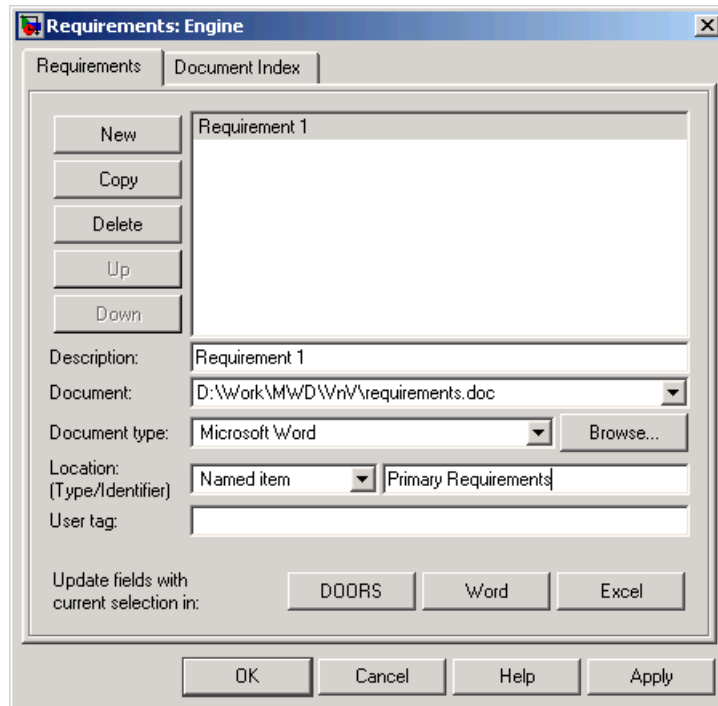


- 5 Click in the **Description** field and enter Requirement 1.
- 6 Click **Browse** next to the **Document Type** field, browse to the requirement file requirements.doc, and select **Open**.

Note that the **Document Type** field is now set to Microsoft Word. If you specify the document type in the **Document Type** field prior to browsing for the requirements document, only the files of the appropriate type are shown in the browser. If you set **Document Type** to Unspecified Type, the browser shows all files.

- 7 To define a particular location in the document, click the **Document Index** tab and select Primary Requirements from an automatically generated list of headings and bookmarks in the document. Alternately, you can just type

Primary Requirements in the **Location** text field on the **Requirements** tab.

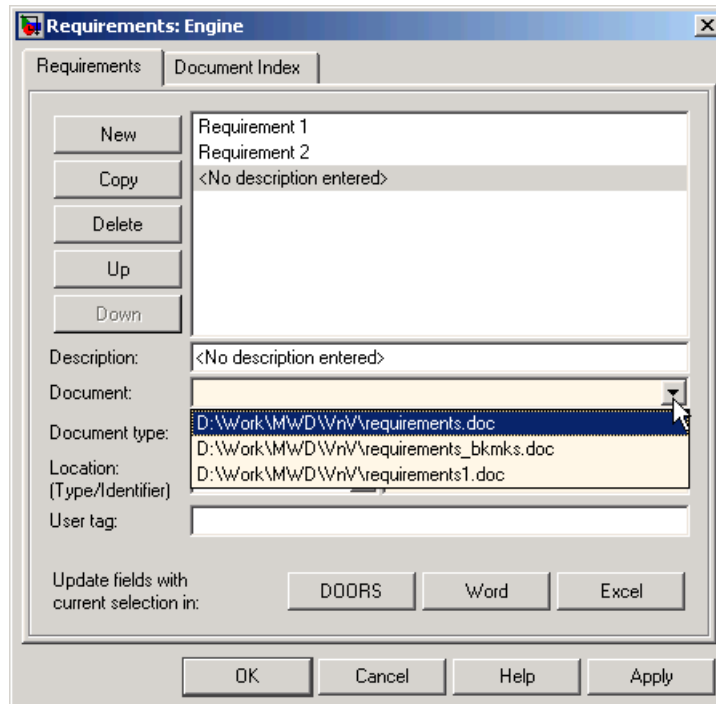


- 8 To associate comments with the requirement link, enter text in the **User tag** field. Use this field when you want to provide further details about the requirement, supplementing the **Description** field. Entering text in the **User tag** field is optional.
- 9 With Requirement 1 selected, click **Copy** to add a copy of Requirement 1 as a new requirement.
- 10 Modify the copy of Requirement 1 to be Requirement 2, pointing to the text “Secondary Requirements” in the document requirements.doc.

In addition to the **Copy** tool, you can edit existing requirements using the following tools.

Tool Button	Description
Delete	Deletes the requirement
Up	Moves the selected requirement up one line in the list of requirements
Down	Moves the selected requirement down one line in the list of requirements

As you add requirements, the Requirements dialog box makes it easier for you to enter a previous document name by remembering up to five of the most recent documents entered. The list of five is taken for all entries made across all models. For example, if you were to add a new requirement after entering two requirements for the Engine block, and click the drop-down arrow in the **Document** field for the new requirement, a selectable list of previous documents like the following example would appear.



- 11 Click **OK** to apply the requirement links you have added and close the Requirements dialog box.

Note When you add a requirement link to a block such as a subsystem, the requirement is not added to children of the block.

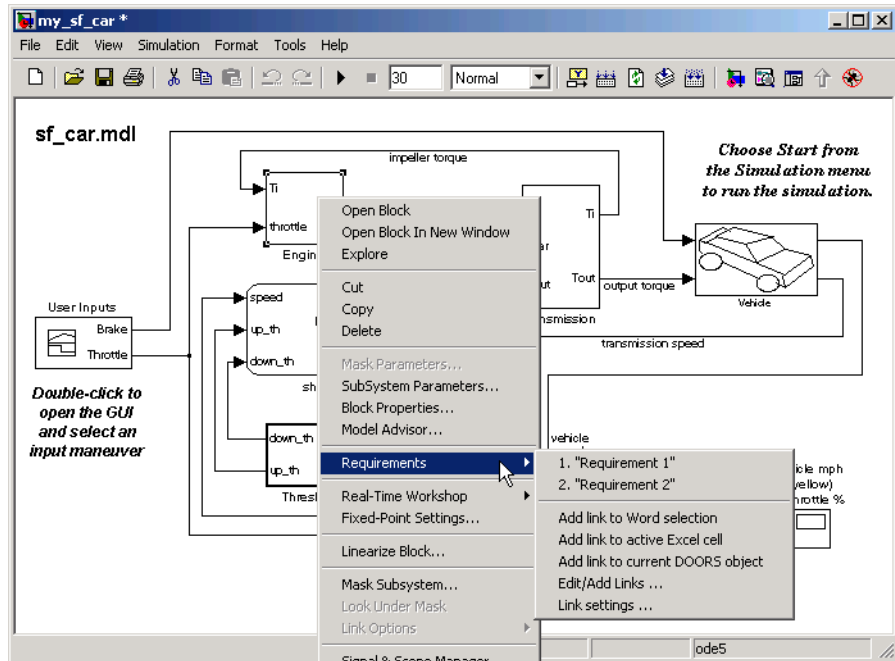
- 12 Save the model as `my_sf_car.mdl`.

Viewing Requirements Documents

You can access a requirements document through its associated model element. You added requirement links to the Engine block of the model `my_sf_car` in “Adding Requirement Links to an Object” on page 2-8. To access the document for the second requirement link, do the following:

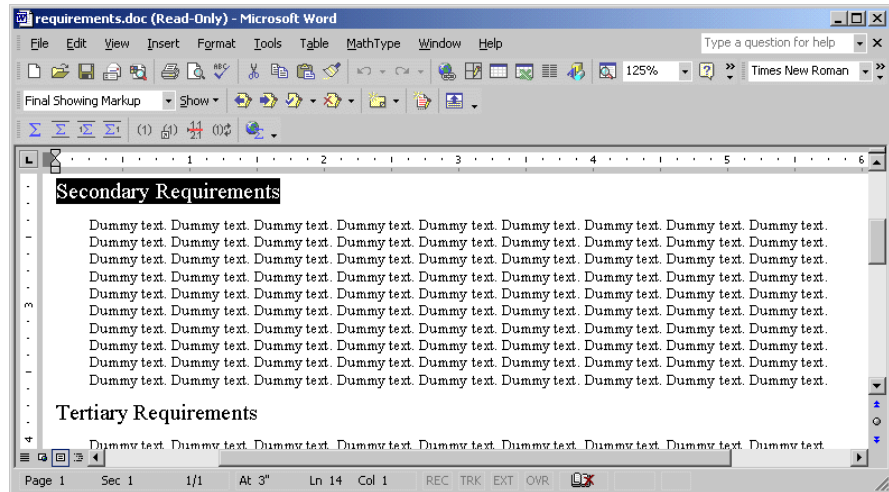
- 1 Open the model `my_sf_car` that you saved in the previous topic, “Adding Requirement Links to an Object” on page 2-8.
- 2 Right-click the Engine block and select **Requirements** from the context menu.

The requirements you added now appear as submenu selections, as shown.



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

The document requirements.doc opens in its editor, Microsoft Word, scrolled to the highlighted first occurrence of the text “Secondary Requirements,” as shown.



If a string location in the file is not specified, the document appears scrolled to the top of the file.

- 4 Try to access the other requirements and make sure that they open scrolled to the specified text location.

Resolving the Document Path

Browsing for a document to enter it in a requirements link enters the location of the document with a fully specified absolute path. You can also enter a relative path for the document location. A relative path can be a partial path or no path at all. In many cases it is preferable to use a relative path so that the document is not constrained to a single location in the file system. With a relative path the Requirements Management Interface resolves the exact location of the requirements document with the following procedure:

- 1 An attempt is made to resolve the path relative to the current MATLAB directory.
- 2 If there is no path specification and the document is not in the current directory, the MATLAB search path is used to locate the file.
- 3 If the document is not located relative to the current directory or the MATLAB search path, it is resolved relative to the model file directory.

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

Relative Path Specified Example

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

No Path Specified Example

Current MATLAB directory:	C:\work\scratch
Model file:	C:\work\models\controllers\pid.mdl
Requirements document:	pid.html
Documents searched for: (in order)	C:\work\reqs\scratch\pid.html <MATLAB path dir>\pid.html C:\work\models\controllers\pid.html

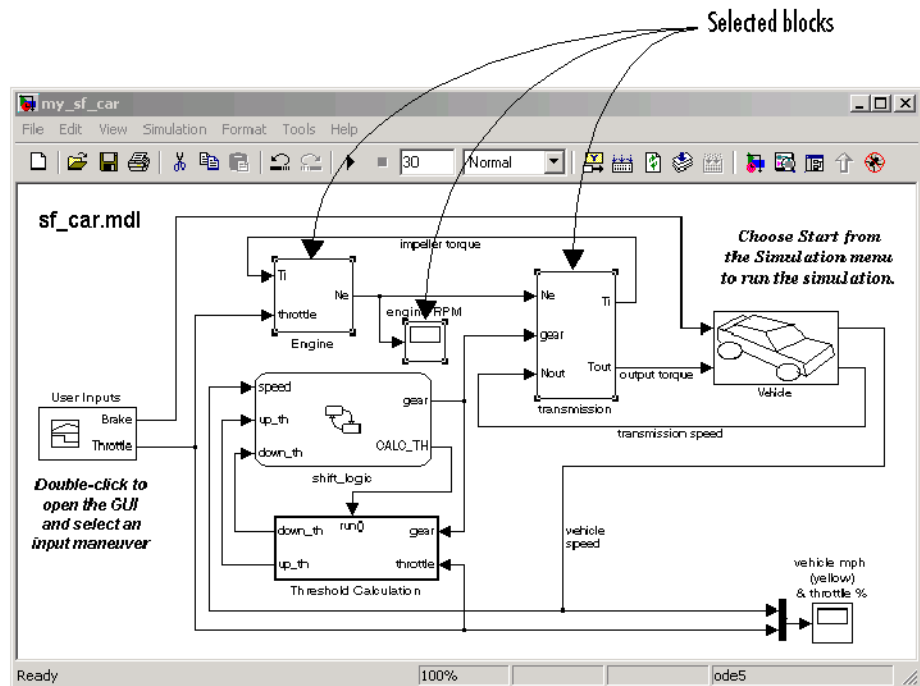
Absolute Path Specified Example

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

Adding Requirement Links to Multiple Objects Simultaneously

In “Adding Requirement Links to an Object” on page 2-8, you added requirement links to an individual block. You can also add or delete requirement links for a selection of multiple Simulink blocks or Stateflow objects as follows:

- 1 Open the demo model my_sf_car you saved in “Adding Requirement Links to an Object” on page 2-8 and select the Engine, engine RPM, and transmission blocks.

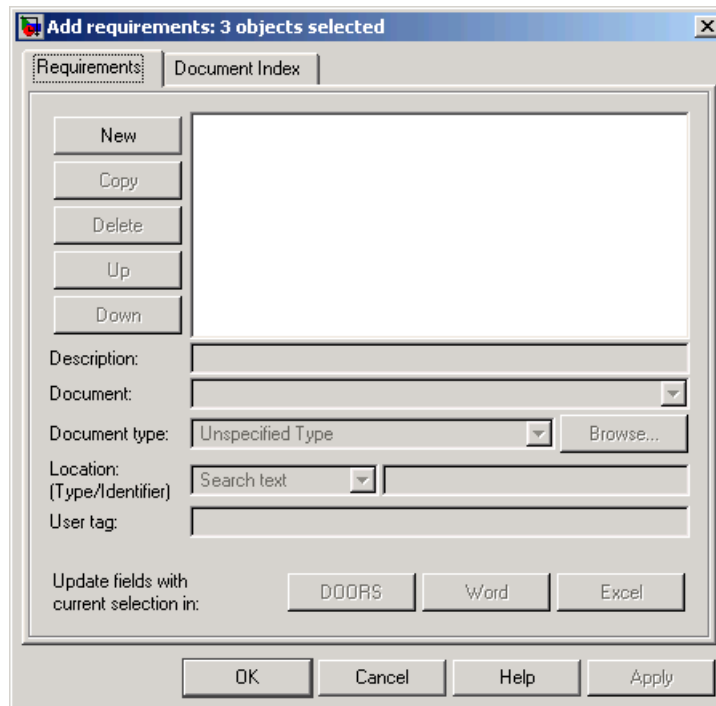


You can select multiple Simulink blocks or Stateflow objects in one of the following ways:

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

- 2 Right-click any of the selected blocks and, from the resulting context menu, select **Requirements > Add Links to All**.

The Add Requirements dialog box appears, as shown, for the three selected blocks.



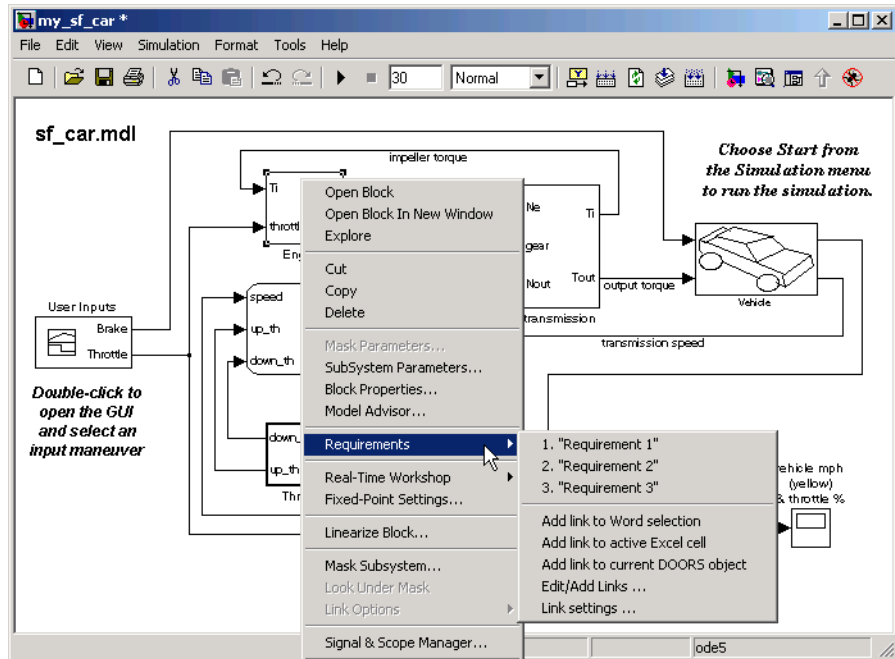
- 3 Add a new Requirement 3 for the set of blocks that points to the text “Tertiary Requirements” in the file requirement.doc.

Add the requirement as you would for a single block as described in “Adding Requirement Links to an Object” on page 2-8.

Note When you add a requirement link to a block such as a subsystem, the requirement is not added to children of the block.

- 4 Click outside the three objects to deselect them.
- 5 Right-click the Engine block and select **Requirements**.

The Engine block now has three requirements, as shown.



- 6 Right-click the engine RPM and transmission blocks to verify that they have only one requirement.
- 7 Save the model (my_sf_car).

Deleting All Requirement Links for Multiple Objects Simultaneously

In “Adding Requirement Links to Multiple Objects Simultaneously” on page 2-17, you added requirement links to each block in a multiple selection of blocks. To delete the existing requirements for a group of selected blocks, right-click any of a group of selected blocks and, from the resulting context menu, select **Requirements > Delete All**. Notice that this deletes all of the

requirement links for all of the selected blocks, whether they were added individually or as a group.

Selection-Based Linking

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

The following tasks show you how to work with selection-based linking:

- “Opening the External Application” on page 2-20
- “Specifying Your Linking Preferences” on page 2-21
- “Making Selection-Based Links” on page 2-22
- “Navigating from the Requirements Document to the Simulink Model” on page 2-25

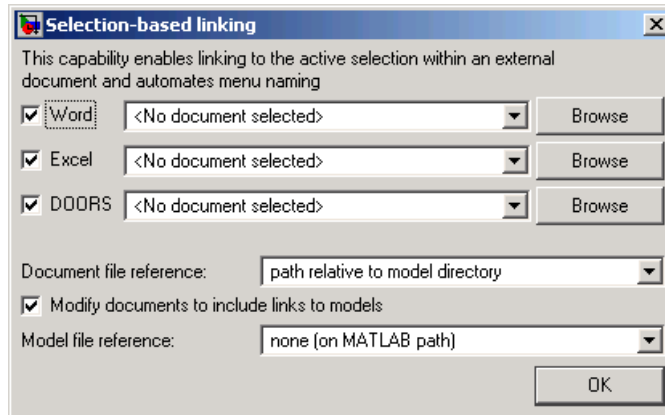
Opening the External Application

Before creating selection-based links, you need to establish communication between Simulink and the external application (Microsoft Word or Microsoft Excel) by opening the application from Simulink. The document you open cannot be read-only, or be already open in another application.

Note If you open a document some other way and try to create selection-based links to it, Simulink displays an error message.

Use the following procedure to open a document for selection-based linking:

- 1 Open the model `my_sf_car`.
- 2 In Simulink, from the **Tools** menu, select **Requirements > Link settings**. The Selection-based linking dialog box opens.



Another way to open the Selection-based linking dialog box is to right-click on a Simulink block and, from the resulting pop-up menu, select **Requirements > Link settings**.

- 3 From the drop-down list next to **Word**, select `requirements.doc`. Note that the **Browse** button to the right of it changes to **Open**. Click the **Open** button. The document `requirements.doc` opens in its editor, Microsoft Word.

Note If you click **Browse** with no document selected, the Requirement Management Interface lets you browse to the document of the appropriate type to open it.

You can now make links between the blocks in the `my_sf_car` model and portions of the `requirements.doc` document, as described in “Making Selection-Based Links” on page 2-22.

Specifying Your Linking Preferences

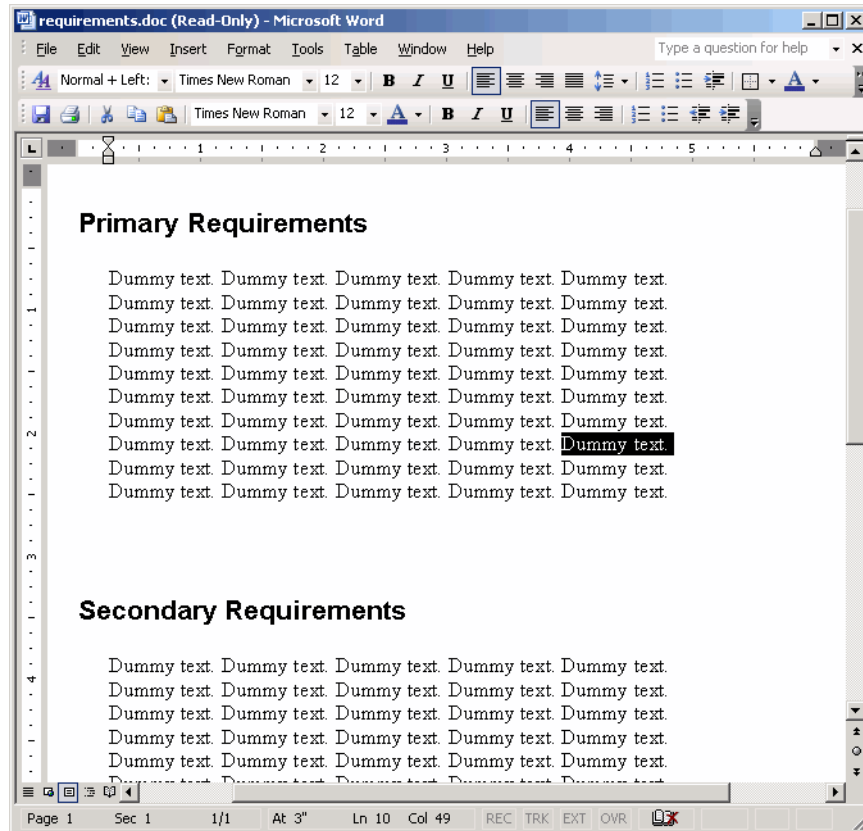
In selection-based linking, the process of making links is streamlined to minimize the number of menu clicks necessary to create a link. The Selection-based linking dialog box lets you specify your preferences:

- Two-way linking embeds an ActiveX control in your requirements document, which lets you navigate from the requirements document to the Simulink model. If you don't want to create two-way links, clear the **Modify documents to include links to models** check box.
- Use the **Document file reference** drop-down list to specify how to store the document location. For more information on the choices available, see “Resolving the Document Path” on page 2-15.
- If you create two-way links, use the **Model file reference** drop-down list to similarly specify how to resolve the model location when you navigate from a requirements document to the Simulink model.

Making Selection-Based Links

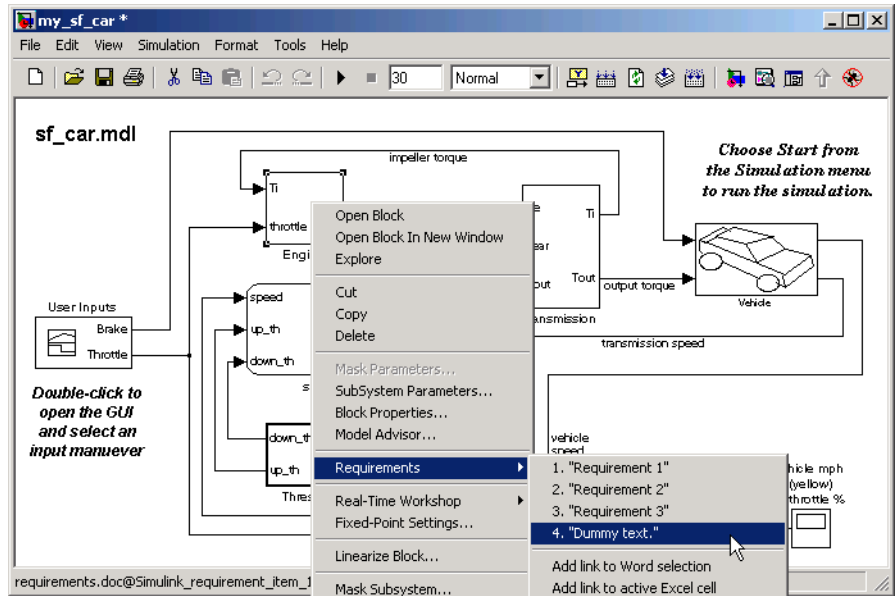
Use the following procedure to create selection-based requirement links:


- 1** Open the requirements.doc document, as described in “Opening the External Application” on page 2-20.
- 2** Select a portion of the text that documents the desired requirement. For example, select a “Dummy text.” string, as shown in the following illustration.

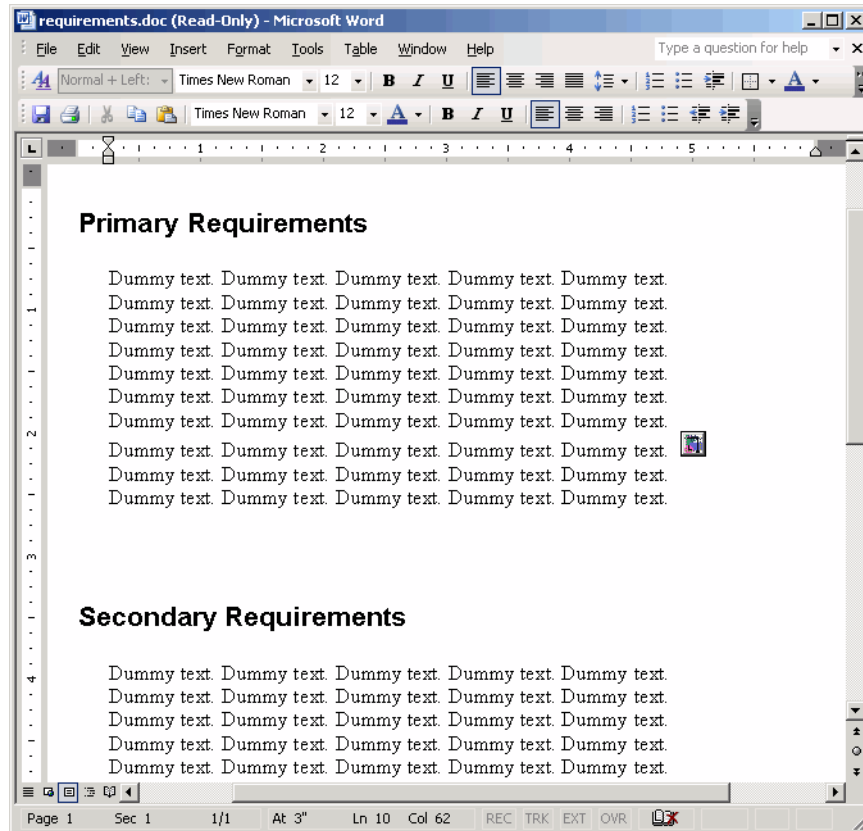


- 3 In the Simulink diagram, right-click the Engine block and, from the resulting pop-up menu, select **Requirements > Add link to Word selection**.

The Requirement Management Interface creates the link. If you right-click on the Engine block and select **Requirements**, the Engine block will now have four requirement links, as shown. You can use this link to navigate from the model to the requirements document, as described in “Viewing Requirements Documents” on page 2-13.



Because your linking preferences are set to two-way linking, an ActiveX control  is embedded in the requirements document next to the selected string, as shown in the next illustration. You can use it to navigate from the requirements document to the model, as described in “Navigating from the Requirements Document to the Simulink Model” on page 2-25.



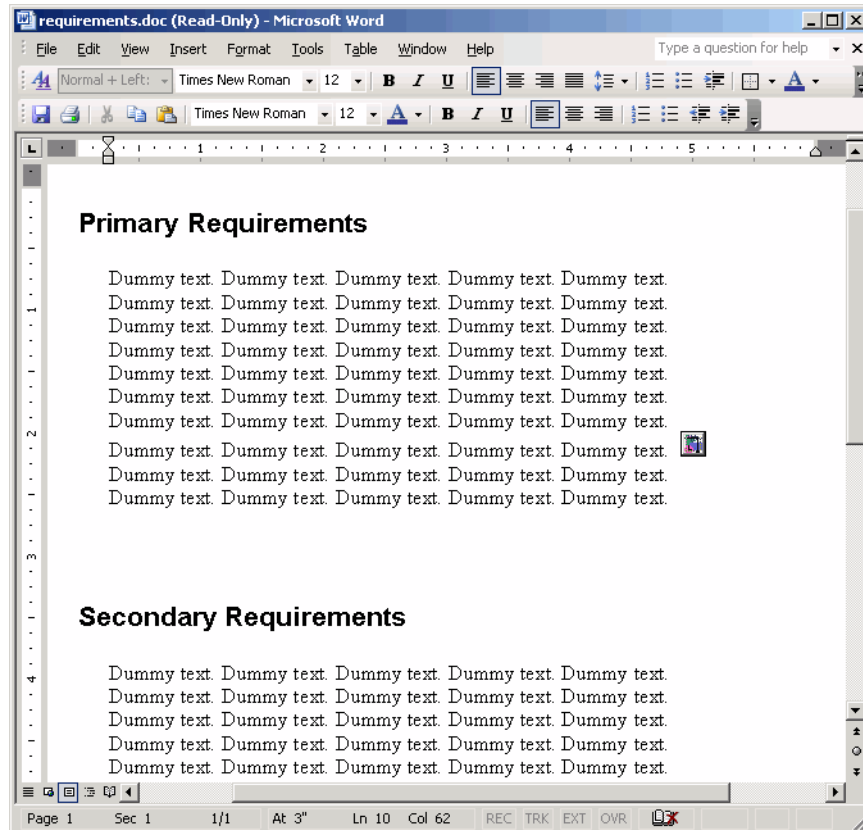
Note If you have the Requirements dialog box open, you can use the **Word** and **Excel** buttons next to the **Update fields with current selection in** label to create selection-based links to Microsoft Word or Microsoft Excel requirements documents, respectively.

Navigating from the Requirements Document to the Simulink Model

When you create two-way links, the Requirements Management Interface embeds an ActiveX control in your requirements document next to the

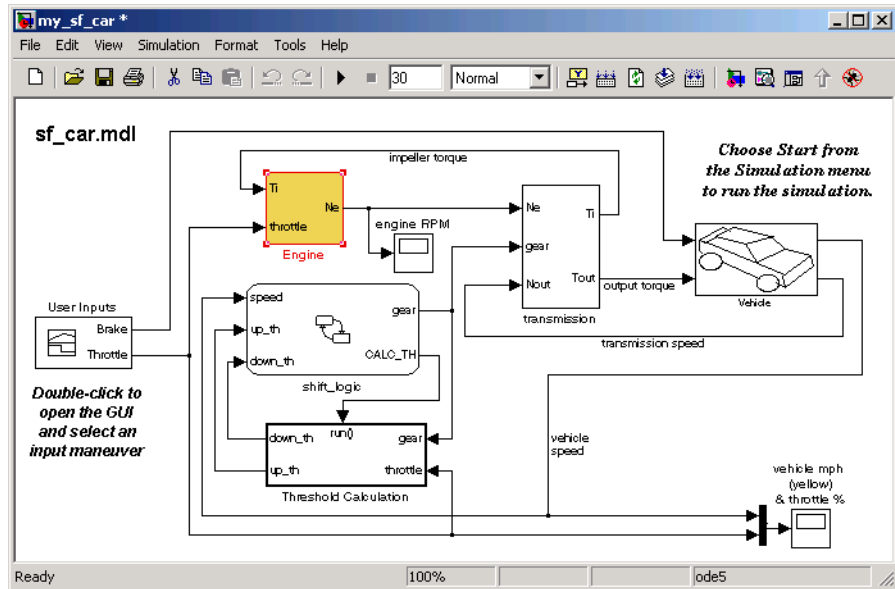
selected string or cell. This allows you to navigate from the requirements document to the Simulink model.

- 1 Open the requirements.doc document in Microsoft Word.



- 2 Click the ActiveX control  embedded in the requirements document.

The Requirements Management Interface opens the model my_sf_car and highlights the Engine block, as shown in the following illustration.



The requirement item that is linked with the ActiveX control is identified internally with a special unique string that is saved in the model. If you rename the item or change the path, the links in the external document will continue to work.

Linking to Custom Types of Requirements Documents

In this section...
“Why Create a Custom Link Type?” on page 2-28
“Custom Link Type Registration” on page 2-29
“Built-In Link Types” on page 2-29
“Link Properties” on page 2-30
“Link Type Properties” on page 2-30
“Creating a Custom Link Requirement Type” on page 2-32
“Navigating to Simulink from External Documents” on page 2-42

Why Create a Custom Link Type?

In addition to linking to built-in types of requirements documents as described previously, you can register your own custom types of requirements documents with the Requirements Management Interface, and then create requirement links to these types of documents.

Custom link types let you define how a document will be opened and navigated, and how you or another user can browse for a document and view an index of its contents. When you define a custom link type, you create MATLAB M-code functions that perform these operations. The Requirements Management Interface invokes the registered code when navigating to a document with the new link type, and when browsing for a document or displaying the index of a document within the Requirements dialog box.

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

Custom link types enable you to

- Link to requirement items in commercial requirement tracking software
- Link to in-house database systems

- Link to document types that are not internally supported in the tool

Custom Link Type Registration

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

```
rmi register mytargetfilename
```

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

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

```
rmi unregister mytargetfilename
```

When you create links using custom link types, the registration name is saved in the model. When you attempt to navigate a link, the Requirements Management Interface resolves the link type against the registered list and displays an error message if the link type is not found.

Built-In Link Types

Built-in link types use the same format and naming convention as custom types, although they use a different system for identification in the model file that supports backwards and forwards compatibility. You can use the built-in types as examples when developing your custom link types. The files for built-in link types are contained in the private directory of the requirements management tool (*matlabroot\toolbox\slvsv\reqmgt\private*):

```
linktype_rmi_doors.m  
linktype_rmi_excel.m  
linktype_rmi_html.m  
linktype_rmi_pdf.m
```

```
linktype_rmi_text.m  
linktype_rmi_word.m
```

Link Properties

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

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

Link Type Properties

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

Property	Description
Registration	The name of the M-file that creates the link type. This is stored in the Simulink model.
Label	A string to identify this link type. It is displayed on the Document Type drop-down list in the Requirements dialog box 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, then the standard method for resolving the path is used and the standard file selection dialog is invoked when the user clicks the Browse button in the Requirements dialog box.

Property	Description
Extensions	An array of file extensions. These are used as filter options for the Browse button in the Requirements dialog box. The extensions are also used to infer the link type based on the document name. If more than one link type is registered for the same file extension, the link type that was registered first will take 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 might refer a specific page number (#4), a named bookmark (@my_tag), or some text to search (?search_text). The valid location delimiters determine the possible entries on the Location drop-down list in the Requirements dialog box.
NavigateFcn	The MATLAB callback that is invoked when a user follows a link. The function is evaluated with two input arguments, the document field and the ID field of the link: <code>feval(LinkType.NavigateFcn, Link.document, Link.id)</code>
ContentsFcn	The MATLAB callback that is invoked when a user clicks Document Index tab in the Requirements dialog box. This function is evaluated with a single input argument containing the full path of the resolved function, or the entry from the Document field if the link type is not a file. The function should return three outputs: <ul style="list-style-type: none"> • Labels • Depths • Locations
BrowseFcn	The MATLAB callback that is invoked when a user presses the Browse button in the Requirements dialog box. This function is unnecessary when the link type is a file. The function should not take any input arguments and should return a single output argument identifying the document that the user selected.

Creating a Custom Link Requirement Type

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

You provide the ability to see a document index, containing a listing of all the requirement items. When navigating from the Simulink model to the requirements document, the document opens in the MATLAB editor and pans the display to the line containing the desired requirement item.

Use the following procedure to create a custom link requirement type:

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

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

```
% >> which demo_req_1.abc

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

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

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

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

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

% Uncomment the functions that are implemented below
linkType.NavigateFcn = @NavigateFcn;
linkType.ContentsFcn = @ContentsFcn;

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

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

function openFileToItem(fileName, itemName)
    reqStr = ['Requirement:: "' itemName '"'];
    lineNumber = 0;
    fid = fopen(fileName);
    i = 1;
    while lineNumber == 0
        lineStr = fgetl(fid);
        if ~isempty(strfind(lineStr, reqStr))
            lineNumber = i;
        end;
        if ~ischar(lineStr), break, end;
        i = i + 1;
    end;
    fclose(fid);
    openFileToLine(fileName, lineNumber);
end

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

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

    % Combine and sort the list
    items = [fList1{:}];
```

```

items = sort(items);
items = strcat('@',items);

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

depths = [];

```

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

```
rmi register rmicustabcinterface
```

This will cause the ABC file type to be added to the drop-down list of document types in the Requirements dialog box.

- 3** Create a text file with the .abc extension, containing several requirement items marked by the Requirement:: string, as described above. For your convenience, an example of such a file is included in the installation. It is named demo_req_1.abc, and is located in *matlabroot\toolbox\slvnx\rmidemos*. The contents of this file is displayed below.

```
Requirement:: "Altitude Climb Control"
```

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

```
Units:
Actual Altitude - feet
Desired Altitude - feet
```

```
Description:
```

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

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

<END "Altitude Climb Control">

Requirement:: "Altitude Hold"

Altitude hold mode is entered whenever:
 $| \text{Actual Altitude} - \text{Desired Altitude} | < 30 * \text{Sample Period} * (\text{Pilot Climb Rate} / 60)$

Units:

Actual Altitude - feet

Desired Altitude - feet

Sample Period - seconds

Pilot Climb Rate - feet/minute

Description:

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

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

<END "Altitude Hold">

Requirement:: "Autopilot Disable"

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

Description:

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

<END "Autopilot Disable">

Requirement:: "Glide Slope Armed"

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

Units:

Glide Slope Enable - Logical

Glide Slope Signal - Logical

Description:

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

<END "Glide Slope Armed">

Requirement:: "Glide Slope Coupled"

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

Distance < 10000

Units:

Glide Slope Angle Error - Logical

Distance - feet

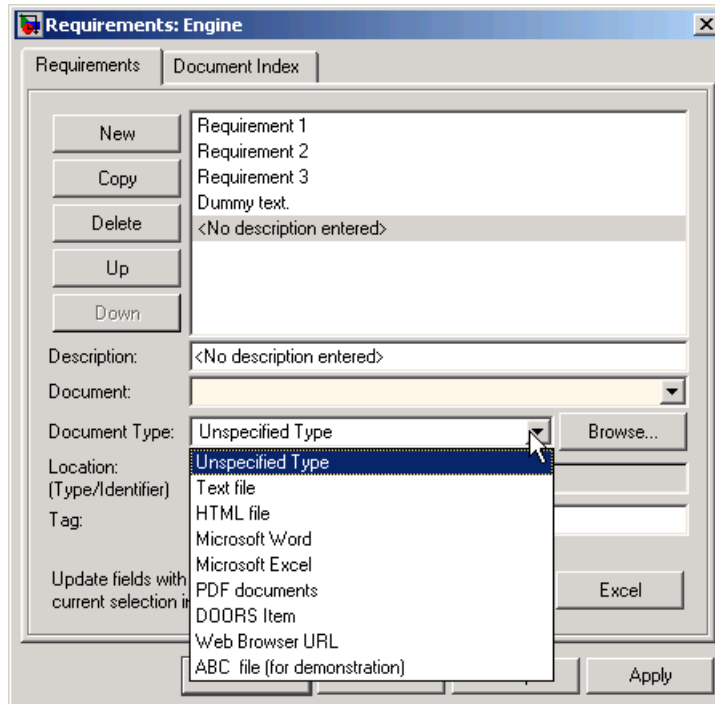
Description:

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

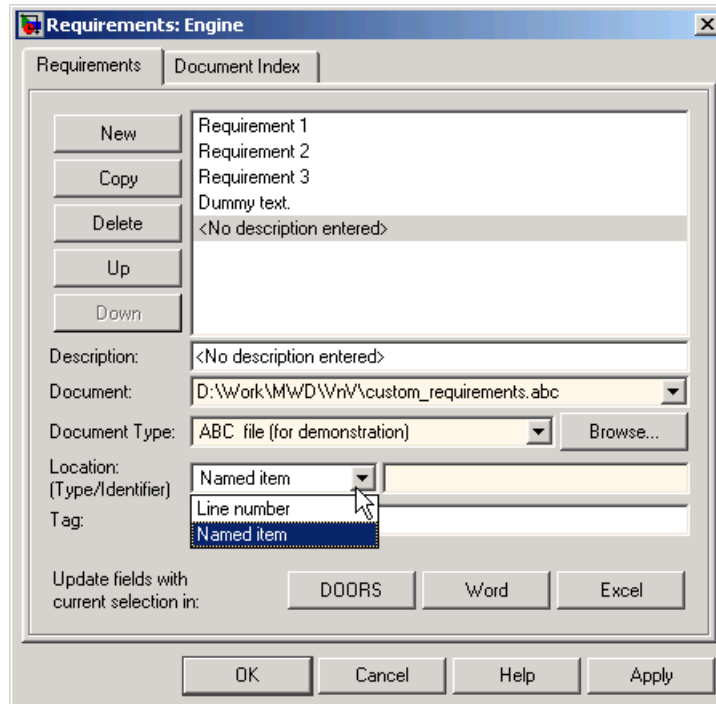
The user-selectable climb rate is always a positive number if the current altitude is above the target altitude the actual target climb rate is the negative of the user setting.

<END "Glide Slope Coupled">

- 4 Open the model my_sf_car.
- 5 Right-click the Engine block and, from the resulting pop-up menu, select **Requirements > Edit/Add Links**.
The Requirements dialog box appears.
- 6 Click **New** to add a new default requirement. Note that ABC file type is now available in the **Document Type** drop-down list, as shown.



- 7 Set **Document Type** to ABC file (for demonstration) and browse to the demo_req_1.abc file, or to your own .abc requirements file that you created in Step 3. Note that the browser shows only the files with the .abc extension.
- 8 Define a particular location in the document. In this example, you can either use a line number or a requirement name as the item identifier, so the location delimiters in the rmicustabcinterface function are specified as '>@'. As a result of this parameter, the **Location** drop-down menu contains these two items whenever the document type is set to ABC file, as shown.



Creating a Document Index

The example file format clearly defines requirement items that are easily listed. To generate a document index, set the ContentsFcn to a valid function. The MATLAB code uses file I/O to read the contents into the MATLAB variable. The Requirements Management Interface uses the regular expression utility in MATLAB to extract a list of requirement items that it returns.

The following code generates an index for the ABC files.

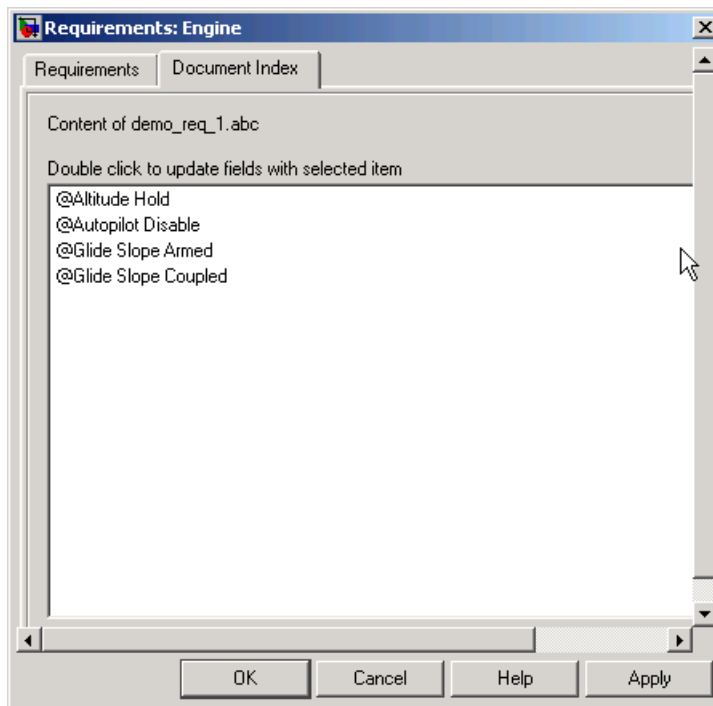
```
function [labels, depths, locations] = ContentsFcn(filePath)
    % Read the entire M-file into a variable
    fid = fopen(filePath,'r');
    contents = char(fread(fid)');
    fclose(fid);
```

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

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

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

For example, for the `demo_req_1.abc` file discussed earlier in “Creating a Custom Link Requirement Type” on page 2-32, this function generates the document index as shown in the following illustration.



Navigating to Simulink from External Documents

The Requirements Management Interface includes several functions that simplify creating navigation interfaces in external documents. The external application that displays your document must support an application program interface (API) for communicating with MATLAB.

Providing Unique Object Identifiers

Whenever you create a requirement link for an object in Simulink or Stateflow, a globally unique identifier is created for that object. This identifier is used to identify the object and will not change if the object is renamed or moved or when requirement links are added or deleted. Although the unique identifier is only used to resolve an object within a model, the identifier is globally unique and should not collide with identifiers in other models unless the two models derive from the same original model. Unique object identifiers have formats like GIDa_cd14afcd_7640_4ff8_9ca6_14904bdf2f0f.

Using the rmiobjnavigate Utility

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

The first time you navigate to an item in a particular model, there may be a slight delay while the internal navigation table is constructed. Subsequent navigation should have minimal delay.

Determining the Navigation Command

Once you have created a requirement link for a Simulink or Stateflow object, you can find the appropriate navigation command string using the `rmi` function in MATLAB. The return value of the `navCmd` method is a string that navigates to the correct object when evaluated in MATLAB:

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

You need to send this exact string to MATLAB for evaluation as part of navigating from the external application to Simulink.

Using the ActiveX Navigation Control

A special ActiveX control is included with Simulink Verification and Validation and is used to enable linking to Simulink models from Microsoft Word and Microsoft Excel documents. You can use this same control from any other application that supports ActiveX within its documents.

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

Typical Code Sequence for Establishing Two-Way Links

When you create an interface to an external tool, the procedure for establishing links can often be automated so that no dialog fields need to be manually updated. This type of automation is part of the selection-based linking that is implemented for certain built-in types, such as Microsoft Word and Microsoft Excel.

In generic terms, use the following process:

- 1** Select an object in Simulink or Stateflow and an item in the external document.
- 2** Invoke the link creation action either from a menu or command in Simulink, 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 MATLAB and create a requirement link on the selected object using `rmi('createempty')` and `rmi('cat')`.
- 4** Determine the MATLAB navigation command string that must be embedded in the external tool using the `navCmd` method:

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

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

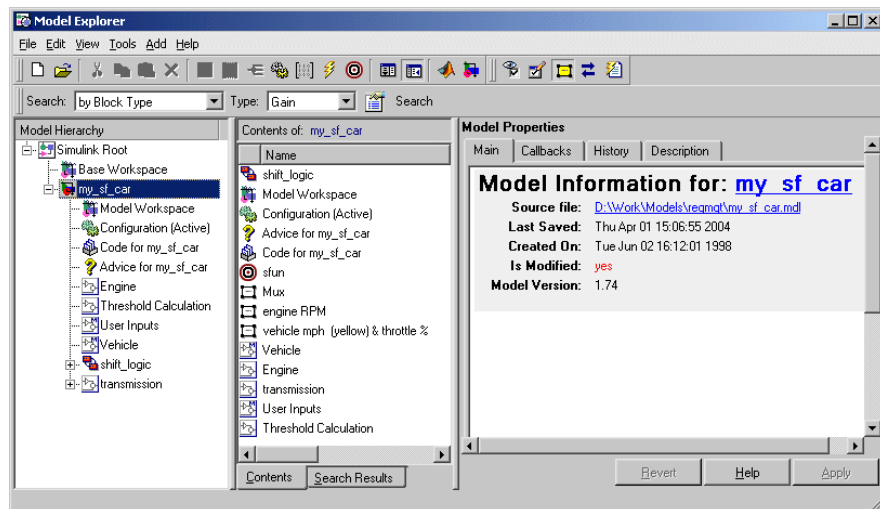
You can use the code for selection-based linking to Word, Excel, and DOORS as an example of this type of automation. The files are contained in *matlabroot\toolbox\slvnv\reqmgt\private*:


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

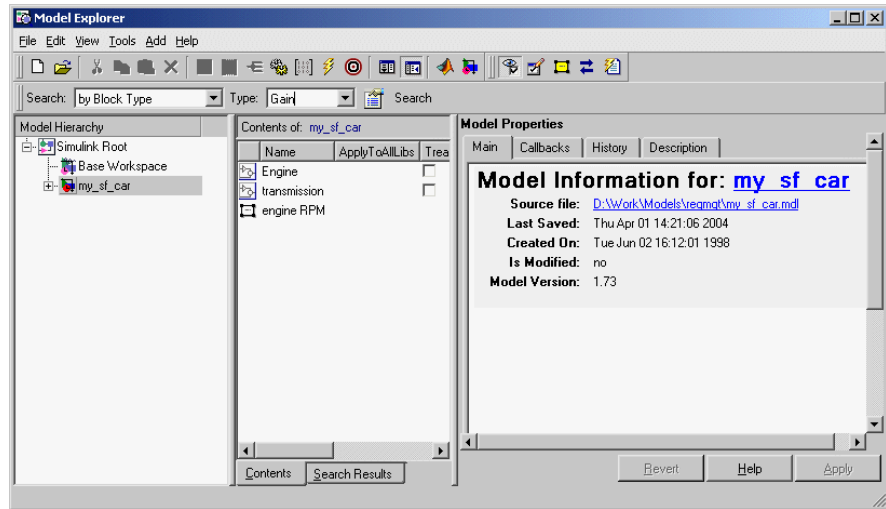
Viewing Objects with Requirement Links

After you have added requirements to blocks in a model, you can change the view in the Model Explorer window to show only objects that have requirements associated with them. In “Adding Requirement Links to an Object” on page 2-8 and “Adding Requirement Links to Multiple Objects Simultaneously” on page 2-17, you add requirements to the Engine, Engine RPM, and transmission blocks in the model you save as `my_sf_car`. Use the following procedure to highlight these objects in the Model Explorer and Simulink:

- 1 Open the model `my_sf_car`.
- 2 In Simulink, from the **View** menu, select **Model Explorer**.
- 3 The model and its elements appear in the Model Explorer window as shown.




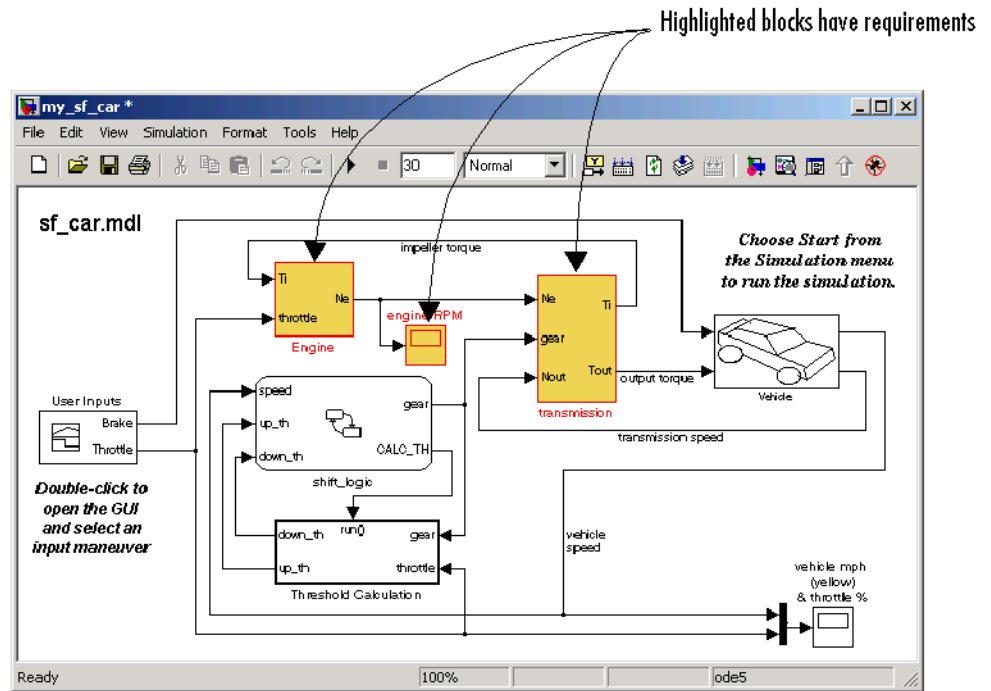
- 4 In the **Model Explorer** toolbar, select the Display Objects with Linked Requirements tool .



- 5 To see all objects, click the Display Objects with Linked Requirements tool again to deselect it.

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

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



- 2 To drop the highlighting, click the Highlight Items with Requirements in Model tool again to deselect it.

Generating a Requirements Report

After you have added requirements to a model, you can generate a report on all the requirements associated with the model and its blocks. In “Adding Requirement Links to an Object” on page 2-8 and “Adding Requirement Links to Multiple Objects Simultaneously” on page 2-17, you add requirements to the Engine, Engine RPM, and transmission blocks in the model you save as `my_sf_car`. Use the following procedure to generate a requirements report:

- 1 Open the model `my_sf_car`.
- 2 In Simulink, from the **Tools** menu, select **Requirements > Generate Report**.

The Requirements Management Interface searches through all the blocks and subsystems in the model for associated requirements, generates a complete report in HTML format with the default name `rmi.html`, and displays it in your system Web browser, as shown.

Model Requirements - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address <D:\Work\MWD\vn\Ymi.html>

Links [Customize Links](#) [FAQ Page](#) [Free Hotmail](#) [Help Page](#)

Model Requirements

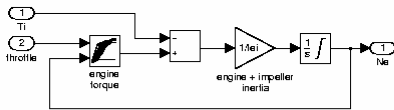
jsmith

17-Feb-2005 15:14:53

Table of Contents

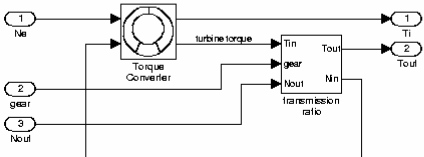
[1. System - Engine](#)
[2. System - transmission](#)
[3. Block - engine RPM](#)

Chapter 1. System - Engine



Description	Document	ID
Requirement 1	requirements.doc	Primary Requirements
Requirement 2	requirements.doc	Secondary Requirements
Requirement 3	requirements.doc	Tertiary Requirements

Chapter 2. System - transmission



- To save the report with a meaningful name, in the Web browser main menu click **File > Save As**, type the desired filename, and click **Save**.

Displaying the System Requirements in a Diagram

In this section...
“About the System Requirements Block” on page 2-50
“Adding the System Requirements Block” on page 2-50
“Renaming the System Requirements Block” on page 2-53
“Changing Fonts for the System Requirements Block” on page 2-54

About the System Requirements Block

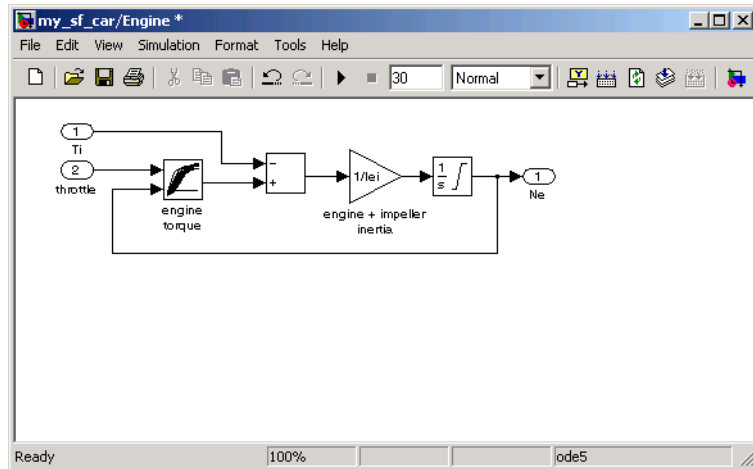
You can list all the requirements for a model or a subsystem directly on the Simulink diagram. You do this by adding the System Requirements block from the Simulink Verification and Validation library to the diagram. You can place this block anywhere in a diagram. It is not connected to other Simulink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

Once you place the System Requirements block in a Simulink diagram, it automatically lists the requirements associated with the model or subsystem depicted in the current diagram. It does not list requirements associated with individual blocks in the diagram.

Adding the System Requirements Block

In “Adding Requirement Links to an Object” on page 2-8, you added requirement links to the Engine block of the model `my_sf_car`. You can list these requirements in the block diagram of the Engine subsystem as follows:

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



- 3 Click the Library Browser tool .

The Simulink Library browser opens.

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

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

- 5 Select the System Requirements block in the right pane of the Simulink Library browser and drag it to an empty space in the Engine diagram.

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

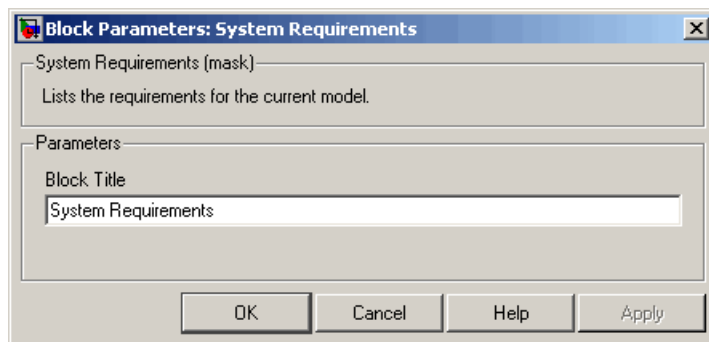
Once the System Requirements block is placed in a diagram, it automatically updates the listing as you add, modify, or delete requirements for the model or subsystem.

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

Renaming the System Requirements Block

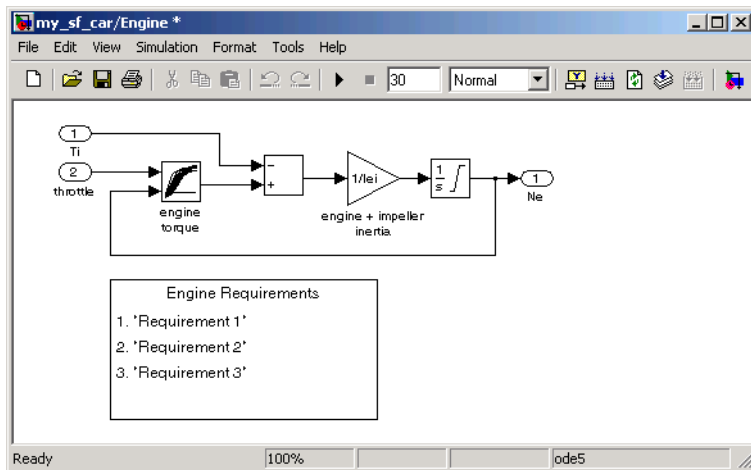
By default, the list of the system requirements in a diagram appears under a heading System Requirements. You can change the heading by renaming the System Requirements block in the diagram, as follows:

- 1 Right-click on the System Requirements block in the `my_sf_car/Engine` diagram.
- 2 From the resulting pop-up menu, select **Mask Parameters**. The Block Parameters dialog box opens, as shown.



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

The requirements heading in the diagram is updated as shown.

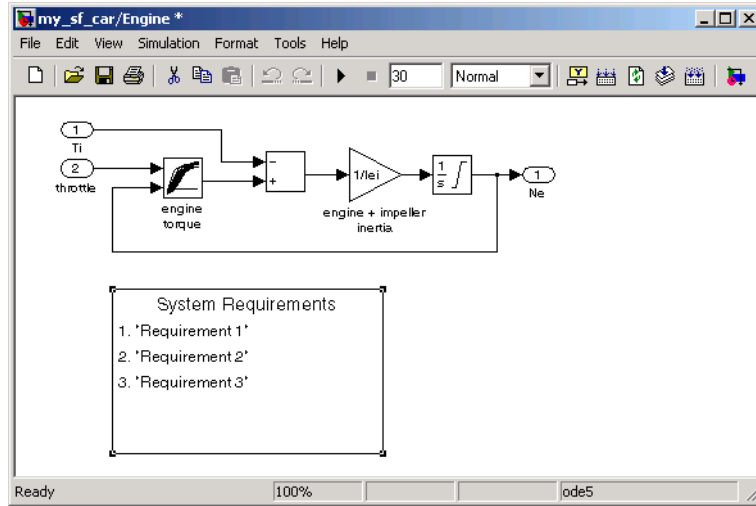


Changing Fonts for the System Requirements Block

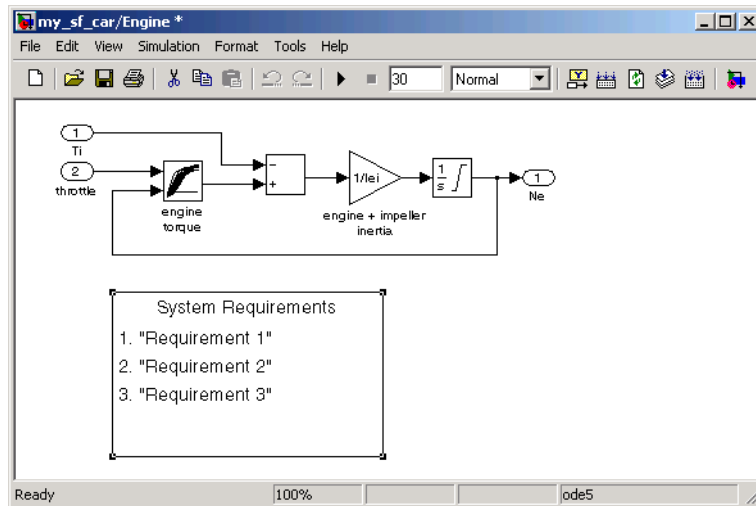
The System Requirements block is implemented using a set of empty subsystems. Because of this, occasionally the appearance is not refreshed correctly, for example, when you make a change to the font style or size. You can easily fix this problem by double-clicking the top label for the block, which causes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

Use the following procedure to change the font used in the block.

- 1 Right-click on the System Requirements block in the `my_sf_car/Engine` diagram.
- 2 From the resulting pop-up menu, select **Format > Font**. The Set Font dialog box opens.
- 3 Under **Size**, select 14, then click **OK**. The block display partially refreshes, as shown.



4 To refresh the entire block display, double-click the top label, System Requirements. The block diagram now looks as shown below.



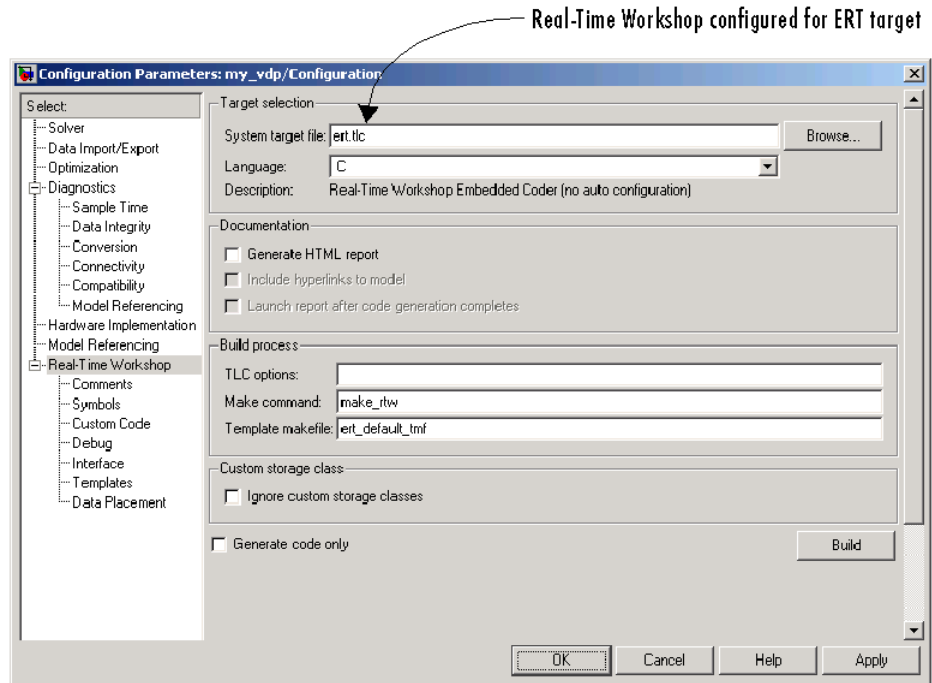
Including Requirements with Generated Code

Once you finish simulating your model and verifying its performance against the requirements, you might want to use it to generate code for an embedded real-time application. Simulink includes the requirements that you assign to Simulink blocks in generated code for Embedded Real-Time (ERT) targets of Real-Time Workshop® Embedded Coder.

To specify that requirements be included in the generated code of an ERT target, do the following:

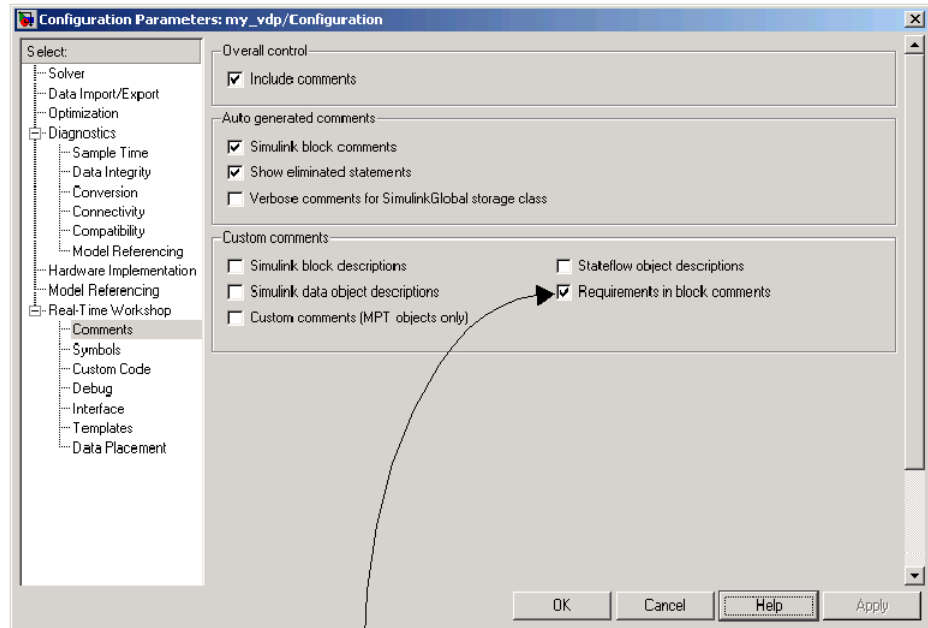
- 1 Load the model.
- 2 In the Simulink window, from the **Simulation** menu item, select **Configuration Parameters**.
- 3 In the **Select** pane of the Configuration Parameters dialog box, select the Real-Time Workshop node.

The currently configured system target must be an ERT target, as shown.



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

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



Include requirements descriptions
in generated code comments

Requirement descriptions are included with generated code in the following locations.

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

Managing Model Requirements with DOORS

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

What Is the Requirements Management Interface for DOORS? (p. 3-2)

Shows how the Requirements Management Interface associates objects with DOORS requirements

Configuring the Requirements Management Interface for DOORS (p. 3-3)

Specifies additional files from MATLAB that you need to copy into your installation for DOORS

Starting the Requirements Management Interface for DOORS (p. 3-6)

Describes how to start the Requirements Management Interface for DOORS and create projects

Linking Objects to DOORS Requirements (p. 3-8)

Describes how to add two-way links between Simulink or Stateflow objects and DOORS requirements

Synchronizing DOORS with the Simulink Model (p. 3-13)

Describes how to synchronize the model and link synchronized objects with DOORS requirements

Navigating Between Model Objects and DOORS (p. 3-25)

Describes how to move between the model and the DOORS synchronized module, and how to view the nodes in an object that have associated requirements

What Is the Requirements Management Interface for DOORS?

DOORS is a requirements management application that captures, tracks, and manages user requirements. The Requirements Management Interface for DOORS is a special interface between your Simulink model and DOORS.

Configuring the Requirements Management Interface for DOORS

In this section...

“Before You Begin” on page 3-3

“Installing DOORS Before RMI” on page 3-3

“Installing DOORS After RMI” on page 3-3

“Upgrading DOORS” on page 3-4

“Manual Installation for DOORS” on page 3-4

Before You Begin

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

Installing DOORS Before RMI

If DOORS is installed before you install the RMI and run the setup script, as described in “Configuring the Requirements Management Interface” on page 2-4, no additional installation for DOORS is necessary. The setup script automatically copies all the necessary files to the correct location.

Installing DOORS After RMI

If you install DOORS after you install the RMI, run the setup script again, as described in “Configuring the Requirements Management Interface” on page 2-4.

Upgrading DOORS

If you upgrade the DOORS installation after installing the RMI, run the setup script again, as described in “Configuring the Requirements Management Interface” on page 2-4.

If you upgrade from DOORS 7.1 to DOORS 8.0, follow these additional steps:

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

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

It should now look like this:

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

- 4 Save and close the file.
- 5 Start DOORS and MATLAB.
- 6 Run the setup script.

Manual Installation for DOORS

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

- 1 If DOORS is running, close DOORS.
- 2 Copy the following files from `matlabroot\toolbox\slvnx\reqmgt` to the `<doors>\lib\dxl\addins` directory:

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


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

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

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

Replace any existing versions of these files.

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

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

Starting the Requirements Management Interface for DOORS

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

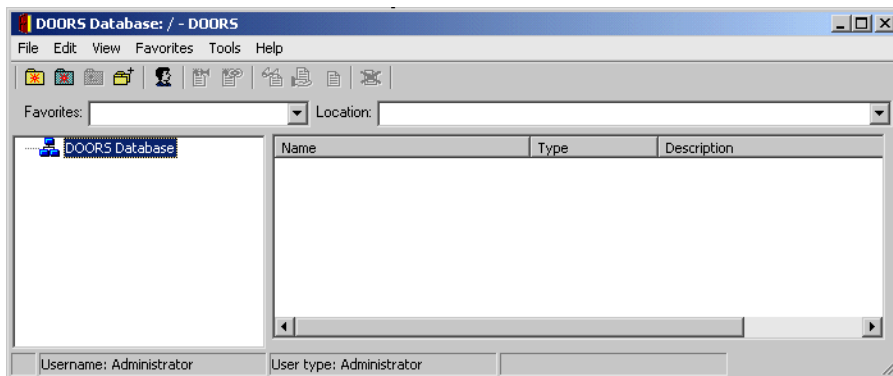
- 1 Start MATLAB in DOS or UNIX with the following command:

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

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

- 2 Start DOORS.

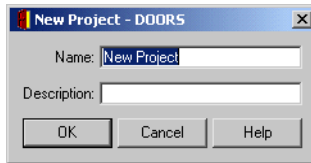
The DOORS Database window appears, as shown.



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

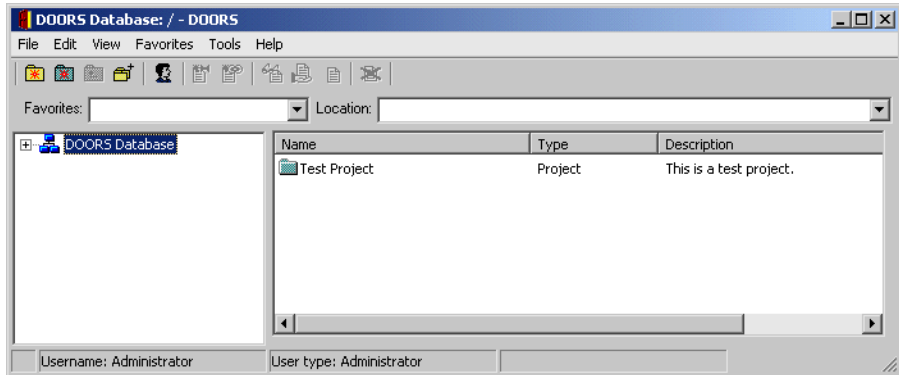
- 1 Right-click the DOORS Database node in the left pane and, from the resulting menu, select **New > Project**.

The New Project dialog box appears, as shown.

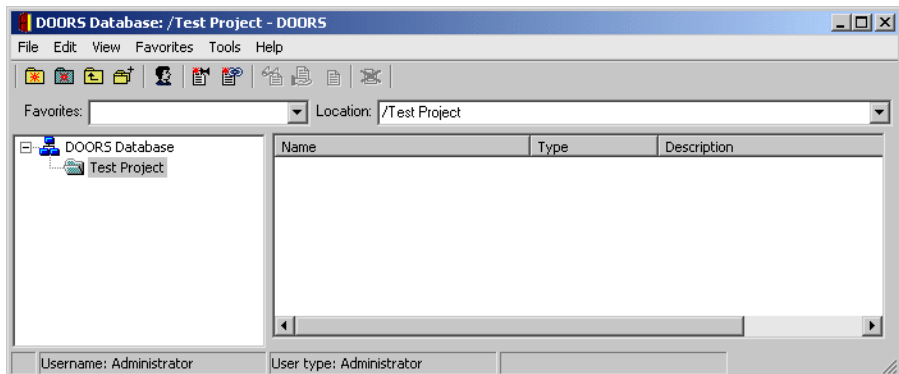


- 2 Enter the name Test Project and the description This is a test project. and click **OK**.

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



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



Linking Objects to DOORS Requirements

In this section...
“About Linkages Between a Simulink Model and DOORS” on page 3-8
“Creating a DOORS Requirement Object” on page 3-8
“Linking a Simulink or Stateflow Object to a DOORS Requirement” on page 3-10

About Linkages Between a Simulink Model and DOORS

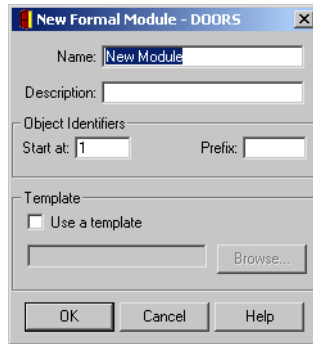
The Requirements Management Interface for DOORS lets you add links to DOORS requirements directly to the Simulink or Stateflow objects, without having to synchronize the model and navigate to the surrogate DOORS module. This linking mechanism is similar to selection-based linking to Microsoft Word and Microsoft Excel documents, described in “Selection-Based Linking” on page 2-20. It also creates two-way links by creating a special navigation object in DOORS, which allows you to navigate from the DOORS requirement to the associated object in the Simulink or Stateflow diagram.

Creating a DOORS Requirement Object

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

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

The New Formal Module dialog box appears, as shown.



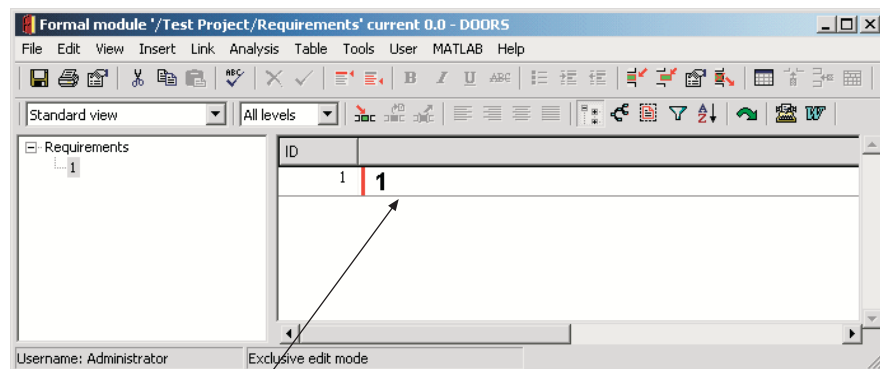
- 2 In the **Name** text field, enter the name Requirements and click **OK**.

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

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

- 4 In the formal module window Requirements, from the **Insert** menu select **Object**.

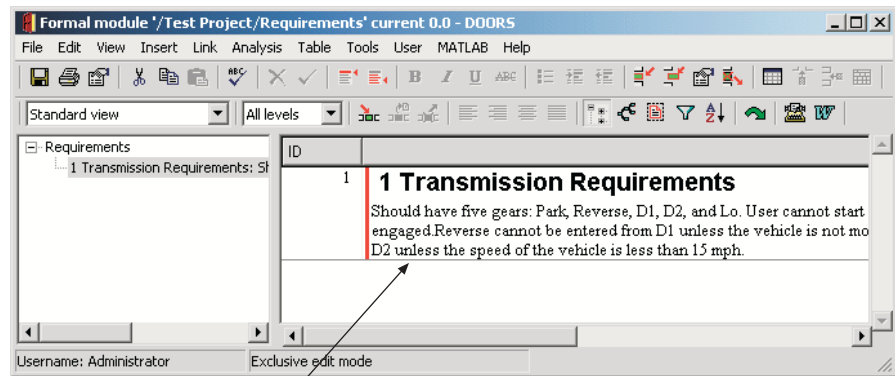
A new object appears in the formal module, as shown.



New object in Requirements module

- 5 Right-click the object in the right pane and, from the resulting context menu, select **Properties**.
- 6 In the resulting Object properties dialog box, enter the **Heading** Transmission Requirements, some text for **Object Text**, and select **OK**.

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

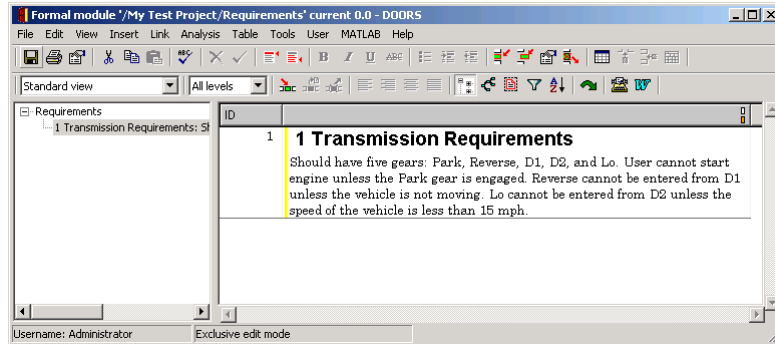


Specified object in Requirements module

Linking a Simulink or Stateflow Object to a DOORS Requirement

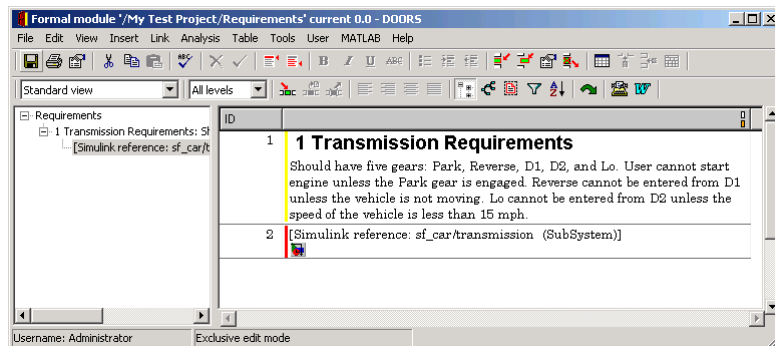
In “Creating a DOORS Requirement Object” on page 3-8, you created a Transmission Requirements object in the Requirements formal module in DOORS. Now use the following procedure to link the transmission block in the sf_car model to this DOORS requirement.

- 1 In the MATLAB Command Window, type sf_car at the MATLAB prompt to open the demo model sf_car.mdl.
- 2 In the formal module window Requirements, select the Transmission Requirement node in the left pane, as shown.



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

The Requirements Management Interface for DOORS adds the link to the DOORS requirement object, as shown.



- 4 Save the DOORS module.
- 5 Save the Simulink model as sf_car_doors.mdl.

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

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

Synchronizing DOORS with the Simulink Model

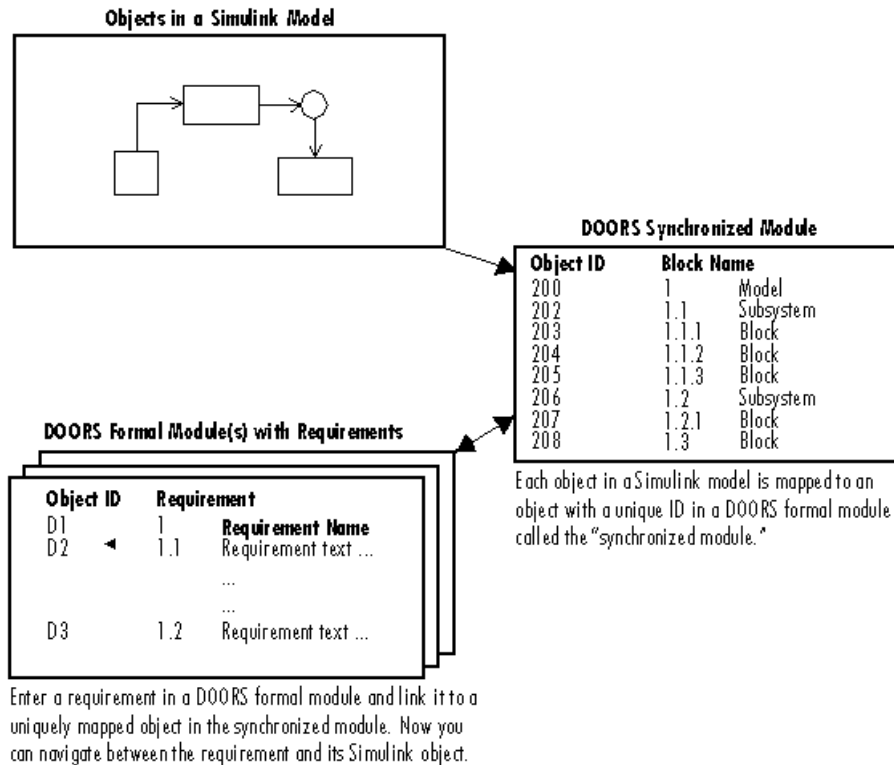
In this section...
“About Module Synchronization” on page 3-13
“Synchronizing a Model with DOORS” on page 3-14
“Customizing the Level of Synchronization Detail” on page 3-16
“Customizing the DOORS Synchronization Settings” on page 3-21
“Linking Requirements to the DOORS Synchronized Module” on page 3-23

About Module Synchronization

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

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

The following diagram illustrates the synchronization process.



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

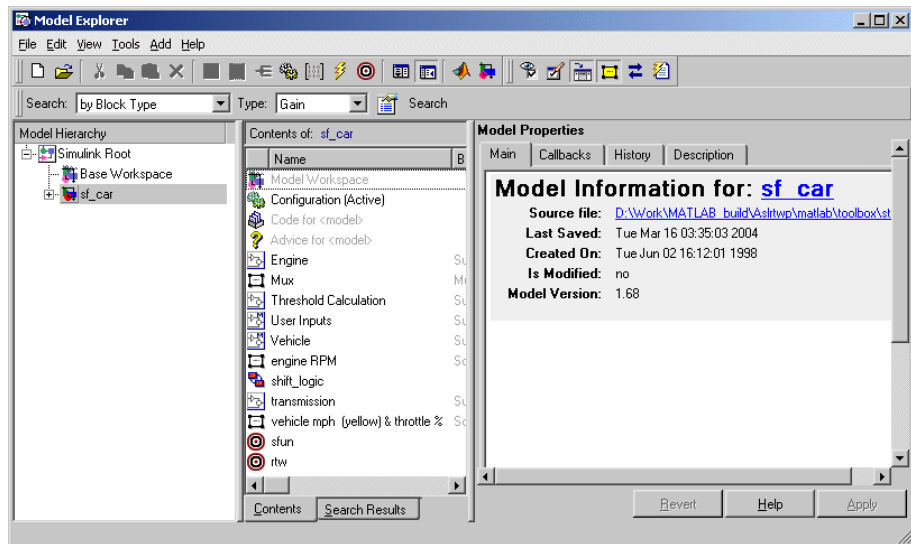
Synchronizing a Model with DOORS


In "Starting the Requirements Management Interface for DOORS" on page 3-6, you open MATLAB, open DOORS, and open a project in DOORS. Begin

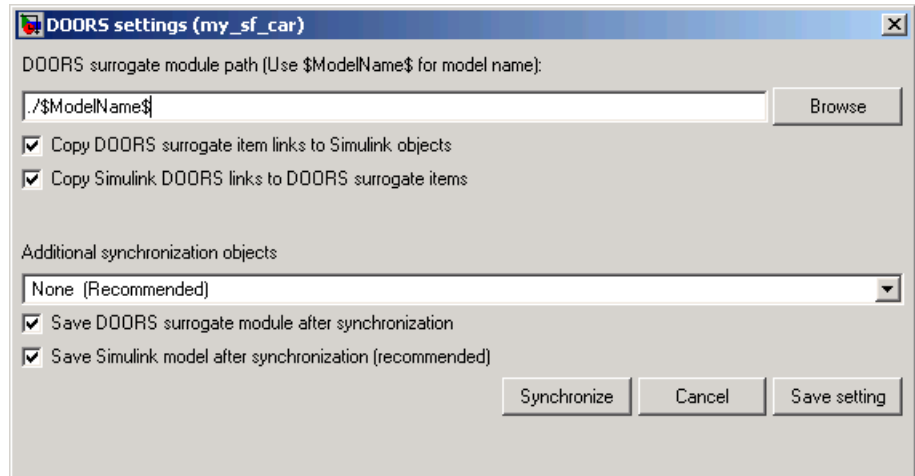
the process of mapping requirements in DOORS to a Simulink model by first synchronizing the model with the open project in DOORS. Synchronization maps a hierarchical representation of a Simulink model's blocks and Stateflow objects to a formal module in a DOORS project. Later, you use this formal module to add requirements.

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

- 1 In the MATLAB Command Window, type `sf_car` at the MATLAB prompt to open the demo model `sf_car.mdl`.
- 2 In the Simulink model, from the **View** menu, select **Model Explorer**.

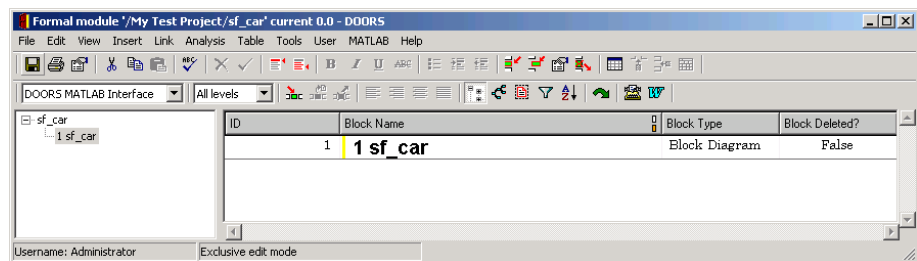


- 3 Select the Synchronize Requirements with DOORS tool  in the Model Explorer window. The DOORS settings dialog box opens, as shown.



4 Click **Synchronize**.

Synchronizing creates and opens a DOORS formal module for the model, which appears as shown.




Notice that by default the DOORS formal module contains only one synchronized object, which corresponds to the top-level diagram. To include all the model blocks in the DOORS formal module, use the following procedure, “Customizing the Level of Synchronization Detail” on page 3-16.

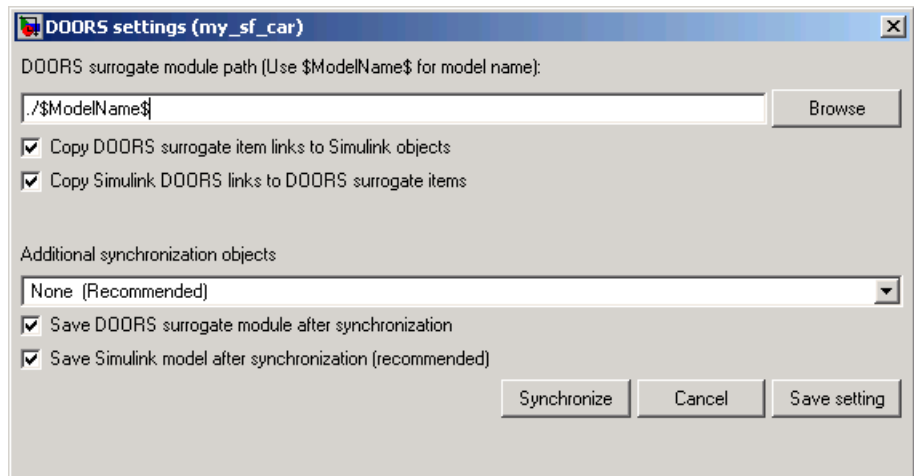
Customizing the Level of Synchronization Detail

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

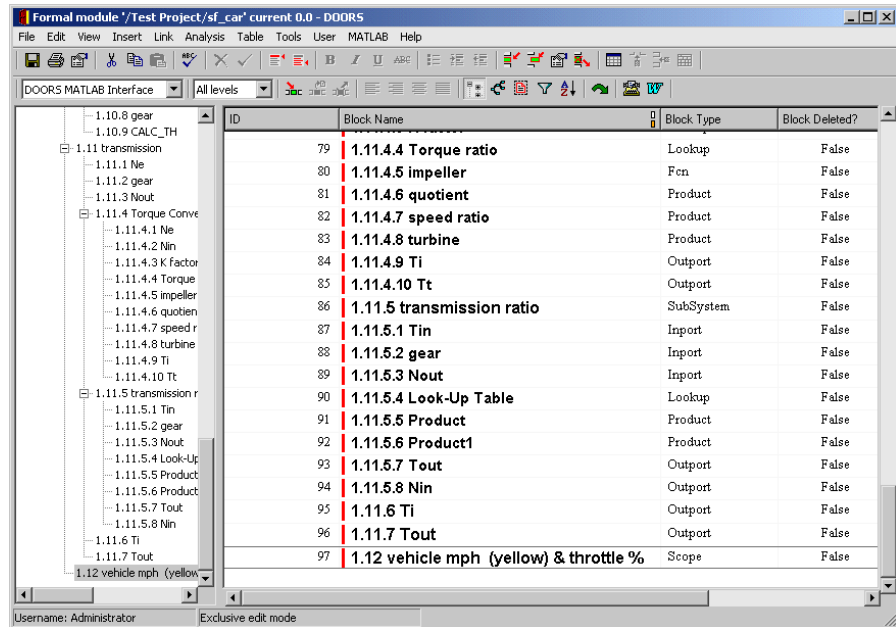
model. Additional synchronization objects improve the surrogate detail at the expense of slower synchronization.

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

- 1 Open the sf_car model.
- 2 From the **Tools** menu in the Simulink window select **Requirements > Synchronize with DOORS**. The DOORS settings dialog box opens, as shown. Another way to access this dialog box is to select the Synchronize Requirements with DOORS tool  in the Model Explorer window.




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

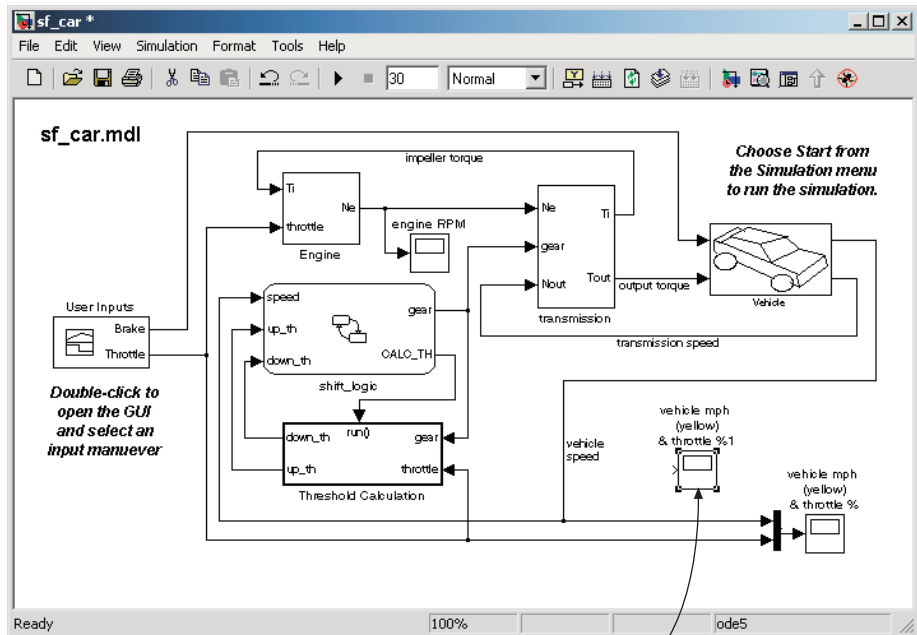


Notice the following:

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

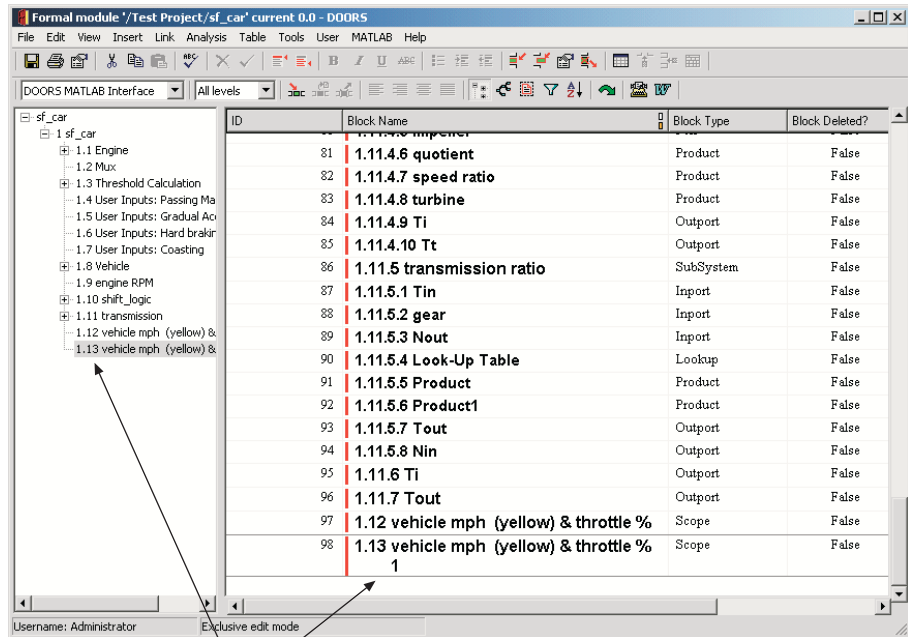
- You can add additional information columns to the right pane with the Insert Column tool  in the DOORS toolbar.

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



6 Select the Synchronize Requirements with DOORS tool  again.

The synchronized module is updated with the new block, as shown.



New block and link

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


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

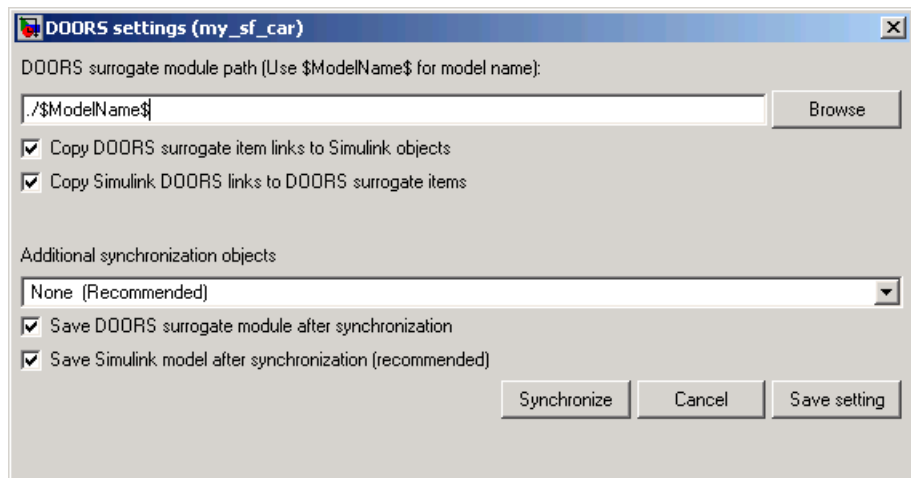
The deleted block appears at the bottom of the list of objects in the formal module and its entry in the **Block Deleted** column is True. If you want, you can delete this entry by right-clicking the line and selecting **Delete**. Otherwise, the module records the former presence of the deleted block.

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

Customizing the DOORS Synchronization Settings

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

- 1 From the **Tools** menu in the Simulink window select **Requirements > Synchronize with DOORS**. The DOORS settings dialog box opens, as shown. Another way to access this dialog box is to select the Synchronize Requirements with DOORS tool  in the Model Explorer window.

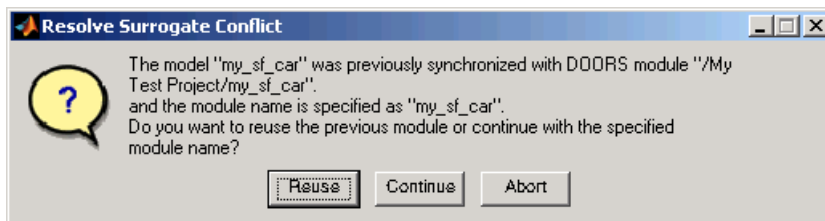


The **DOORS surrogate module path** field identifies the module within the DOORS database. You can specify a module with either a relative path (starting with `./`) or a full path (starting with `/`). Relative paths are appended to the current DOORS project. Absolute paths must specify a project and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

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

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

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



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

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

Keeping both these check boxes selected ensures that your requirement link information is completely synchronized.

- **Additional synchronization objects** — Lets you select the level of synchronization detail, as described in “Customizing the Level of Synchronization Detail” on page 3-16.
- **Save DOORS surrogate module after synchronization** — If this check box is selected, the DOORS formal modules are automatically saved upon synchronization. If you clear the check box, you will have to manually save them.
- **Save Simulink model after synchronization (recommended)** — If this check box is selected, the Simulink model is automatically saved upon synchronization. It is recommended that you use this option.

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

Linking Requirements to the DOORS Synchronized Module

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

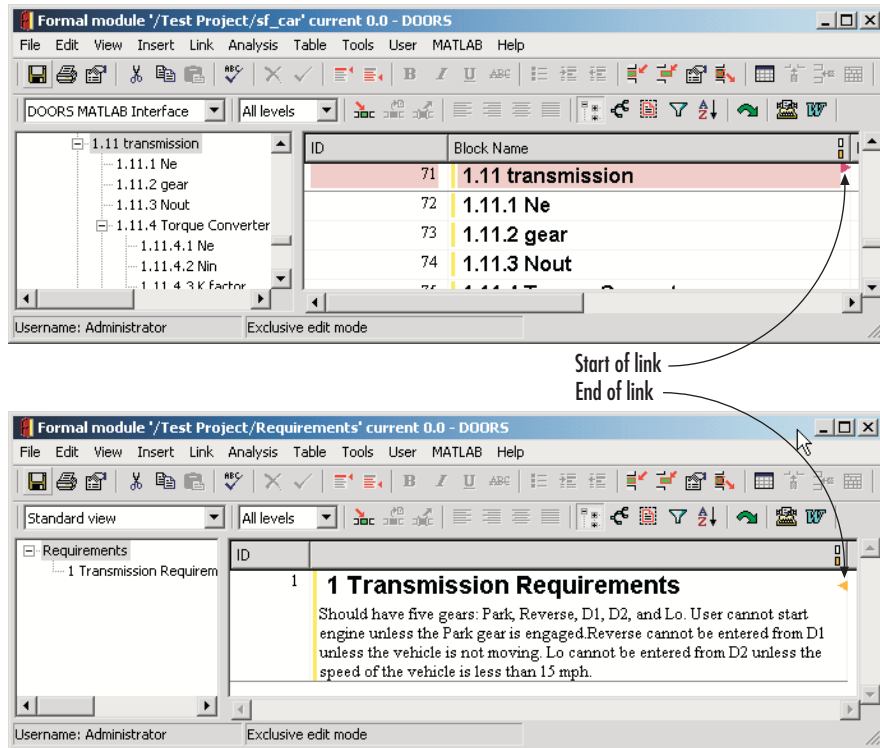
In “Creating a DOORS Requirement Object” on page 3-8, you created a Transmission Requirements object in the Requirements formal module in DOORS.

Now use the following procedure to add this requirement to the synchronized module you created for the `sf_car` model in “Synchronizing DOORS with the Simulink Model” on page 3-13:

- 1** Open the Requirements formal module in DOORS.
- 2** In the DOORS main window, open the synchronized module `sf_car` and scroll down to the `transmission` object.
- 3** Right-click the `transmission` object and select **Link > Start Link** from the resulting context menus.
- 4** In the Requirements formal module window, right-click the Transmission Requirements object and select **Link > Make Link from Start** from the resulting context menus.

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

3 Managing Model Requirements with DOORS



The requirement you install in this section is an example of an official DOORS requirement for the Requirements Management Interface for DOORS. You can navigate between the object in the synchronized module and its requirement in DOORS by right-clicking one of the arrows and selecting from the resulting pop-up menu. You can also establish more links from the object to other requirements. Later on, when you display Simulink objects with DOORS requirements in “Navigating Between Model Objects and DOORS” on page 3-25, these are the requirements that the Requirements Management Interface for DOORS detects.

Navigating Between Model Objects and DOORS

In this section...
“Viewing Model Elements with Requirements” on page 3-25
“Navigating from Simulink to DOORS” on page 3-27
“Navigating from DOORS to Simulink” on page 3-29

Viewing Model Elements with Requirements

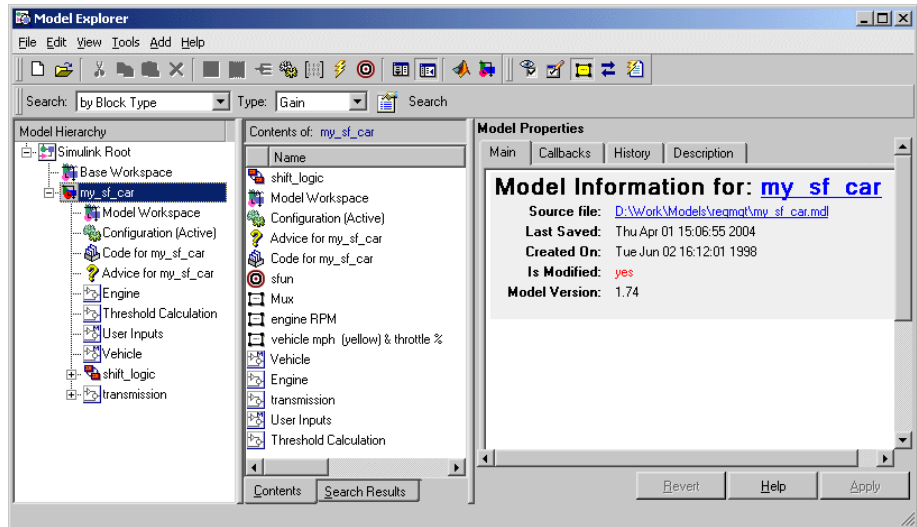
It is sometimes helpful to distinguish model objects with requirements from those without requirements in a single glance. The Requirements Management Interface for DOORS lets you see model elements with requirements linked to the synchronized module both in Simulink and in the Model Explorer.


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

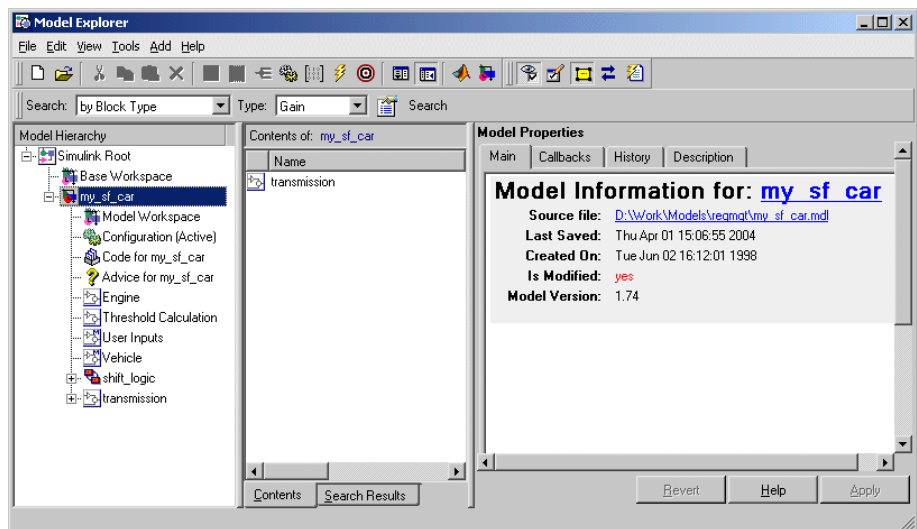
- 1 In the Simulink model, from the **View** menu, select **Model Explorer**.

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


3 Managing Model Requirements with DOORS



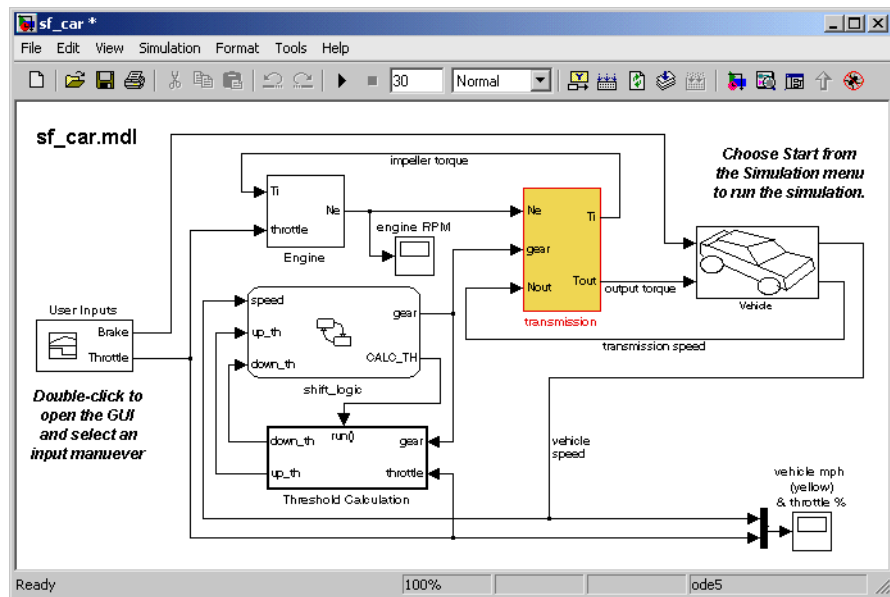
2 Select the Display Objects with Linked Requirements tool  in the Model Explorer toolbar.



The Model Explorer displays only the transmission object, which you added requirements to in “Linking Objects to DOORS Requirements” on page 3-8.

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

The transmission block in the Simulink model becomes highlighted, as shown.

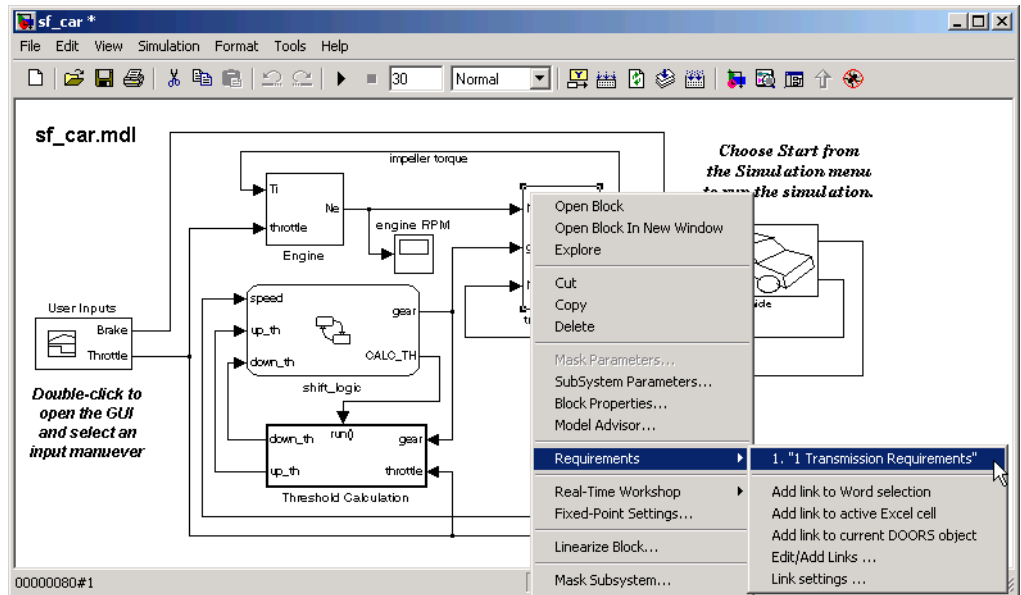


Navigating from Simulink to DOORS

If you create requirement links directly from the Simulink or Stateflow object, you can navigate directly from the object to the DOORS requirement. In “Linking Objects to DOORS Requirements” on page 3-8, you create a link from the transmission block in Simulink to the Transmission Requirements DOORS object.

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

- 1 In Simulink, open the model sf_car_doors.
- 2 Right-click on the transmission block and, from the resulting pop-up menu, select **Requirements > 1. "Transmission Requirements"** as shown.



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

Navigating Through the Synchronized Module

If you use the synchronized module to create requirement links to DOORS, as described in “Linking Requirements to the DOORS Synchronized Module” on page 3-23, then you can navigate between Simulink objects and DOORS requirements by using the synchronized module as an intermediary. You first navigate to the unique object in the synchronized module from its object in Simulink or the Model Explorer. From the synchronized module, you then access requirements for each object through the linking process in DOORS.

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

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

A pop-up menu appears.

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

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

- 3** Access individual requirements in the synchronized module.

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

Navigating from DOORS to Simulink

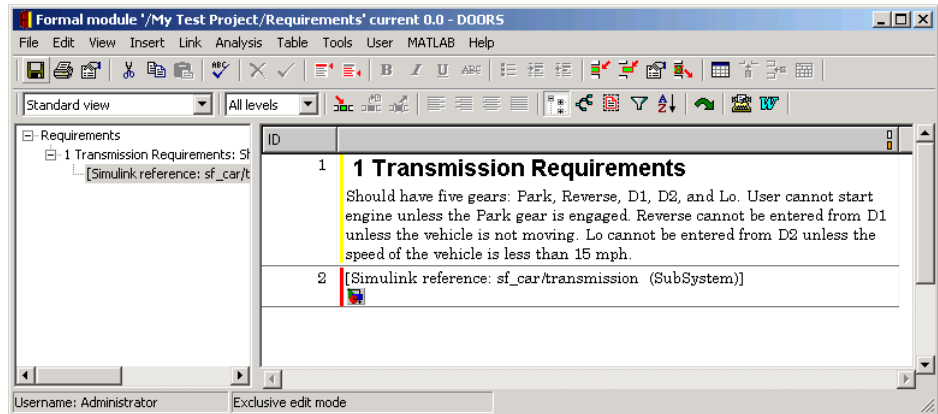
If you create two-way requirement links directly from the Simulink or Stateflow object, you can navigate from the DOORS requirement directly to the associated object in Simulink or Stateflow. In “Linking Objects to DOORS Requirements” on page 3-8, you create a link from the transmission block in Simulink to the Transmission Requirements DOORS object.

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

- 1** In DOORS, open the formal module window Requirements.

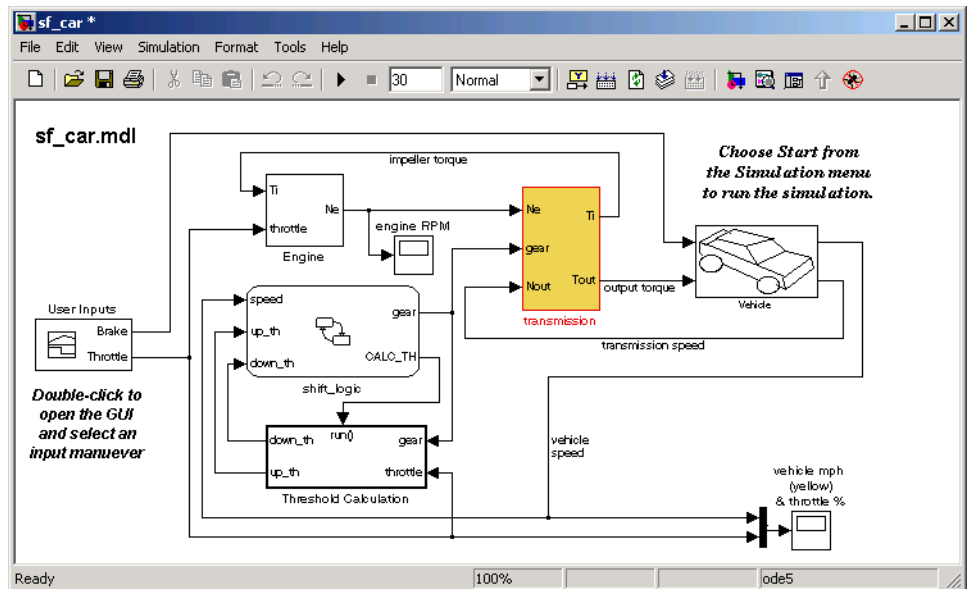
- 2** Select the Simulink Reference sub-node of the Transmission Requirements node in the left pane, as shown.

3 Managing Model Requirements with DOORS



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

The transmission block in the Simulink diagram is highlighted, as shown.



Navigating Through the Synchronized Module

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

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

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

The object opens in its native diagram as follows:

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

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

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

Managing Model Verification Blocks

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

Using Model Verification Blocks
(p. 4-2)

Using Model Verification blocks in Simulink to monitor model signals against a specified limit

Using the Verification Manager
(p. 4-7)

Managing the Model Verification blocks in your models with the Verification Manager

Managing Verification Requirements
(p. 4-24)

Linking requirements documents to test groups and their Model Verification block schedule in the Verification Manager

Using Model Verification Blocks

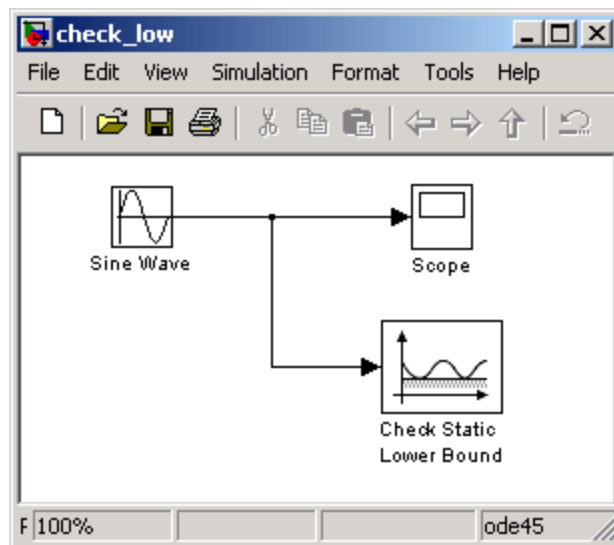
You use Model Verification blocks throughout your model to monitor model signals. You can set a verification block to assert when its signal leaves the specified limit or range. During simulation, when the signal crosses the limit, the verification block can

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

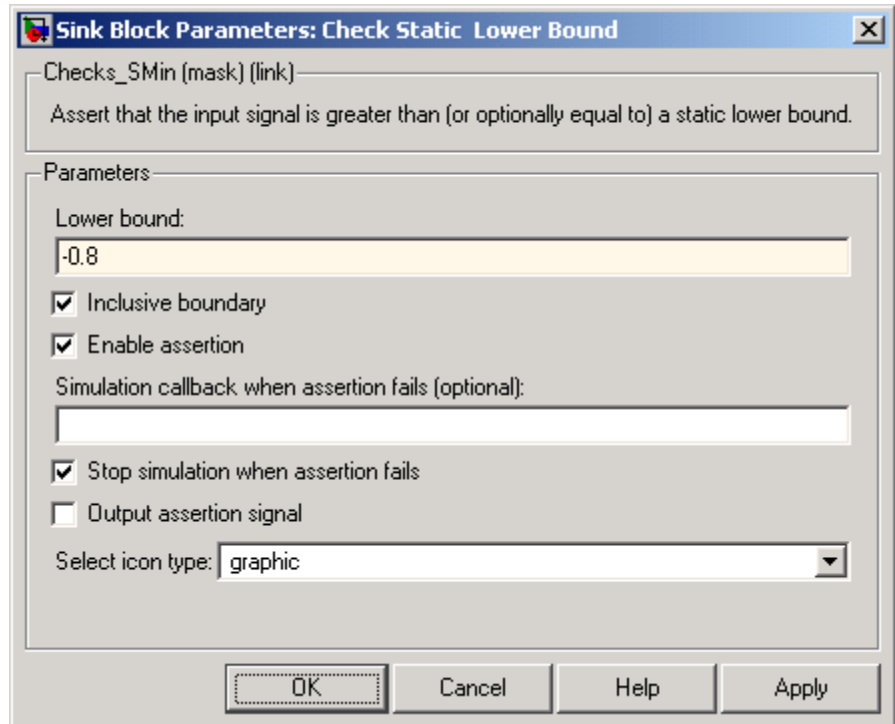
To see a complete list of all Model Verification blocks and references for each, see the “Model Verification” category in the Simulink Block Reference documentation.

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

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



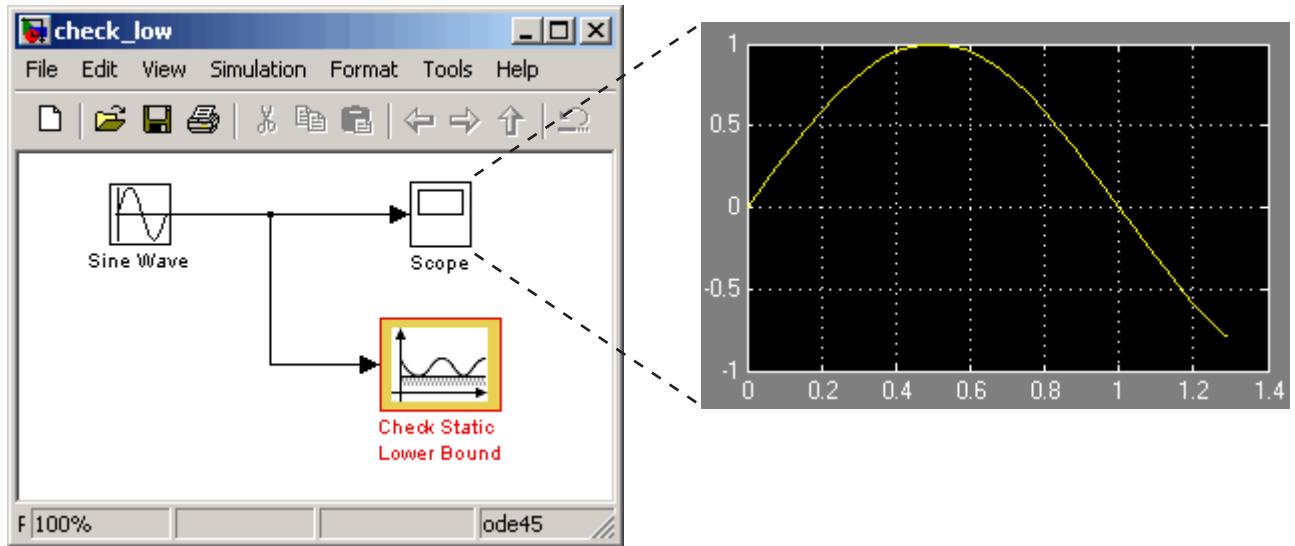
- 2 Set the model to run for 2 seconds while the Sine Wave block outputs a signal with an amplitude of 1 and a frequency of π radians per second.
- 3 Open the Check Static Lower Bound block and set the parameters as follows:



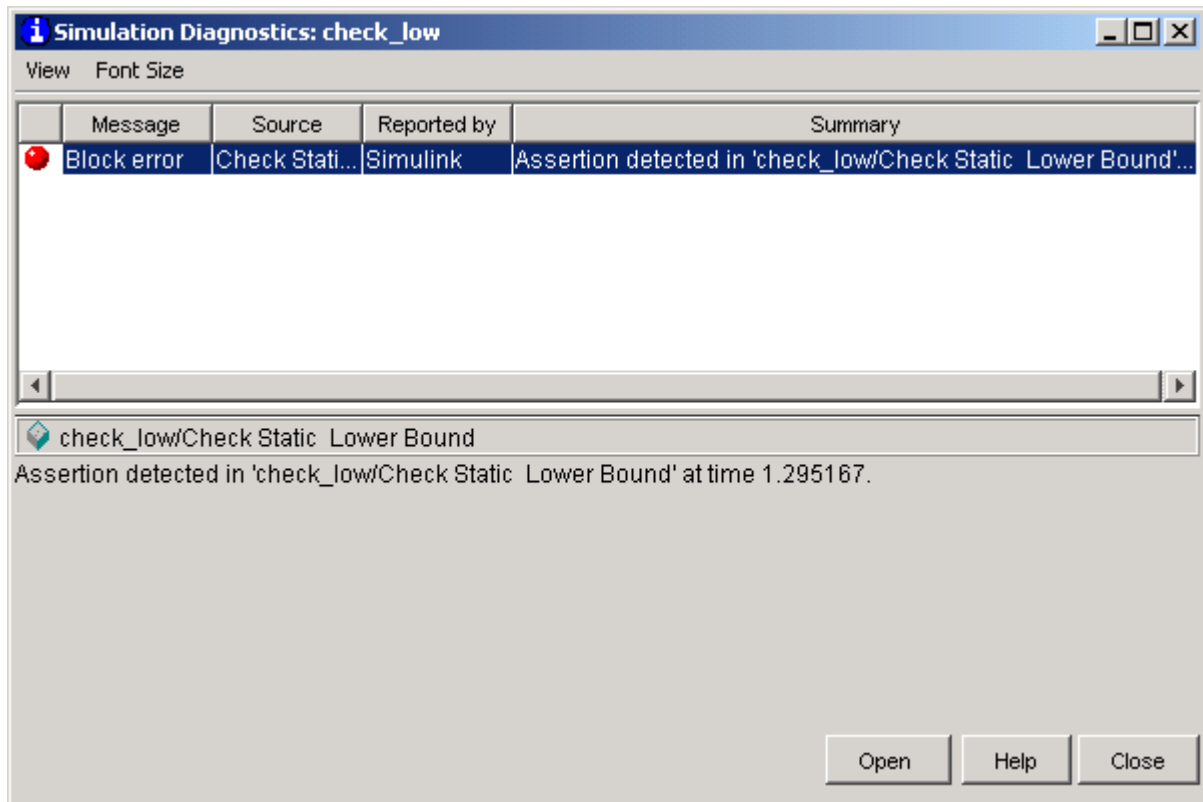
A verification block is enabled for an assertion when the **Enable assertion** check box is selected (this is the default setting). According to the preceding property settings, the Check Static Lower Bound block is set to detect a signal value of -0.8 or lower. If this signal is detected, simulation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

- 4 Run the simulation.

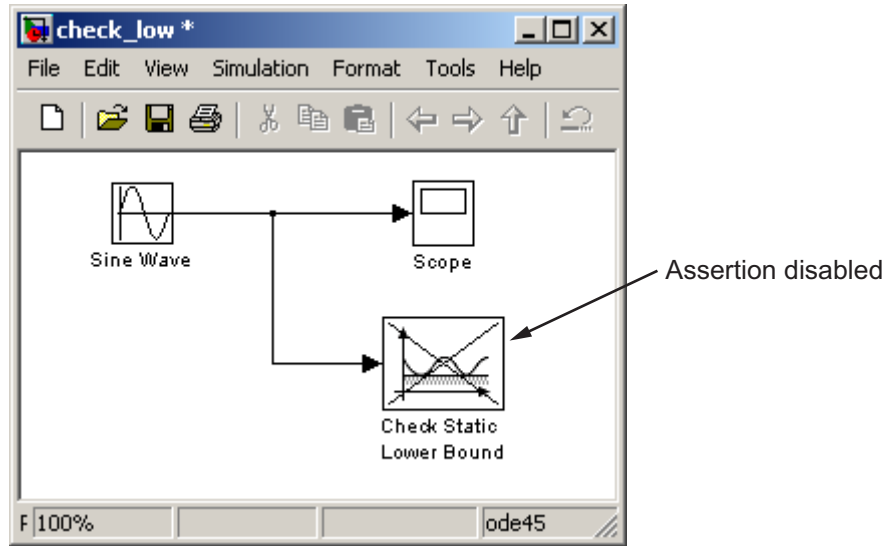
The model stops simulating after 1.295 seconds, when the output is -0.8, as shown. This brings focus to the asserting verification block, which is highlighted.



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



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



Using the Verification Manager

In this section...
“What Is the Verification Manager?” on page 4-7
“Opening the Verification Manager” on page 4-7
“Enabling and Disabling Model Verification Blocks with the Verification Manager” on page 4-15
“Using Enabling and Disabling Tools in the Verification Manager” on page 4-20

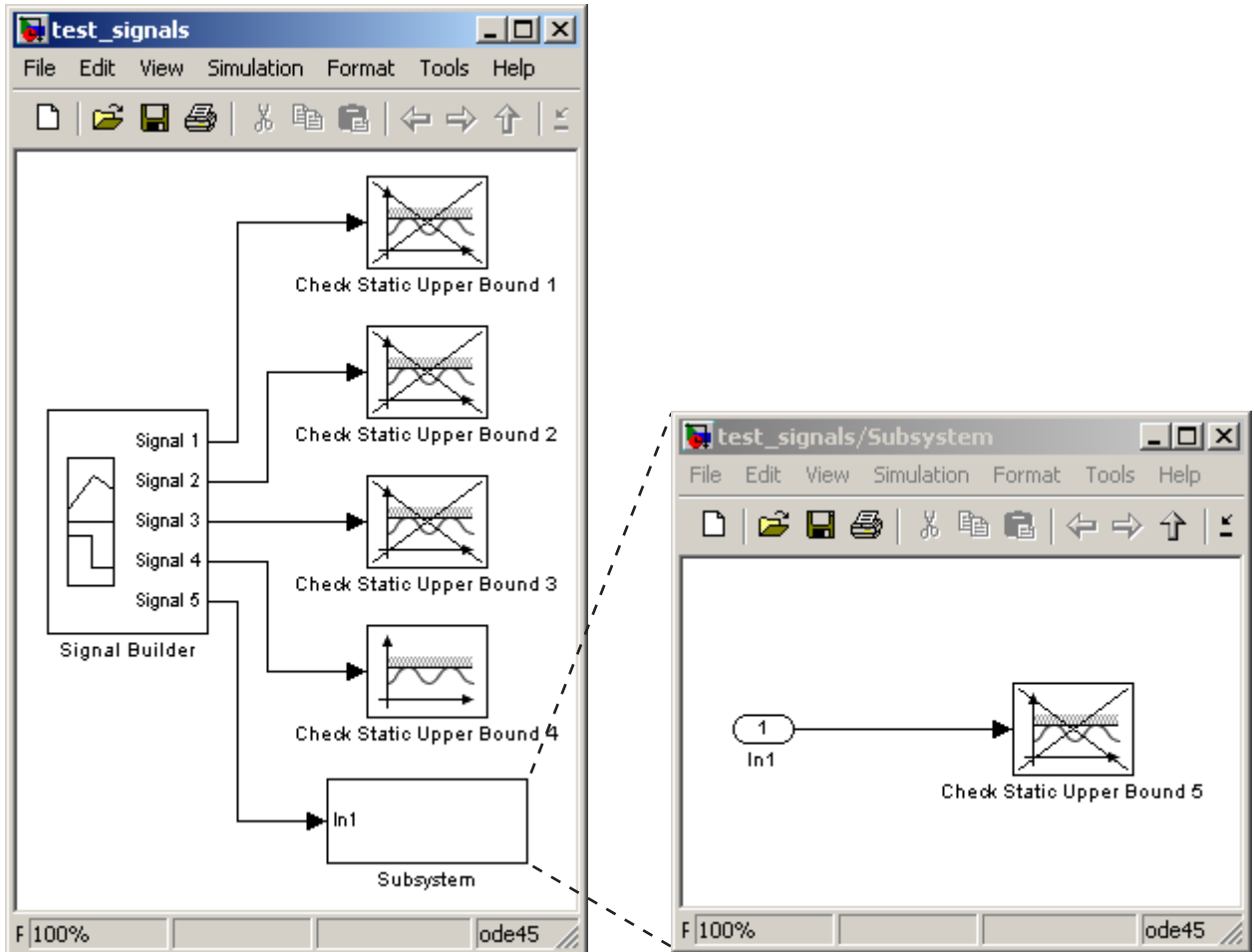
What Is the Verification Manager?

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

Opening the Verification Manager

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

- 1 Create the following example model in Simulink.

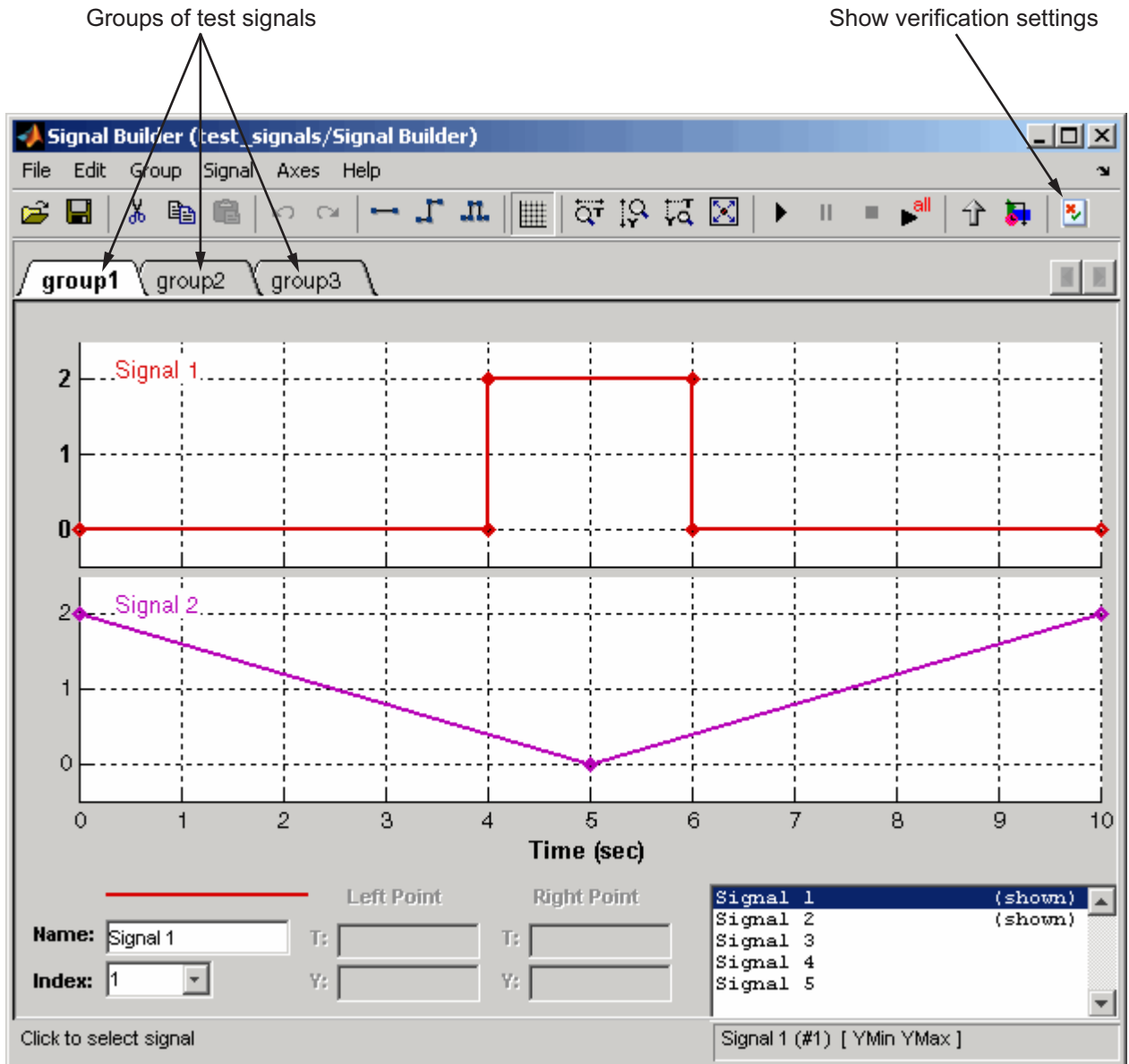


Typically, a Signal Builder block provides test signals for an entire model from one location. The example model contains a Signal Builder block feeding five test signals to Model Verification blocks. Signals 1 through 4 are sent directly to Check Static Upper Bound Model Verification blocks. The fifth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

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


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

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

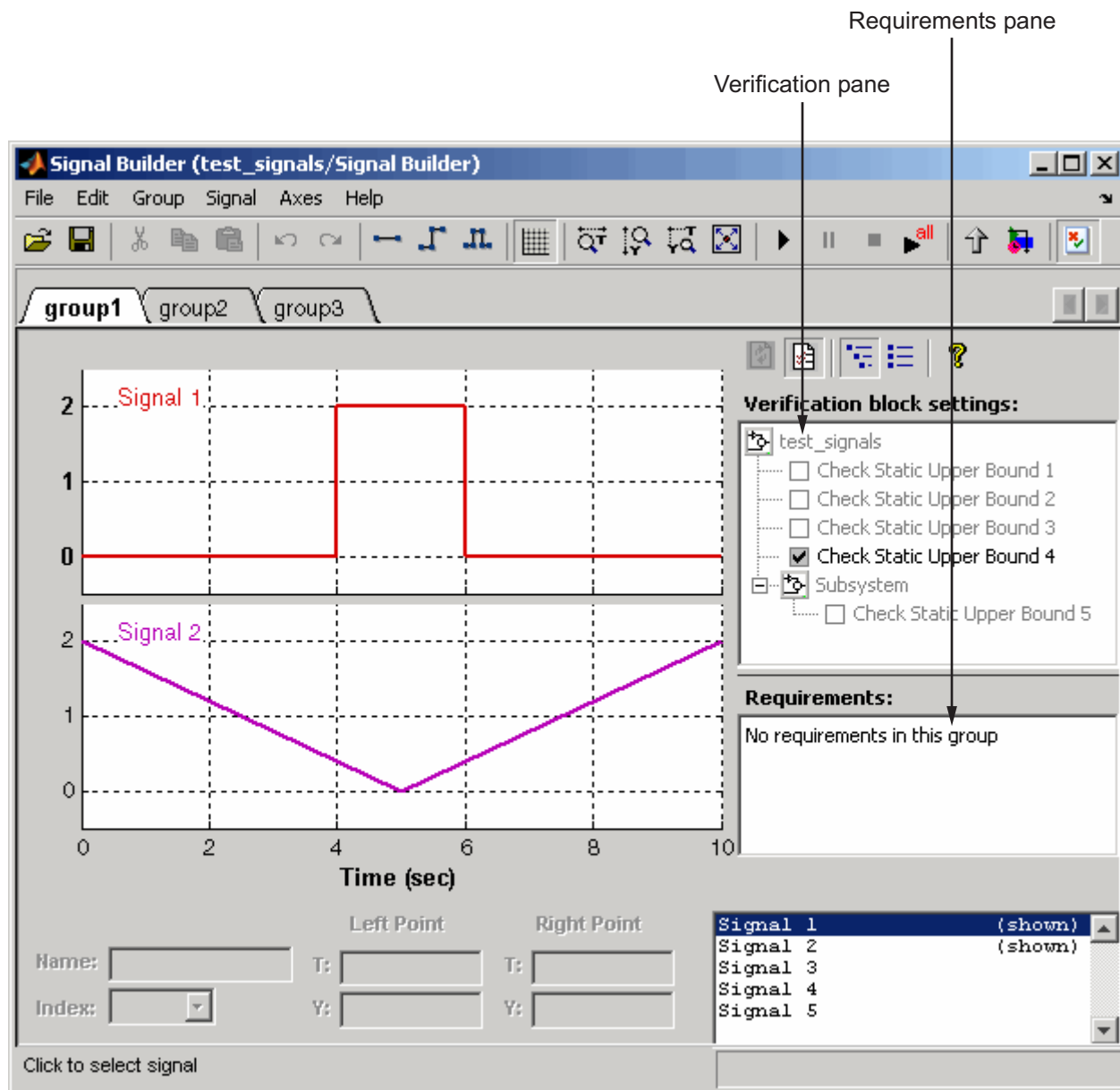


The Signal Builder dialog box displays tabbed pages for three groups of signal values. Each group contains independent values for all five signals.


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

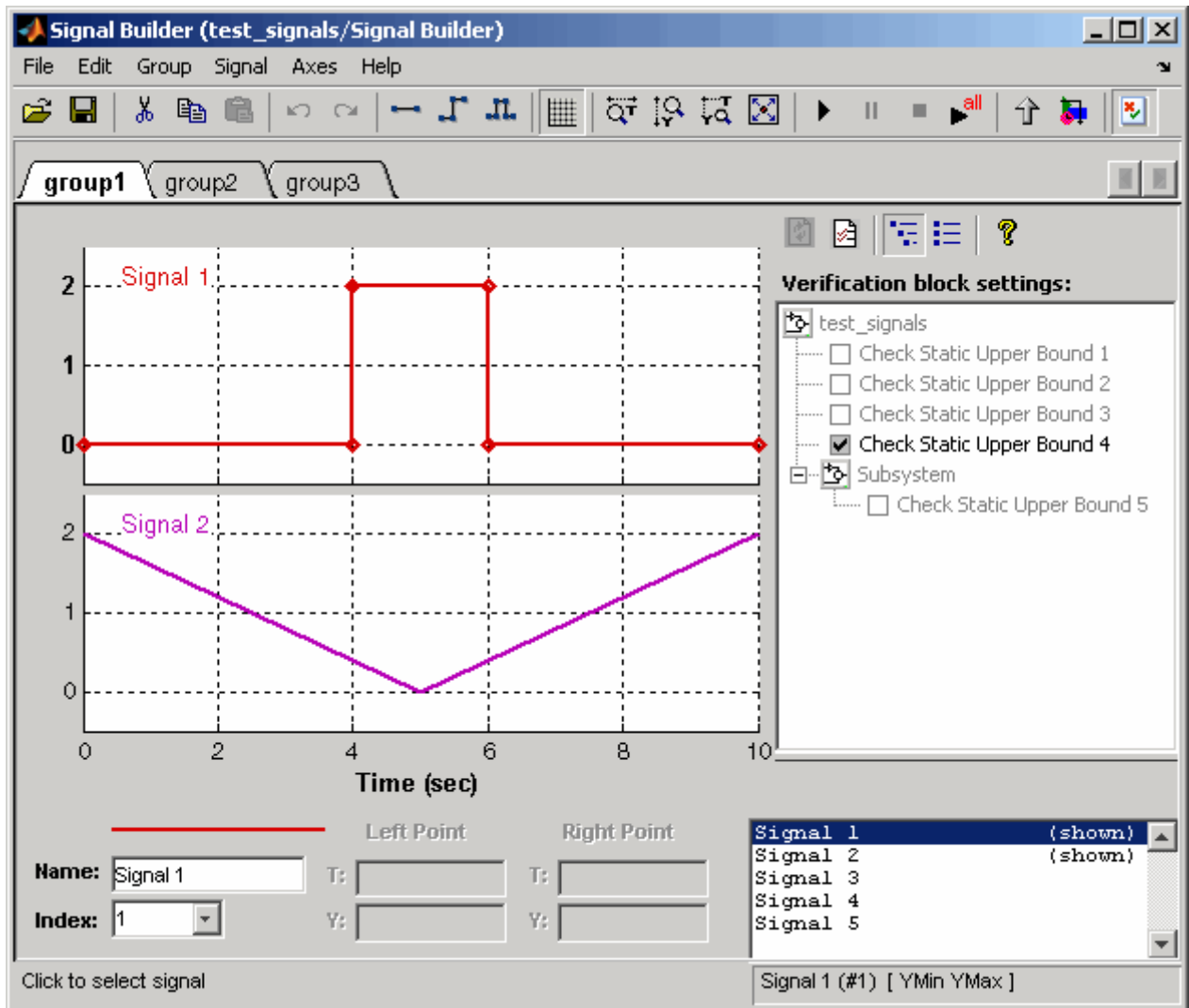
- 3** In the Signal Builder dialog toolbar, select the Show Verification Settings tool .

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




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

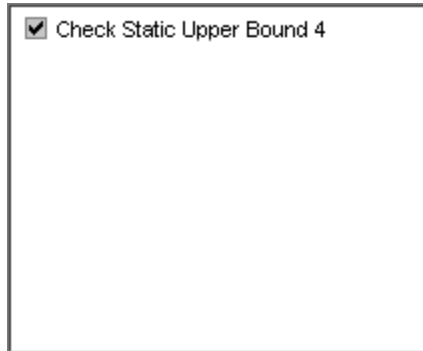
- 4** Just above the **Verification block settings** pane, select  to close the **Requirements** pane.




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

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

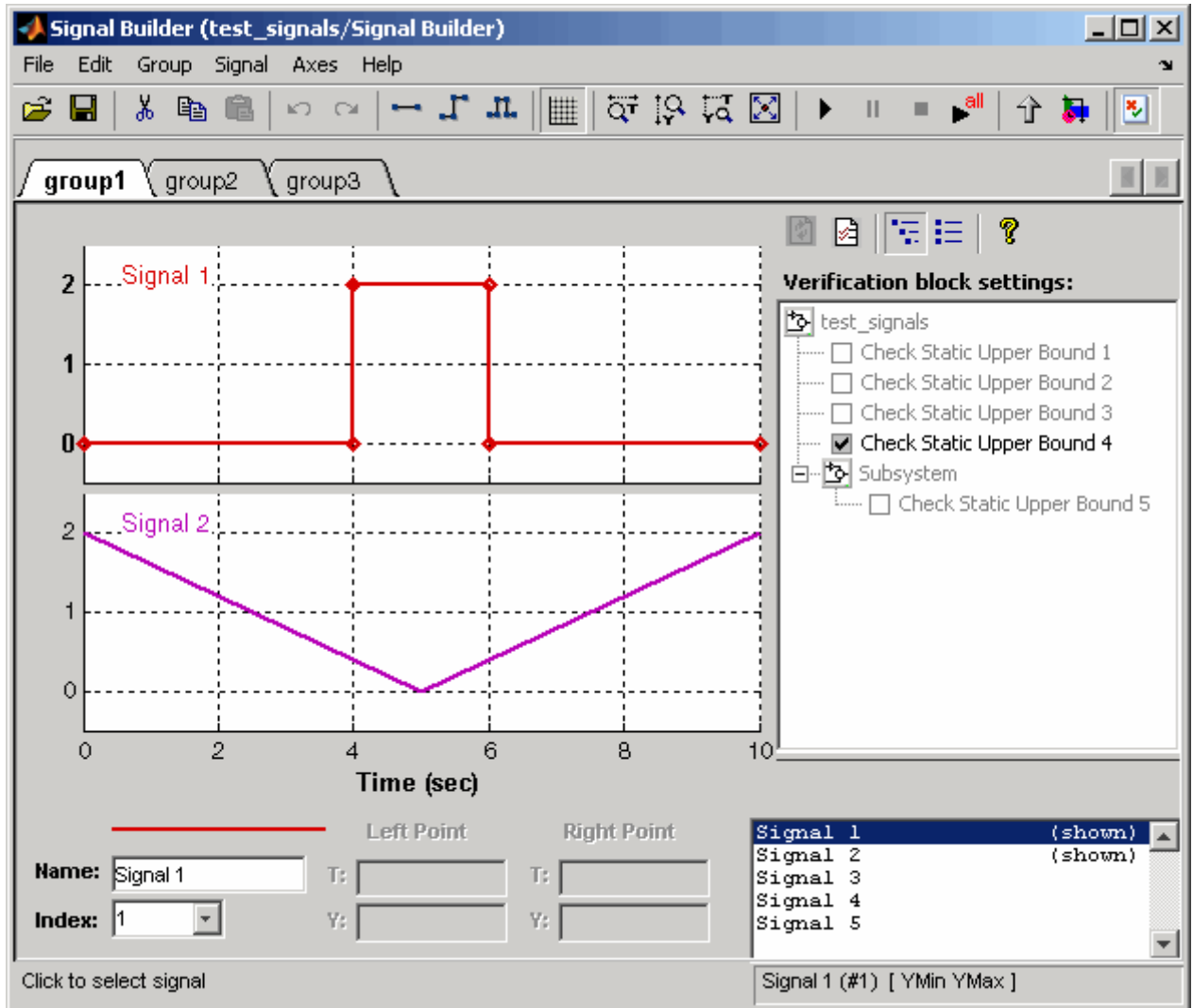
The **Verification block settings** pane now shows only the enabled Model Verification blocks for the current group, as shown.



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

Enabling and Disabling Model Verification Blocks with the Verification Manager

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



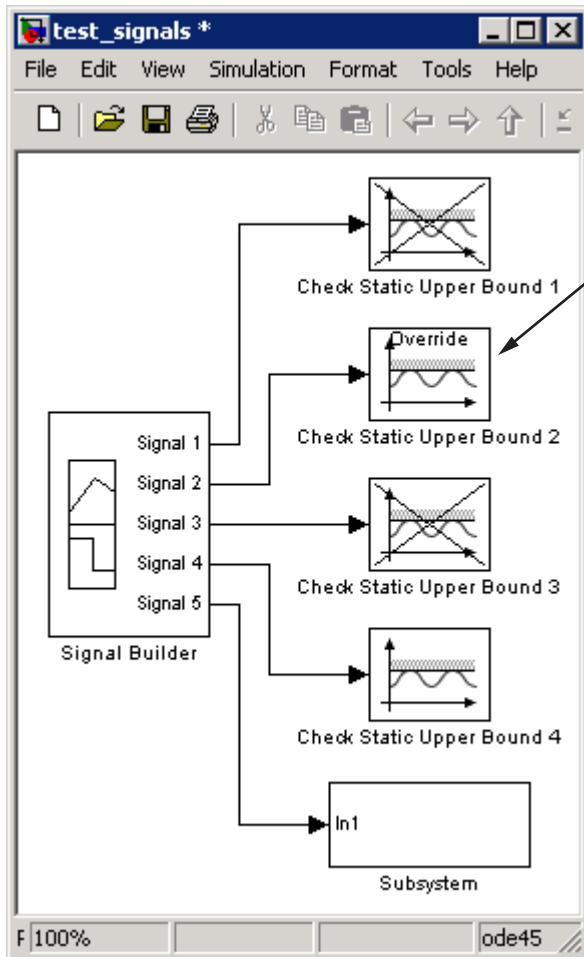
The **Verification block settings** pane in the preceding example lists the Model Verification blocks in the model. Each verification block has a preceding status node that indicates whether its assertion is enabled or disabled and whether that setting applies universally or to the active group. The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

Node	Status
<input type="checkbox"/>	Verification block is disabled for this group. Click to enable for current group.
<input checked="" type="checkbox"/>	Verification block is enabled for the current group. Click to disable for current group.
<input checked="" type="checkbox"/>	Verification block is enabled for all test groups.

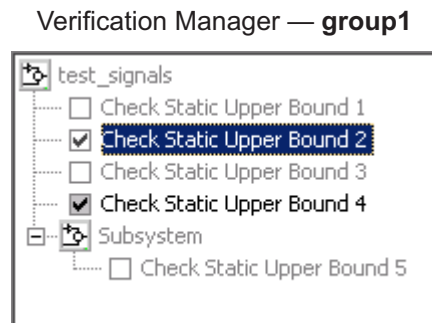
Use the Verification Manager to enable or disable model verification blocks in the `test_signals` model you created in “Opening the Verification Manager” on page 4-7, as follows:

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

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

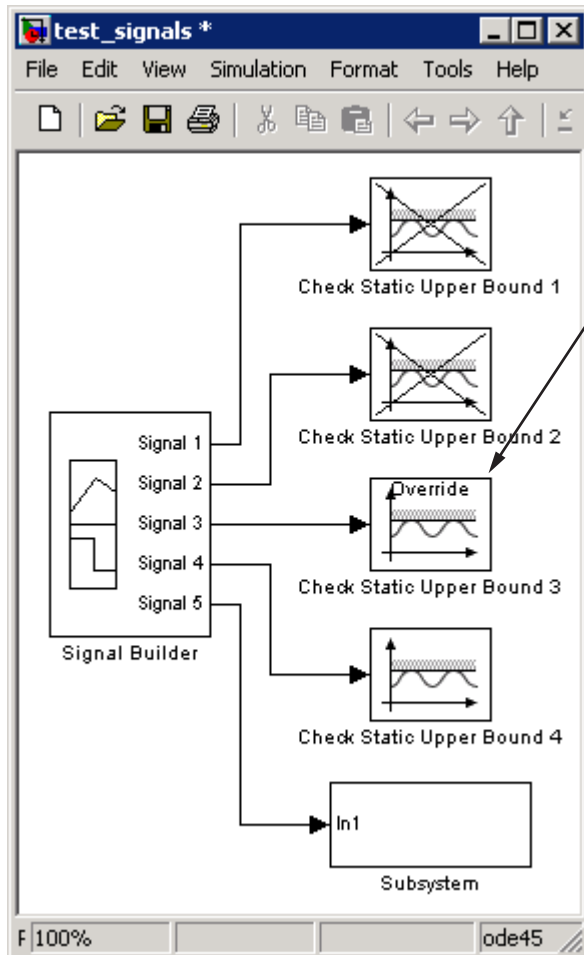


Disabled but enabled in current group (**group1**)



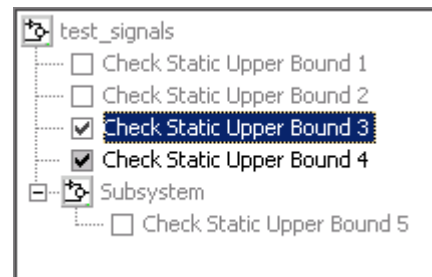
Because it is enabled in the current group, the Check Static Upper Bound 2 block gains an **Override** label and loses its cross-out. The meaning behind the change in appearance becomes clearer when another group is selected.

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






Disabled but enabled in current group (**group2**)

Verification Manager — **group2**







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

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

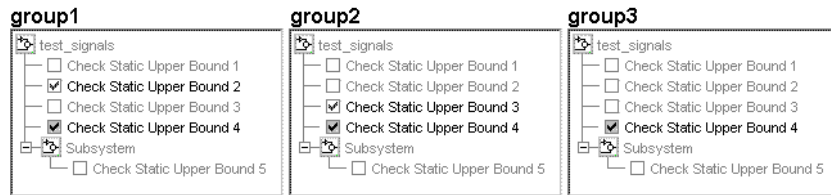
Assert Block	Description
	Enabled for all groups
	Disabled in current group
	Enabled in current group

Using Enabling and Disabling Tools in the Verification Manager

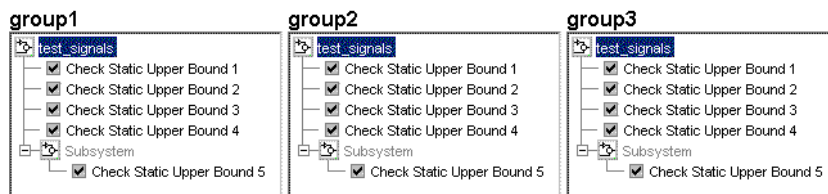
If you have many verification blocks, it is tedious to enable and disable blocks individually. For this reason, the Verification Manager lets you enable and disable blocks through selections from a context menu. These selections vary with the node as follows:

Node	Context Menu Selections
	<ul style="list-style-type: none"> • Contents enable for all groups • Contents enable by group • Contents group enable • Contents group disable
	<ul style="list-style-type: none"> • Block enable by group
	<ul style="list-style-type: none"> • Block enable for all groups • Block group enable
	<ul style="list-style-type: none"> • Block enable for all groups • Block group disable

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

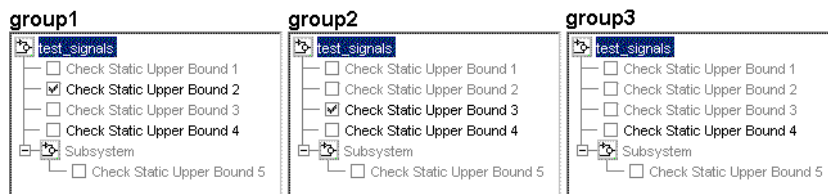


- 1 Right-click the `test_signals` node and select **Contents enable for all groups**.



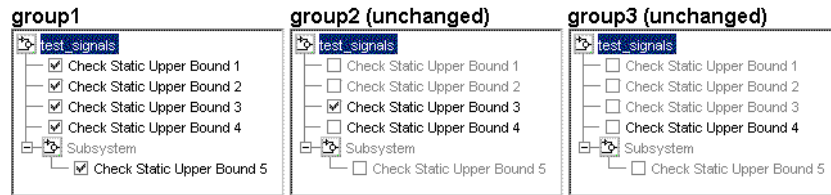
Applying the **Contents enable for all groups** selection to the model node enables all contained Model Verification blocks, for all test groups, in all contained subsystems.

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



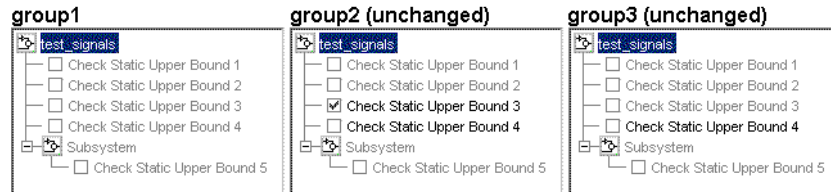
Applying the **Contents enable by group** selection to the model node restores the previous individually enabled/disabled settings for each block in each group.

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



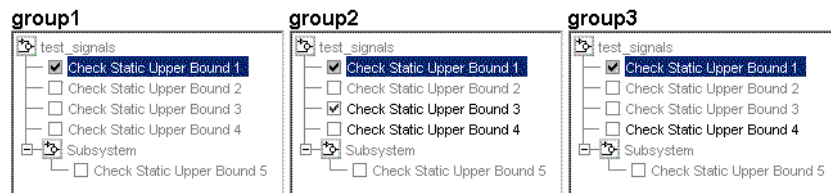
Applying **Contents Group enable** to the `test_signals` model node in **group1** individually enables all contained blocks for **group1**, but leaves the other groups untouched.

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



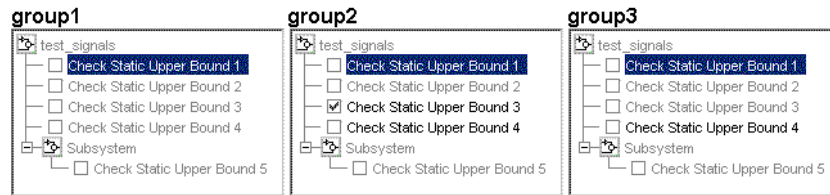
Applying **Contents group disable** to the `test_signals` model node in **group1** individually disables all contained blocks for **group1**, but leaves the other groups untouched.

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



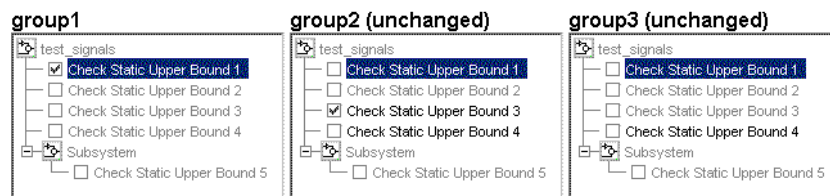
Applying **Block enable for all groups** to the individual **group1** block node for Check Static Upper Bound 1 in **group1** enables this block for all groups.

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



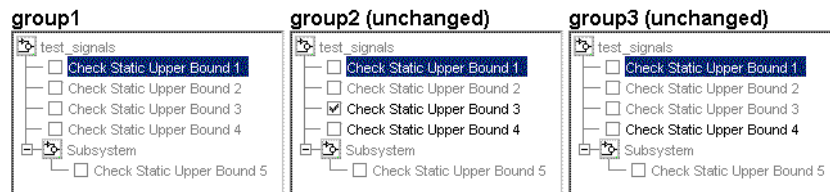
Applying **Block enable by group** to the individual **group1** block node for Check Static Upper Bound 1 in **group1** restores the previous individually enabled/disabled state to this block for all groups. This lets you enable or disable this node individually for each group.

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



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

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



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

Managing Verification Requirements

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

You can link requirements documents to individual verification blocks just as you can for any Simulink block. See “Adding Requirement Links to an Object” on page 2-8 for details on linking requirements documents to individual Simulink blocks.

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

Requirements pane

The screenshot shows the Signal Builder application window titled "Signal Builder (test_signals/Signal Builder)". The interface includes a menu bar (File, Edit, Group, Signal, Axes, Help), a toolbar with various icons, and a workspace with three tabs labeled "group1", "group2", and "group3".

The main workspace contains two signal plots. The top plot, labeled "Signal 1", shows a red square wave that is at 0 from 0 to 4 seconds, jumps to 2 from 4 to 6 seconds, and returns to 0 from 6 to 10 seconds. The bottom plot, labeled "Signal 2", shows a purple V-shaped wave that starts at 2 at 0 seconds, reaches 0 at 5 seconds, and returns to 2 at 10 seconds. The x-axis for both plots is "Time (sec)" ranging from 0 to 10, and the y-axis ranges from 0 to 2.

On the right side, there is a "Verification block settings" pane. It shows a tree view with "test_signals" expanded, containing five checkboxes: "Check Static Upper Bound 1", "Check Static Upper Bound 2", "Check Static Upper Bound 3", "Check Static Upper Bound 4" (which is checked), and "Check Static Upper Bound 5". Below this is a "Requirements:" pane that currently displays "No requirements in this group".

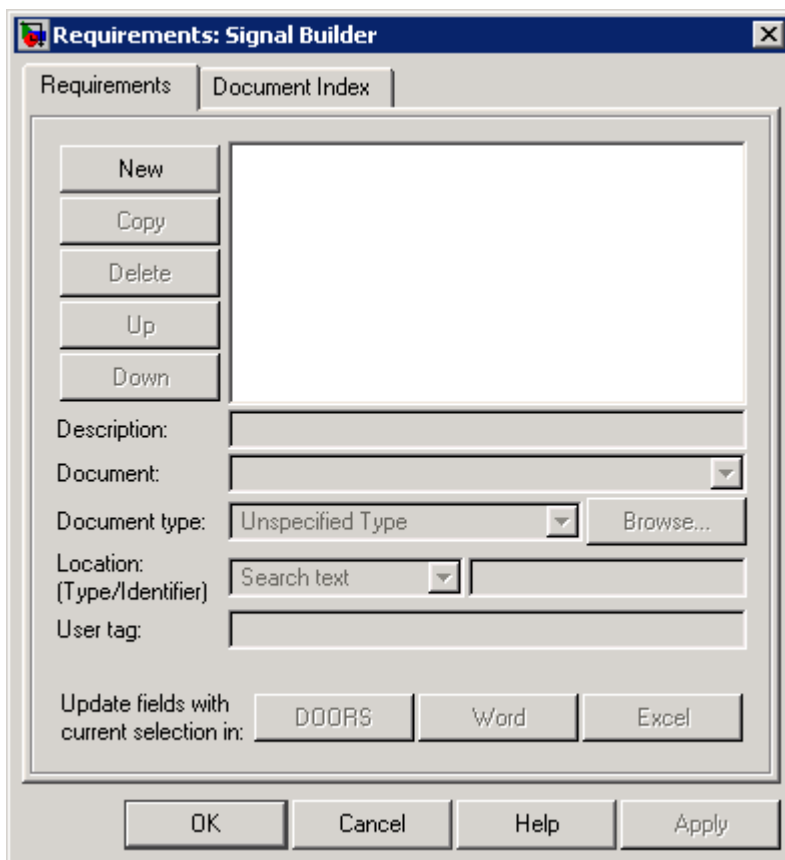
At the bottom of the workspace, there are input fields for "Name:", "Index:", "Left Point" (with T and Y sub-fields), and "Right Point" (with T and Y sub-fields). A "Click to select signal" button is located at the bottom left. A list box on the bottom right contains the following items: "Signal 1 (shown)", "Signal 2 (shown)", "Signal 3", "Signal 4", and "Signal 5".

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

A pop-up menu appears.

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

The Requirements dialog box appears, as shown.



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

- Add links to requirements documents as described in steps 4 through 9 of “Adding Requirement Links to an Object” on page 2-8.

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

New requirements

The screenshot shows the Signal Builder application window with the following components:

- Signal 1 Plot:** A red square wave signal that is at 0 from 0 to 4 seconds, jumps to 2 from 4 to 6 seconds, and returns to 0 from 6 to 10 seconds.
- Signal 2 Plot:** A purple V-shaped signal that starts at 2 at 0 seconds, reaches 0 at 5 seconds, and returns to 2 at 10 seconds.
- Verification block settings:** A tree view showing a list of checks:
 - test_signals
 - Check Static Upper Bound 1
 - Check Static Upper Bound 2
 - Check Static Upper Bound 3
 - Check Static Upper Bound 4
 - Subsystem
 - Check Static Upper Bound 5
- Requirements:** A list containing "Requirement 1" and "Requirement 2". An arrow labeled "New requirements" points from the top right to "Requirement 2".
- Signal List:** A list at the bottom right showing "Signal 1 (shown)", "Signal 2 (shown)", "Signal 3", "Signal 4", and "Signal 5".
- Input Fields:** Fields for Name, Index, Left Point (T, Y), and Right Point (T, Y).

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

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

Using Model Coverage

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

Introduction to Model Coverage
(p. 5-3)

Introduces you to the concept of model coverage and how it measures the effectiveness of your model during testing

Using Model Coverage (p. 5-8)

Shows you how to use model coverage and the tests to determine the effectiveness of your model during testing

Specifying Model Coverage Reporting Options (p. 5-12)

Shows you how to select the model coverage measurements and reports that are performed during simulation

Understanding Model Coverage Reports (p. 5-21)

Shows and describes the different parts of a basic model coverage report

N-Dimensional Lookup Table Report (p. 5-27)

Describes the interactive chart that summarizes the extent to which elements of a Lookup Table are accessed

Signal Range Analysis Report (p. 5-33)

Lists the maximum and minimum signal values at each block in the model measured during simulation

Colored Simulink Diagram Coverage Display (p. 5-36)	Shows you how to use the option to display coverage of a model by coloring its elements
Using Model Coverage Commands (p. 5-41)	Shows you how to perform and report model coverage tests during simulation with MATLAB commands
Model Coverage for Referenced Models (p. 5-47)	How to use model coverage for Model blocks that represent referenced models
Model Coverage for Embedded MATLAB Function Blocks (p. 5-52)	How to use model coverage for Embedded MATLAB Function blocks and interpret the results

Introduction to Model Coverage

In this section...

- “What Is Model Coverage?” on page 5-3
- “How Model Coverage Works” on page 5-3
- “Types of Model Coverage” on page 5-3
- “Blocks That Receive Model Coverage” on page 5-5

What Is Model Coverage?

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

How Model Coverage Works

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

See “Understanding Model Coverage Reports” on page 5-21 for an example of a model coverage report along with descriptions of the coverages it contains. Before you do, you might need to review the types of coverages that model coverage performs in “Types of Model Coverage” on page 5-3.

Types of Model Coverage

The tool performs several types of coverage analysis, depending on the coverage options you select.

- “Cyclomatic Complexity” on page 5-4

- “Decision Coverage (DC)” on page 5-4
- “Condition Coverage (CC)” on page 5-4
- “Modified Condition/Decision Coverage (MC/DC)” on page 5-5
- “Lookup Table Coverage (LUT)” on page 5-5

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. In general, the McCabe complexity measure is slightly higher because of error checks that the model coverage analysis does not consider.

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

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

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

Decision Coverage (DC)

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

Condition Coverage (CC)

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

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

Modified Condition/Decision Coverage (MC/DC)

Modified condition/decision coverage examines blocks that output the logical combination of their inputs (for example, the Logic block), and Stateflow transitions to determine the extent to which the test case tests the independence of logical block inputs and transition conditions. A test case achieves full coverage for a block if, for every input, there is a pair of simulation times when changing that input alone causes a change in the block's output. A test case achieves full coverage for a transition if, for each condition on the transition, there is at least one time when a change in the condition triggers the transition.

Lookup Table Coverage (LUT)

Lookup table coverage examines blocks, such as the 1D Lookup block, that output the result of looking up one or more inputs in a table of inputs and outputs, interpolating between or extrapolating from table entries as necessary. Lookup table coverage records the frequency that table lookups use each interpolation interval. A test case achieves full coverage if it executes each interpolation and extrapolation interval at least once. For each LUT block in the model, the coverage report displays a colored map of the lookup table indicating where each interpolation was performed.

Blocks That Receive Model Coverage

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

Block	Decision	Condition	MC/DC	LUT
1D Lookup				•
2D Lookup				•
ND Lookup				•

Block	Decision	Condition	MC/DC	LUT
Interpolation Using Prelookup				•
ND Direct Lookup				•
Abs	•			
Combin. Logic	•	•		
Discrete-Time Integrator (when saturation limits are enabled)	•			
Embedded MATLAB Function	•	•	•	
Fcn (Boolean operators only)		•		
For	•			
If	•			
Logic		•	•	
MinMax	•			
Model	•	•	•	
Multiport Switch	•			
Rate Limiter	• (Relative to slew rates)			
Relay	•			

Block	Decision	Condition	MC/DC	LUT
Saturation	•			
Stateflow (see note below)	•	•	•	
Subsystem	•	•	•	
Switch	•			
SwitchCase	•			
While	•			

Note Model coverage provides decision coverage for Stateflow states, events, and state temporal logic decisions. It also provides decision, condition, and MCDC coverage for Stateflow transitions. See “Understanding Model Coverage for Stateflow Charts” in the Stateflow documentation for details on the model coverage of Stateflow charts.

Using Model Coverage

In this section...
“Basic Workflow for Using Model Coverage” on page 5-8
“Creating and Running Test Cases” on page 5-8

Basic Workflow for Using Model Coverage

To develop effective tests with model coverage,

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

Note Simulink comes with an online demonstration of the use of model coverage to validate model tests. To run the demo, enter `simcovdemo` at the MATLAB command prompt.

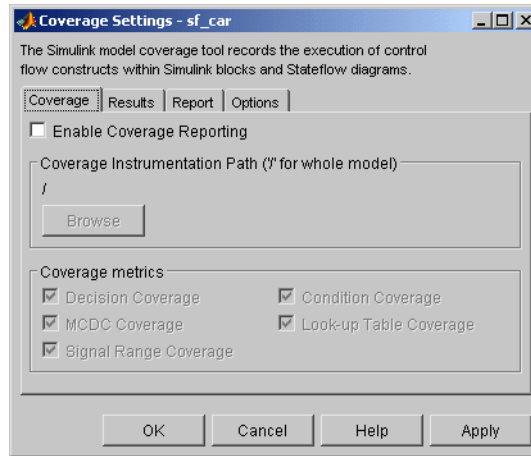
Creating and Running Test Cases

Model coverage provides two MATLAB commands, `cvtest` and `cvsim`, for creating and running test cases. The `cvtest` command creates test cases to be run by the `cvsim` command (see “Creating Tests with `cvtest`” on page 5-41 and “Running Tests with `cvsim`” on page 5-43).

You can also run the coverage tool interactively as follows:

1 From the Simulink **Tools** menu, select **Coverage Settings**.

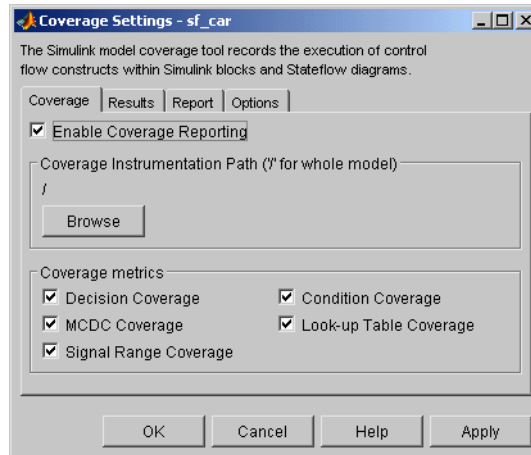
Simulink displays the Coverage Settings dialog box, as shown.



The Coverage Settings dialog box has four tabs. The **Coverage** tab is displayed by default.

2 Select **Enable Coverage Reporting**.

By default, this setting is disabled. When you select it, the **Browse** button for the **Coverage Instrumentation Path** is enabled along with the check boxes of the **Coverage metrics** section, as shown.



Selecting **Enable Coverage Reporting** also enables fields on the other tabs of the Coverage Settings dialog box.

- 3 Select the coverages you want to appear in the coverage report.

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

- 4 Select **OK** to close the dialog box.
- 5 Select **Start** from the **Simulation** menu or the **Start** button on the Simulink toolbar to start simulation of the model.

By default, Simulink saves coverage data for the current run in the workspace object `covdata` and cumulative coverage data in `covCumulativeData`. This data appears in an HTML report that is displayed automatically at the end of simulation.

Note You cannot run simulations with both model coverage reporting and acceleration options enabled. Simulink disables model coverage reporting if the accelerator is enabled.

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

Specifying Model Coverage Reporting Options

In this section...

“The Coverage Settings Dialog Box” on page 5-12

“Coverage Tab” on page 5-13

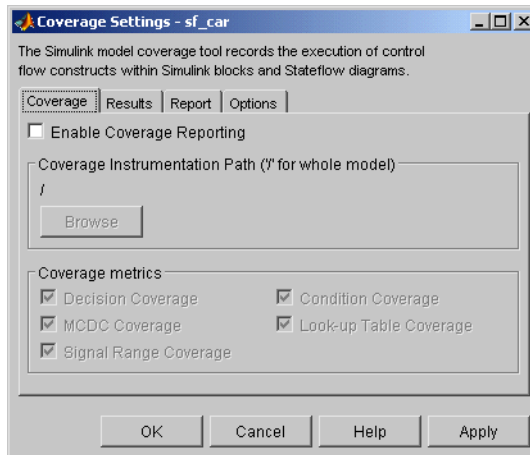
“Results Tab” on page 5-14

“Report Tab” on page 5-15

“Options Tab” on page 5-19

The Coverage Settings Dialog Box

A large part of using model coverage is specifying model coverage reporting options in the Coverage Settings dialog box. You open this dialog from the **Tools** menu of a Simulink window by selecting **Coverage Settings**. The Coverage Settings dialog box appears with its **Coverage** tab in focus, as shown.



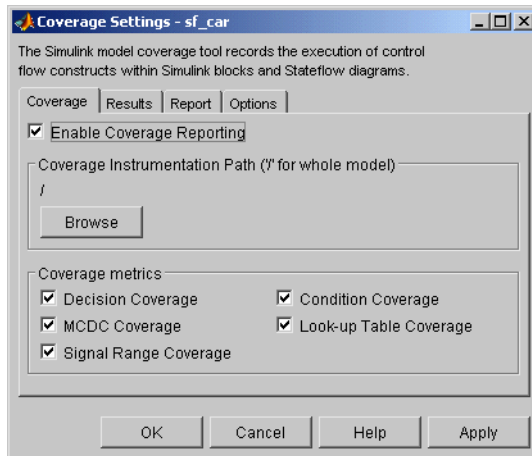
The sections that follow describe the settings for each tab of the Coverage Settings dialog box.

Coverage Tab

You select the model coverages calculated during simulation in the fields of the **Coverage** tab of the Coverage Settings dialog box.

Enable Coverage Reporting

Causes Simulink to gather and report the specified model coverages during simulation. When you select the **Enable Coverage Reporting** field, all other fields in the **Coverage** page of the Coverage Settings dialog box are enabled, as shown.



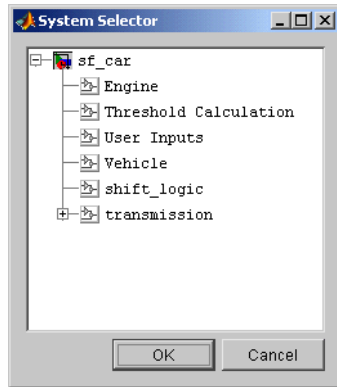
Coverage Instrumentation Path

Specifies path of the subsystem for which Simulink gathers and reports coverage data. By default, Simulink generates coverage data for the entire model.

To restrict coverage reporting to a particular subsystem,

- 1 In the **Coverage** page of the Coverage Settings dialog box, click **Browse**.

Simulink displays a System Selector dialog box.



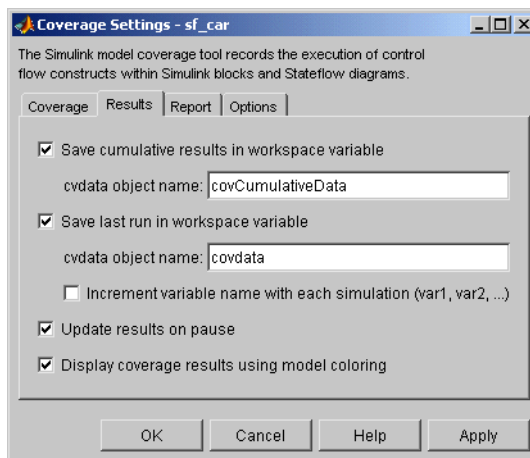
- 2 Select the subsystem for which you want coverage reporting to be enabled and click **OK** to close the dialog box.

Coverage Metrics

Select the types of test case coverage analysis that you want the tool to perform. See “Types of Model Coverage” on page 5-3 for more information.

Results Tab

You select the destination of model coverage results from model coverage in the **Results** page of the Coverage Settings dialog box.



Save Cumulative Results in Workspace Variable

Causes model coverage to accumulate and save the results of successive simulations in the workspace variable specified in the **cvdata object name** field below. The coverage running total in the workspace variable is updated with new results at the end of each simulation.

Save Last Run in Workspace Variable

Causes model coverage to save the results of the last simulation run in the workspace variable specified in the **cvdata object name** field below.

Increment Variable Name with Each Simulation

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

Update Results on Pause

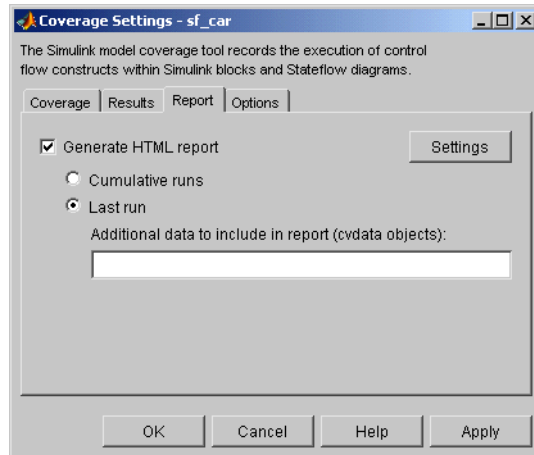
When you pause during simulation the first time, causes the HTML model coverage report to appear with model coverage results recorded up to the pause point. When you resume simulation and later pause or stop simulation, the model coverage report reappears in updated form with coverage results up to the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

Display Coverage Results Using Model Coloring

After simulation, causes coloring of Simulink blocks according to their level of model coverage. Blocks highlighted in light green received full coverage during testing. Blocks highlighted in light red received incomplete coverage. In addition, model coverage results for each block receiving it is available in context-sensitive form. See “Colored Simulink Diagram Coverage Display” on page 5-36 for a complete description.

Report Tab

You select the model coverage test sessions (runs) reported by model coverage in the **Report** page of the Coverage Settings dialog box.

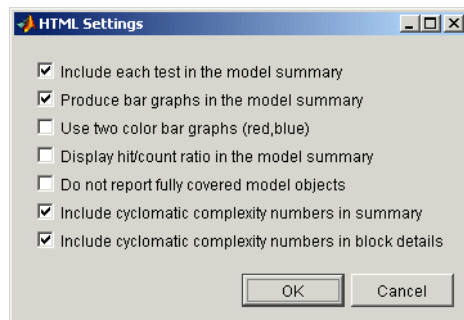


Generate HTML Report

Causes Simulink to create an HTML report containing the coverage data. Simulink displays the report in the MATLAB Help browser at the end of the simulation. Click the **Settings** button to select various reporting options (see “Settings” on page 5-16).

Settings

The HTML Settings dialog box allows you to choose various model coverage report options. To display the dialog box, click **Settings** on the **Report** page of the Coverage Settings dialog box. The HTML Settings dialog box appears.



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

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

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

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

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

Include cyclomatic complexity numbers in summary. Includes the cyclomatic complexity (see “Types of Model Coverage” on page 5-3) of the model and its top-level subsystems and charts in the report summary. A cyclomatic complexity number shown in boldface indicates that the analysis considered the subsystem itself to be an object when computing its complexity. This occurs for atomic and conditionally executed subsystems as well as Stateflow blocks.

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

Cumulative Runs

Display the coverage results from successive simulations in the report.

Whenever the check box **Save cumulative results in workspace variable** in the **Results** page is selected, a coverage running total is updated with new results at the end of each simulation. In a cumulative coverage report the results in the rightmost column reflect that 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.

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

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

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

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

Calculating cumulative coverage results is also possible at the command line via the `+` operator. The following script demonstrates this usage:

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

Last Run

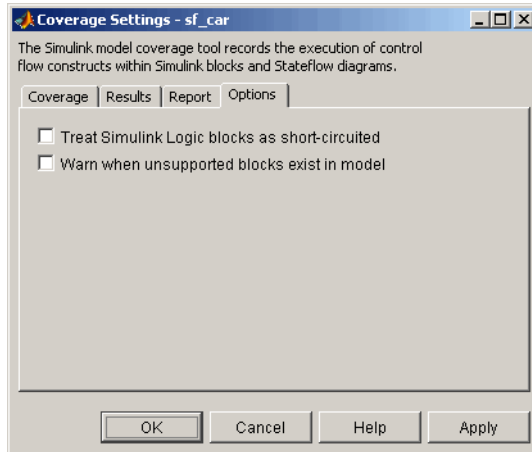
Display only the results of the most recent simulation run in the report.

Additional Data to Include in Report

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

Options Tab

You select important options for model coverage reports in the **Options** page of the Coverage Settings dialog box.



Treat Simulink Logic Blocks as Short-Circuited

Applies only to condition and MC/DC coverage. If enabled, coverage analysis treats Simulink logic blocks as though they short-circuit their input. In other words, Simulink treats such a block as if the block ignores remaining inputs if the previous inputs alone determine the block's output. For example, if the first input to an And block is false, MC/DC coverage analysis ignores the values of the other inputs in determining MC/DC coverage for a test case.

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

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

Warn When Unsupported Blocks Exist in a Model

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

Understanding Model Coverage Reports

In this section...
“About Model Coverage Reports” on page 5-21
“Summary Report Section” on page 5-21
“Details Report Section” on page 5-22
“Decisions Analyzed Table” on page 5-24
“Conditions Analyzed Table” on page 5-24
“MC/DC Analysis Table” on page 5-25

About Model Coverage Reports

The coverage report generated by model coverage contains several parts, each of which is described in the sections that follow.

For an understanding of model coverage reports for Stateflow diagrams and their objects, see “Understanding Model Coverage for Stateflow Charts” in the Stateflow documentation.

Summary Report Section

The coverage summary section has two subsections: Tests and Summary.

Coverage Report for fuelsys










Tests

Test 1

Started Execution: 05-Apr-2001 15:51:41

Ended Execution: 05-Apr-2001 15:52:08

Summary

Model Hierarchy:	Test 1			
	D1	C1	MCDC	TBL
1. fuelsys	39% 	34% 	13% 	1%
2. . . . E&O_sensor	50% 	NA	NA	NA
3. . . . MAP_sensor	50% 	NA	NA	NA
4. . . . engine_speed	50% 	NA	NA	NA
5. . . . engine_gas_dynamics	60% 	NA	NA	NA
6. Mixing & Combustion	50% 	NA	NA	NA
7. Throttle & Manifold	63% 	NA	NA	NA

The Tests section lists 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 the test case label, for example “Test throttle,” specified using the `cvtest` command.

The Summary section summarizes the results for each subsystem. Clicking the name of the subsystem takes you to a detailed report for that subsystem.

Details Report Section

The Details section reports the model coverage results in detail.

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	86
Decision (D1)	NA	38% (53/140) 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 intervals

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

Switch block "[Switch](#)"

Parent: [fuelsys/EGO sensor](#)

Uncovered Links: 

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

Decisions analyzed:

logical trigger input	50%
false (output is from 3rd input port)	0/17940
true (output is from 1st input port)	17940/17940

The Details section starts with a summary of results for the model as a whole followed by a list of subsystems and charts that the model contains. Subsections on each subsystem and chart follow. Clicking the name of a subsystem or chart in the model summary takes you to a detailed report on that subsystem or chart. The section for each subsystem starts with a summary of the test coverage results for the subsystem and a list of the subsystems that it contains. The overview is followed by block reports, one for each block that contains a decision point in the subsystem.

Each section of the detailed report summarizes the results for the metrics used to test the object (model, subsystem, chart, block) to which the section

applies. The sections for models and subsystems give results for the model and subsystem considered as a covered object and for the contents of the model or subsystem.

You can also access an individual object's subsection of the Details section from the Simulink model as follows:

- 1 Right-click a Simulink block

A pop-up menu appears.

- 2 In the pop-up menu, select **Coverage**, and from the resulting pop-up submenu, select **Report**.

The model coverage report appears, scrolled to the applicable Details subsection.

Each section can include coverage results for more than one simulation run. The report displays the results for each simulation run in a separate column. A numeric prefix in the column heading indicates the run that produced the data.

Decisions Analyzed Table

This table lists possible outcomes for a decision and the number of times that an outcome occurred in each test simulation.

Decisions analyzed:

logical trigger input	50%
false (output is from 3rd input port)	0/17940
true (output is from 1st input port)	17940/17940

The report highlights outcomes that did not occur in red. Clicking the block name causes Simulink to display the block diagram containing the block. Simulink also highlights the block to help you find it in the diagram.

Conditions Analyzed Table

This table lists the number of occurrences of true and false conditions on each input port of a block.

Conditions analyzed:

Description:	#1 T	#1 F
input port 1	481	17060
input port 2	0	17541

MC/DC Analysis Table

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

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

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

Each row of the 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 such that changing the value of input *n* alone is sufficient to change the value of the block's output. Input *n* is called the *deciding input* of the condition case.

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

- The position of a character in the string corresponds to the input port number.
- The character at the position represents the value of the input (T means true, F means false).
- Boldfacing a character indicates that it corresponds to the value of the deciding input.

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

The table's **Decision/Condition** column specifies the deciding input for an input condition case. The **#1 True Out** column specifies the deciding input value that causes the block to output a true value for a condition case. The

#1 True Out entry uses a condition case expression, for example, **FF**, to express the values of all the inputs to the block, with the value of the deciding variable indicated by boldfacing.

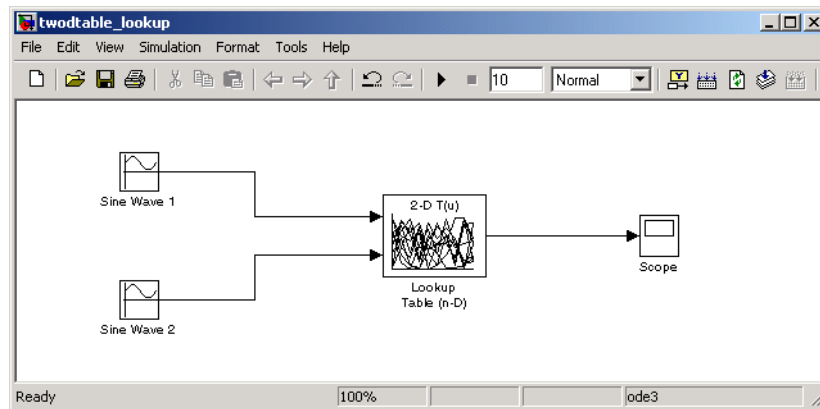
Parentheses around the expression indicate that the specified combination of inputs did not occur during the first (or only) test case included in this report. In other words, the test case did not cover the corresponding condition case. The **#1 False Out** column specifies the deciding input value 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. The report adds additional **#n True Out** and **#n False out** columns for additional test cases, where n is the number of the test case.

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

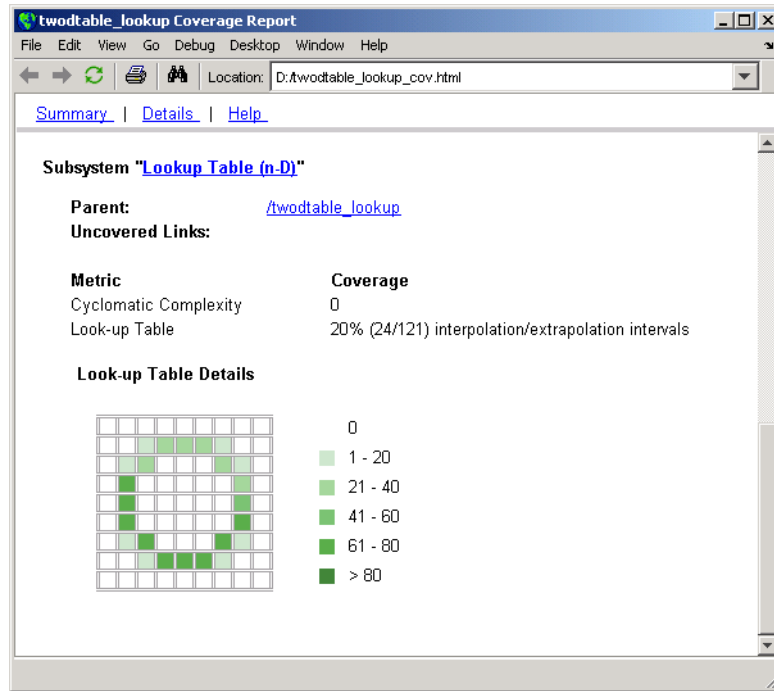
Navigation Arrows. The section for each block contains a backward and a forward arrow. Clicking the forward arrow takes you to the next section in the report that lists an uncovered outcome. Clicking the back arrow takes you back to the previous uncovered outcome in the report.

N-Dimensional Lookup Table Report

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



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

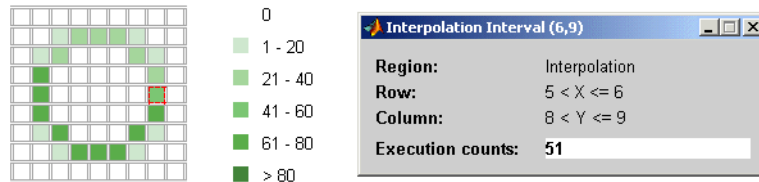


The report contains a two-dimensional table representing the elements of the Lookup Table. The element indices are represented by the cell border grid lines, which number 10 in each dimension. Areas where the Lookup Table interpolates between table values are represented by the cell areas. Areas of extrapolation 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 shading and the range of execution counts represented are displayed on the side of the table.

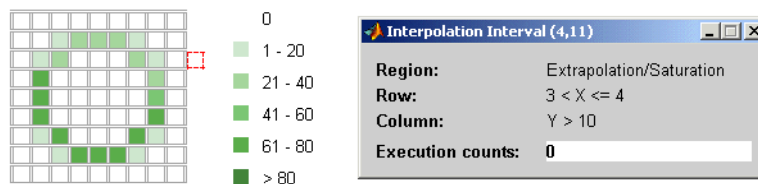
If you click an individual table cell, you receive a dialog that displays the index location of the cell and the exact number of execution counts generated for it during testing. The following example shows the contents of a color shaded cell on the right edge of the circle:

Lookup Table Details



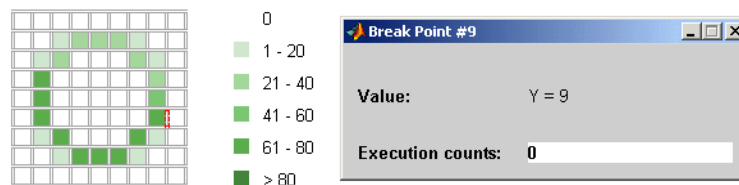
Notice that the selected cell is outlined in red. You can also click on extrapolation cells on the edge of the table, as shown.

Lookup Table Details

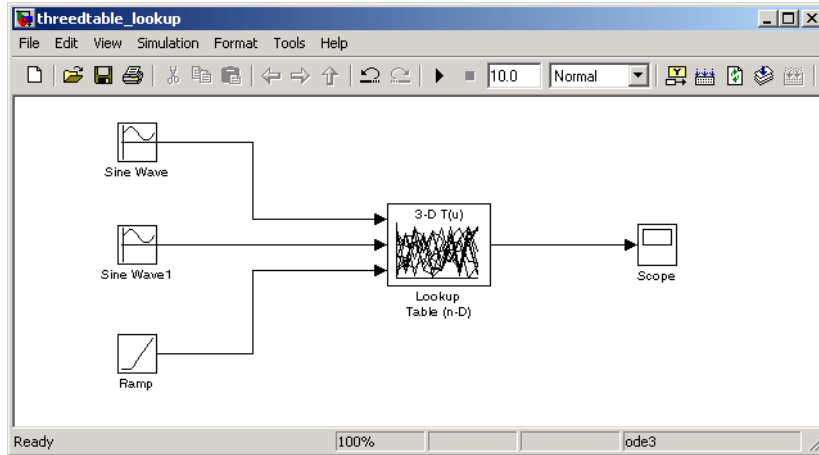


A bold grid line indicates that at least one block input equal to its exact index value occurred during the simulation. Click the border to display the exact number of hits for that index value, as shown in the following example:

Lookup Table Details



The following example model uses a Lookup Table of 10-by-10-by-5 elements filled with random values.



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

After simulation, the following Lookup Table report appears:

Subsystem "**Lookup Table (n-D)**"

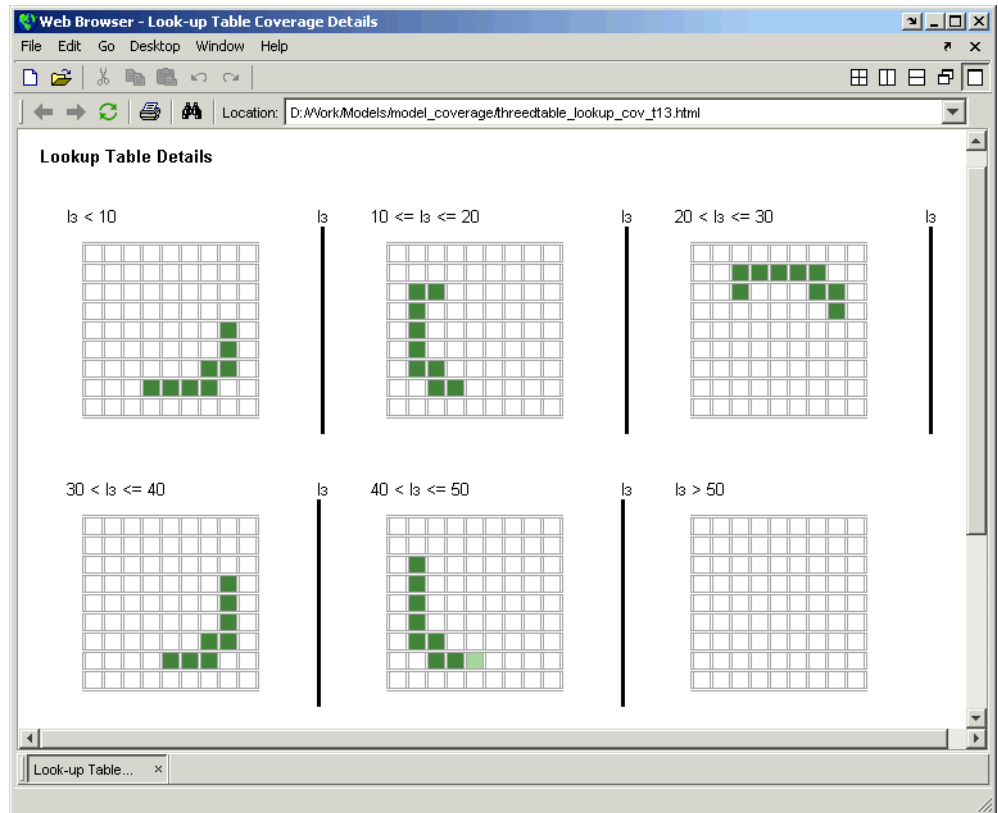
Parent: [/threedtable_lookup](#)

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 appears, which displays the following tables:



Notice that Lookup Table coverage for a three-dimensional Lookup Table block is reported as a set of two-dimensional tables. If you overlay these tables last on top of first, you notice that the coverage values corkscrew up to the reader.

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

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

Signal Range Analysis Report

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

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

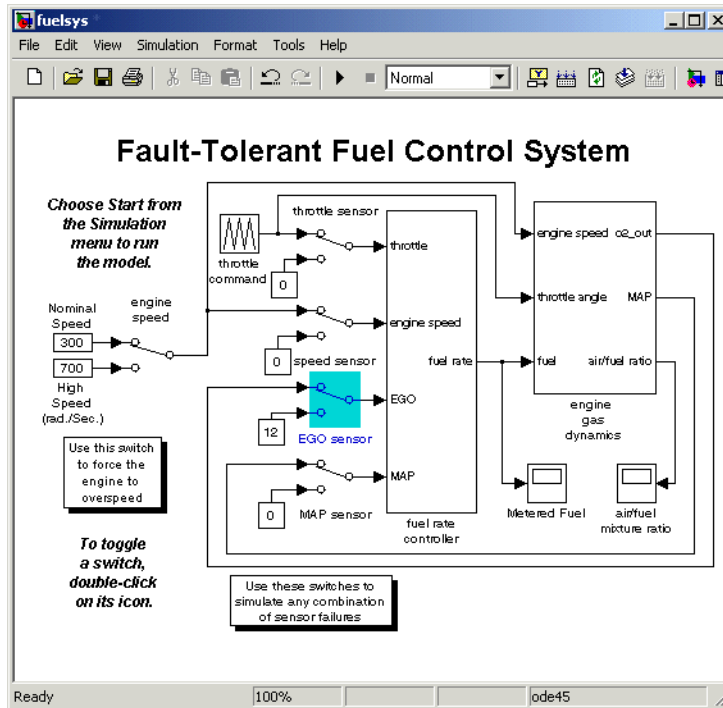
You can access the Signal Range Analysis report quickly with the **Signal Ranges** link in the nonscrolling region at the top of the model coverage report, as shown for the fuelsys model.

The screenshot shows a web browser window titled "Web Browser - fuelsys Coverage Report". The address bar displays "D:\Work\Models\fuelsys_cov.html". The page content is organized into three tabs: "Summary", "Details", and "Signal Ranges". The "Signal Ranges" tab is active, showing a table with the following data:

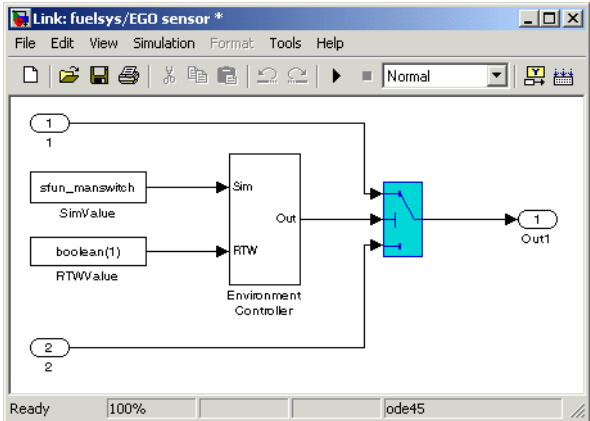
Hierarchy	Min value	Max value
fuelsys		
... Constant2	0	0
... Constant3	12	12
... Constant4	0	0
... Constant5	0	0
... High Speed (rad./Sec.)	700	700
... Nominal Speed	300	300
... EGO sensor		
..... Switch	0.180421	12
..... RTWValue	0	1

Link to Signal Range Coverage report

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



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



Colored Simulink Diagram Coverage Display

In this section...

- “How Model Coverage Highlighting Works” on page 5-36
- “Enabling the Colored Diagram Display” on page 5-36
- “Displaying Model Coverage with Model Coloring” on page 5-37
- “Accessing Coverage Information for Colored Blocks” on page 5-39

How Model Coverage Highlighting Works

Simulink displays model coverage results for individual blocks directly in Simulink diagrams. If you enable model coverage, Simulink does the following:

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

Coloring is used to highlight structural coverage in Simulink models. When you enable coloring for model coverage results, the tool highlights blocks that received the following types of model coverage:

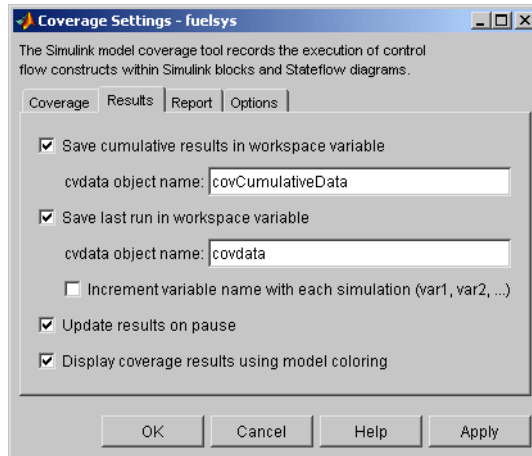
- “Decision Coverage (DC)” on page 5-4
- “Condition Coverage (CC)” on page 5-4
- “Modified Condition/Decision Coverage (MC/DC)” on page 5-5

Enabling the Colored Diagram Display

You enable the model coverage colored diagram display as follows:

- 1** In the Simulink window, from the **Tools** menu, select **Coverage Settings**.
The Coverage Settings dialog box appears.
- 2** In the **Coverage** tab of the Coverage Settings dialog box, select **Enable Coverage Reporting**.

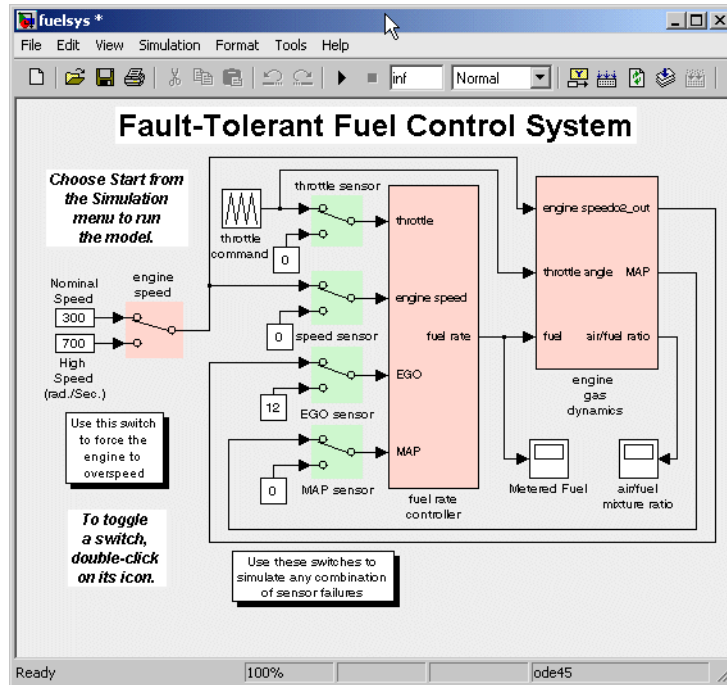
3 Select the **Results** tab, as shown.



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

Displaying Model Coverage with Model Coloring

You enable display coverage as described in “Enabling the Colored Diagram Display” on page 5-36. After you enable this display, any time that the model generates a model coverage report, individual blocks receiving coverage are highlighted with light green or light red.



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

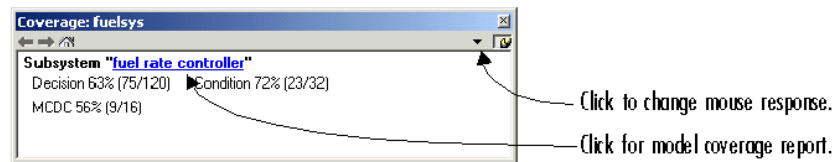
Note To restore the Simulink model diagram to its original colors, right-click a colored block and select **Coverage** from the resulting context menu followed by **Remove information** from the resulting submenu. Alternatively, you can select **Remove Highlighting** from the Simulink **View** menu or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

Accessing Coverage Information for Colored Blocks

“Displaying Model Coverage with Model Coloring” on page 5-37 describes the highlighted Simulink display that appears after simulation when you enable display coverage with model coloring in the coverage settings for the model. Along with the highlighted Simulink display, a Coverage Display window appears, as shown.



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



Summary coverage information appears in the Coverage Display window for the block, whose hyperlinked name appears at the top of the window. Click the hyperlink to access the appropriate section of the coverage report for this block. You can also see this section of the report by right-clicking the block and selecting **Coverage > Report**.

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

- Select the down arrow on the right side of the Coverage Display window, and, from the resulting menu, select **Focus**.

- Right-click a colored block and select **Coverage** from the resulting context menu followed by **Display details on mouse-over** from the resulting submenu.

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

Using Model Coverage Commands

In this section...

“About Model Coverage Commands” on page 5-41

“Creating Tests with `cvtest`” on page 5-41

“Running Tests with `cvsim`” on page 5-43

“Producing HTML Reports with `cvhtml`” on page 5-44

“Saving Test Runs to a File with `cvsave`” on page 5-44

“Loading Stored Coverage Test Results with `cvload`” on page 5-45

“Coverage Script Example” on page 5-46

About Model Coverage Commands

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

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

Creating Tests with `cvtest`

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

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

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

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

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

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

The following command creates a test with a setup command:

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

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

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

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

Running Tests with `cvsim`

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

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

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

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

This command executes the `cvtest` object `cvto` by starting a simulation run for the corresponding model. The results are returned in the `cvdata` object `cvdo`.

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

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

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

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

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

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

You can execute multiple test objects with the `cvsim` command. The following command executes a set of coverage test objects, `cvto1`, `cvto2`, ... and returns the results in a set of `cvdata` objects, `cdvo1`, `cvdo2`,

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

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

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

Producing HTML Reports with cvhtml

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

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

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

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

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

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

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

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

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

Saving Test Runs to a File with cvsave

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

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

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

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

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

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

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

Loading Stored Coverage Test Results with `cvload`

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

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

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

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

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

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

`cvload` Special Considerations

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

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

Coverage Script Example

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

```
mdl = 'slvndemo_ratelim_harness';

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

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

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

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

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

Model Coverage for Referenced Models

In this section...

“Introduction” on page 5-47

“Creating a Test Group with `cv.cvtestgroup`” on page 5-50

“Running Tests with `cvsimref`” on page 5-50

“Extracting Results from `cv.cvatagroup`” on page 5-51

Introduction

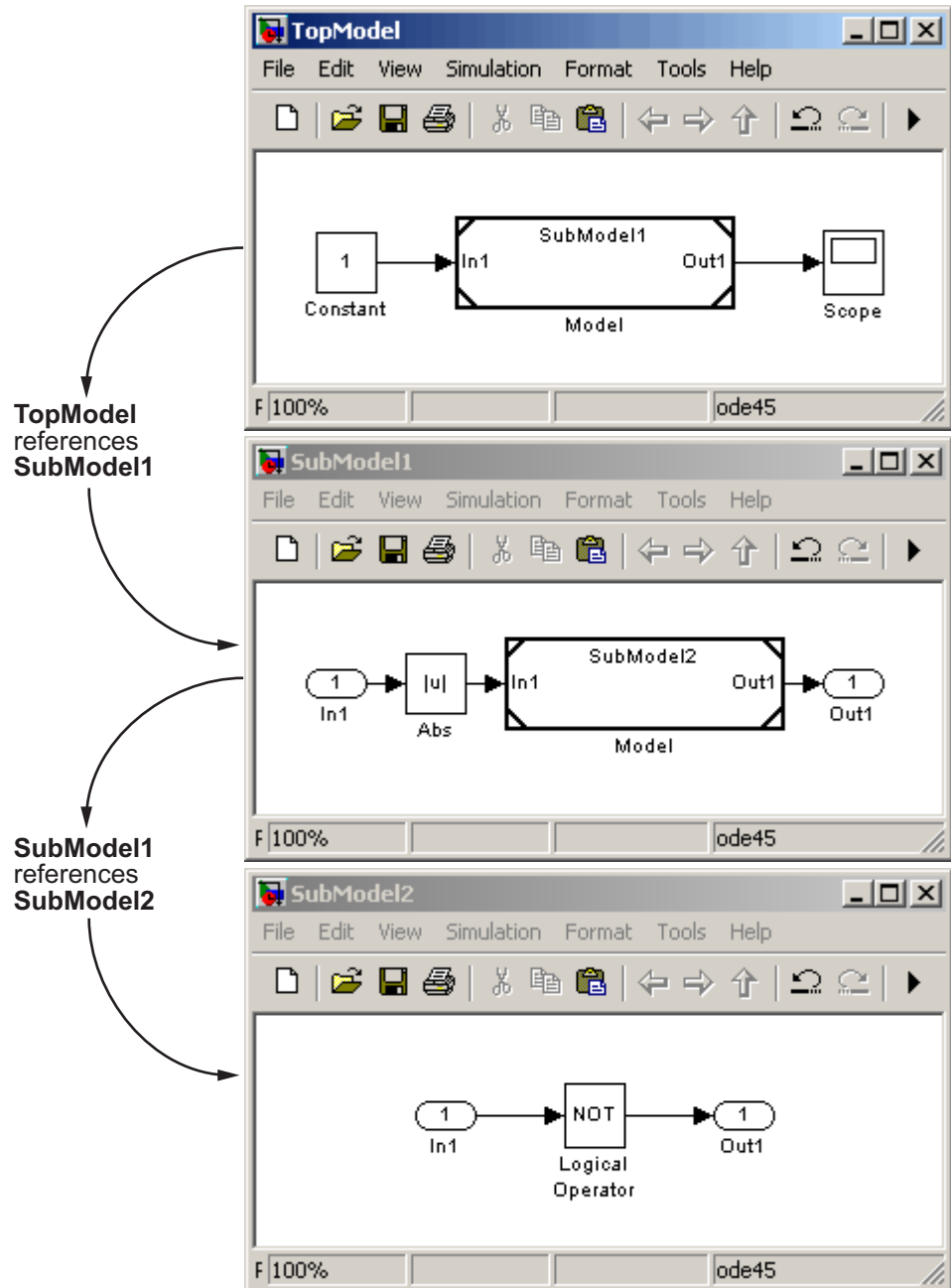
Simulink allows you to include one model in another by using Model blocks. Each Model block represents a reference to another model, called a *referenced model* or *submodel*. A referenced model itself can contain Model blocks that reference other models. You can construct a hierarchy of referenced models,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

Model coverage supports referenced models that operate in Normal mode. That is, you can record coverage only for those Model blocks whose **Simulation mode** parameter specifies Normal. However, you must use model coverage commands to record coverage for referenced models. The following steps describe a basic workflow for using commands to obtain model coverage results for Model blocks:

Step	Description	See...
1	Use <code>cv.cvtestgroup</code> to group together test specification objects that correspond to each model in a hierarchy.	“Creating a Test Group with <code>cv.cvtestgroup</code> ” on page 5-50

Step	Description	See...
2	Use <code>cvsimref</code> to simulate the top model in a hierarchy and record coverage results for its referenced models.	“Running Tests with <code>cvsimref</code> ” on page 5-50
3	Use <code>cv.cvdatagroup</code> to extract the coverage data objects that correspond to each model in a hierarchy.	“Extracting Results from <code>cv.cvdatagroup</code> ” on page 5-51

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



Creating a Test Group with `cv.cvtestgroup`

The `cvtest` command creates a test specification object for a Simulink model (see “Creating Tests with `cvtest`” on page 5-41). But if your model references other models, you might use a different test specification object for each model in the hierarchy. In this case, the `cv.cvtestgroup` object allows you to group together multiple test specification objects. After you create a group of test specification objects, you simulate it using the `cvsimref` function.

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

```
cvto1 = cvtest('TopModel1')
cvto2 = cvtest('SubModel11')
cvto3 = cvtest('SubModel12')
```

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

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

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

Running Tests with `cvsimref`

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

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

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

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

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

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

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

```
[cvdgd,t,x,y] = cvsimref(topModelName, cvtgd)
```

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

```
[cvdgd,t,x,y] = cvsimref(topModelName, cvtgd, timespan, options)
```

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

Extracting Results from `cv.cvdatagroup`

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

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

```
modelName = cvdgd.allNames
```

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

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

After you extract the individual `cvdata` objects, you can use other model coverage commands to operate on the coverage data of a particular model. For example, you can use the `cvhtml` function to create and display an HTML report of the coverage results (see “Producing HTML Reports with `cvhtml`” on page 5-44).

Model Coverage for Embedded MATLAB Function Blocks

In this section...
“Types of Model Coverage in Embedded MATLAB Function Blocks” on page 5-52
“Creating a Model with Embedded MATLAB Function Block Decisions” on page 5-53
“Understanding Embedded MATLAB Function Block Model Coverage” on page 5-56

Types of Model Coverage in Embedded MATLAB Function Blocks

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

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

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

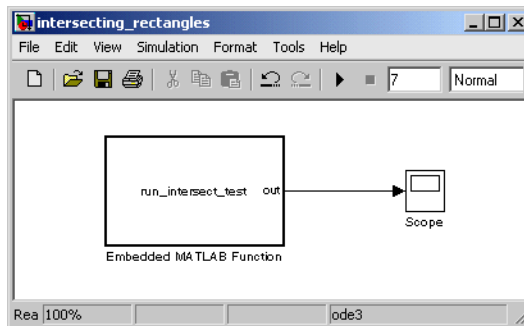
- Function header — Decision coverage is 100% if the function or subfunction is executed.
- `if` — Decision coverage is 100% if the `if` expression evaluates to true at least once, and false at least once.
- `switch` — Decision coverage is 100% if every switch case is taken, including the fall-through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

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

- if statement conditions
- while statement conditions, if present

Creating a Model with Embedded MATLAB Function Block Decisions

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



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

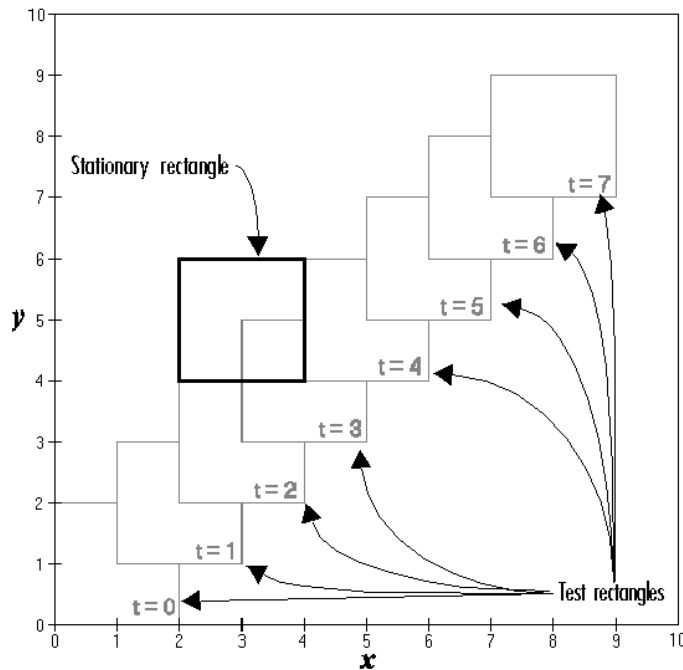
```

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  x1 = x1+1;
6  y1 = y1+1;
7  out = rect_intersect([x1 y1 2 2],[2 4 2 2]);
8
9  function out = rect_intersect(rect1,rect2)
10 % Return 1 if two rectangle arguments intersect; 0 if not.
11
12 left1 = rect1(1);
13 bottom1 = rect1(2);
14 right1 = left1 + rect1(3);
15 top1 = bottom1 + rect1(4);
16
17 left2 = rect2(1);
18 bottom2 = rect2(2);
19 right2 = left2 + rect2(3);
20 top2 = bottom2 + rect2(4);
21
22 if (top1 < bottom2 || top2 < bottom1)
23     out = 0;
24 else
25     if (right1 < left2 || right2 < left1)
26         out = 0;
27     else
28         out = 1;
29     end
30 end

```

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

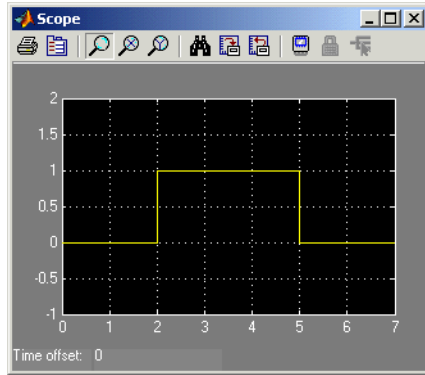
The coordinates for the origin of the moving test rectangle are represented by local data `x1` and `y1`, which are both initialized to -1. For the first sample, `x1` and `y1` are both incremented to 0. From then on, the progression of rectangle arguments during simulation is as follows:



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

The subfunction `rect_intersect` checks to see if its two rectangle arguments intersect. Each argument consists of coordinates for the lower left corner of the rectangle (origin), and its width and height. x values for the left and right sides and y values for the top and bottom are calculated for each rectangle and compared in nested `if-else` decisions. The function returns a logical value of 1 if the rectangles intersect and 0 if they do not.

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



Understanding Embedded MATLAB Function Block Model Coverage

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

When simulation is finished, the model coverage report appears in a browser window. After the summary for the model, the Details section of the model coverage report reports on each of the parts of the model. Model coverage for the parts of the example model in “Creating a Model with Embedded MATLAB Function Block Decisions” on page 5-53 appears in the following model-block-function order.

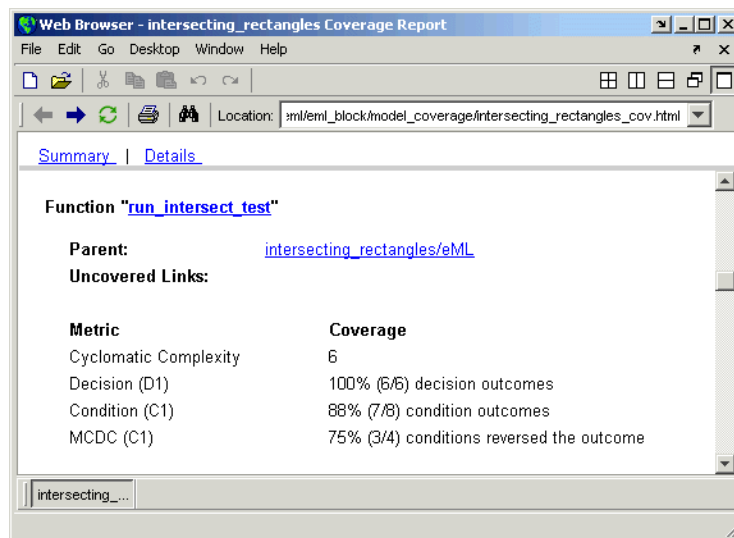
Model:	intersecting_rectangles
Block:	Embedded MATLAB Function
Function:	run_intersect_test
Decision Lines:	1: function out = rect_intersect_test
	9: function out = rect_intersect(rect1, rect2)

	22: if (top1 < bottom2 top2 < bottom1)
	25: if (right1 < left2 right2 < left1)

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

Model Coverage for the Embedded MATLAB Function Block Function `run_intersect_test`

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



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

```

1 function out = run_intersect_test
2 % Call rect_intersect to see if two rectangles: to see if
3 % a test rectangle and a stationary rectangle intersect
4
5 x1 = x1+1;
6 y1 = y1+1;
7 out = rect_intersect([x1 y1 2 2]',[2 4 2 2]');
8
9 function out = rect_intersect(rect1,rect2)
10 % Return 1 if two rectangle arguments intersect; 0 if not.
11
12 left1 = rect1(1);
13 bottom1 = rect1(2);
14 right1 = left1 + rect1(3);
15 top1 = bottom1 + rect1(4);
16
17 left2 = rect2(1);
18 bottom2 = rect2(2);
19 right2 = left2 + rect2(3);
20 top2 = bottom2 + rect2(4);
21
22 if (top1 < bottom2 || top2 < bottom1)
23     out = 0;
24 else
25     if (right1 < left2 || right2 < left1)
26         out = 0;
27     else
28         out = 1;
29     end
30 end

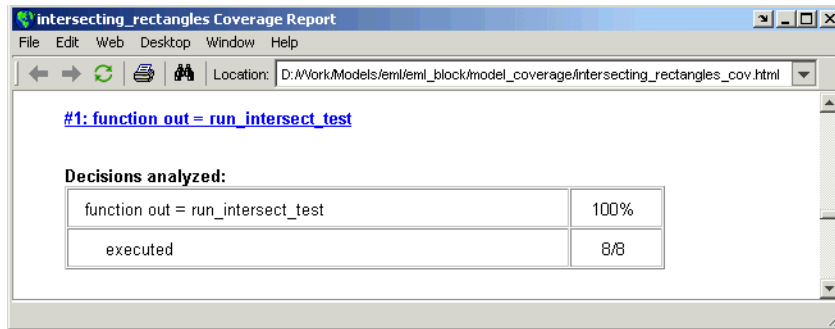
```

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 9 receives decision coverage on whether the subfunction `rect_intersect` is executed. Lines 22 and 25 receive decision, condition, and MCDC coverage for their `if` statements and conditions. Each of these lines is the subject of a report that follows the listing.

Notice that the condition `right1 < left2` in line 25 is highlighted in red. This means that this condition was not tested for all of its possible outcomes during simulation. Exactly which of the outcomes was not tested is answered by the report for the decision in line 25.

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

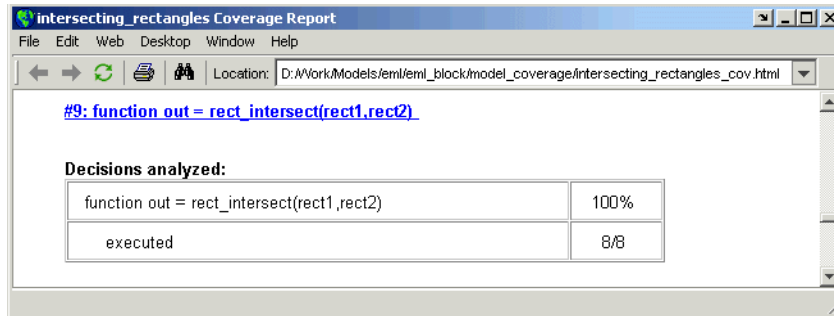
Coverage for Line 1. The coverage metrics for line 1 appear below the listing for the function `run_intersect_test`.



Decisions analyzed:	
function out = run_intersect_test	100%
executed	8/8

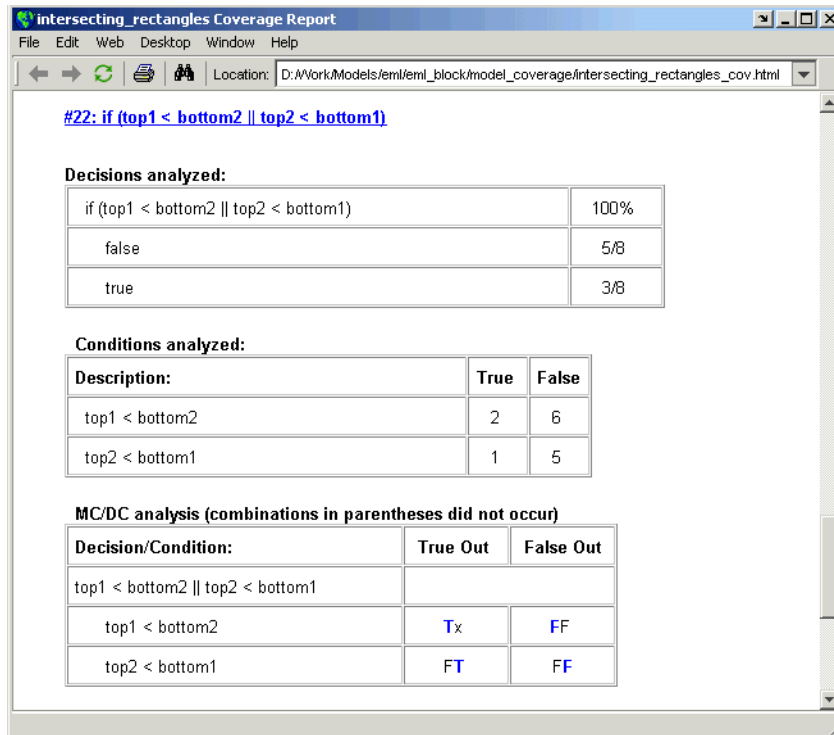
The first line of every function receives coverage analysis indicative of the decision to run the function in response to a call. Coverage for `run_intersect_test` indicates that it executed during testing.

Coverage for Line 9. The coverage metrics for line 9 appear below the coverage metrics for line 1.



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

Coverage for Line 22. Coverage metrics for line 22 appear below the coverage metrics for line 9.



The **Decisions analyzed** table indicates that there are two possible outcomes for the decision in line 22: true and false. Five of the eight times it was executed, the decision evaluated to false, and the remaining three times, it evaluated to true. Because both possible outcomes occurred, decision coverage is 100%.

The **Conditions analyzed** table sheds some additional light on the decision in line 22. Because this decision consists of two conditions linked by a logical OR (|) operation, only one condition must evaluate true for the decision to be true. If the first condition evaluates to true, there is no need to evaluate the second condition. The first condition, `top1 < bottom2`, was evaluated eight times, and was true twice. This means that it was necessary to evaluate the second condition only six times. In only one case was it true, which brings the total true occurrences for the decision to three, as reported in the **Decisions analyzed** table.

MCDC coverage looks for decision reversals that occur because one condition outcome changes from T to F or from F to T. The **MCDC 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 22.

Coverage for Line 25. Coverage metrics for line 25 appear below the coverage metrics for line 22.

#25: 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

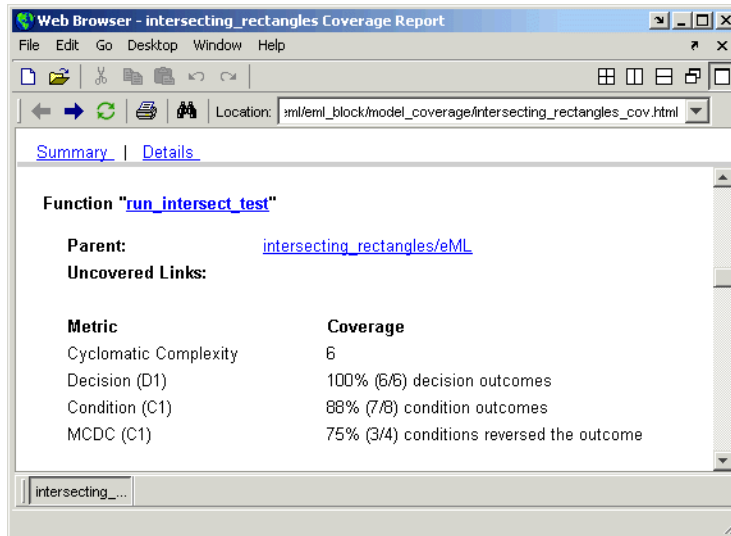
The line 25 decision, `if (right1 < left2 || right2 < left1)`, is nested in the `if` statement of the line 22 decision and is evaluated only if the line 22 decision is false. Because the line 22 decision evaluated false five times, line 25 is evaluated five times, three of which were false. Because both the true and false outcomes were achieved, decision coverage for line 25 is 100%.

Because line 25, like line 22, 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 25 decision does not test true, both outcomes did not occur for that condition and the condition coverage for the first condition is highlighted with a rose color. MCDC coverage is also

highlighted in the same way for a decision reversal based on the true outcome for that condition.

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



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

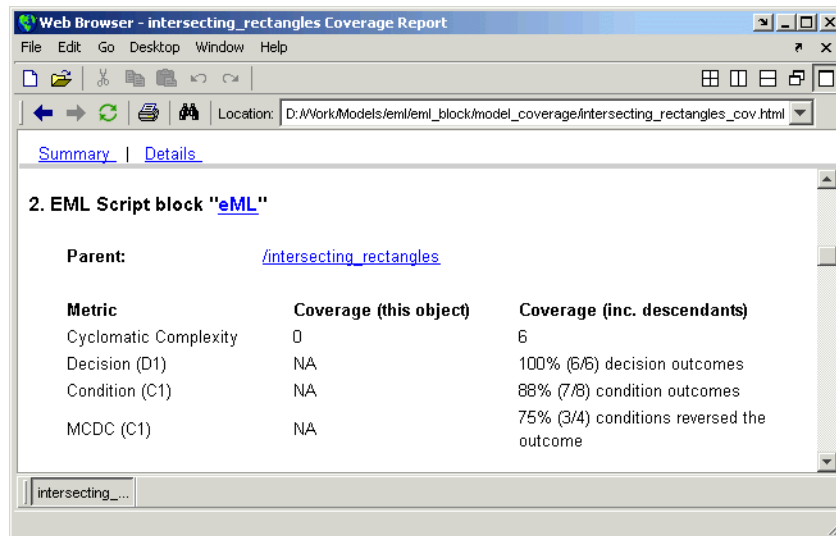
- There are six decision outcomes reported for run_intersect_test in the line reports: one for line 1 (executed), one for line 9 (executed), two for line 22 (true and false), and two for line 25 (true and false). The decision coverage for each line shows 100% decision coverage. This means that decision coverage for run_intersect_test is six of six possible outcomes, or 100%.
- There are four conditions reported for run_intersect_test in the line reports. Lines 22 and 25 each have two conditions, and each condition has two condition outcomes (true and false), for a total of eight condition outcomes in run_intersect_test. All conditions tested positive for both the true and false outcome except for the first condition of line 25 (right1

$< \text{left2}$). This means that condition coverage for `run_intersect_test` is seven of eight, or 88%.

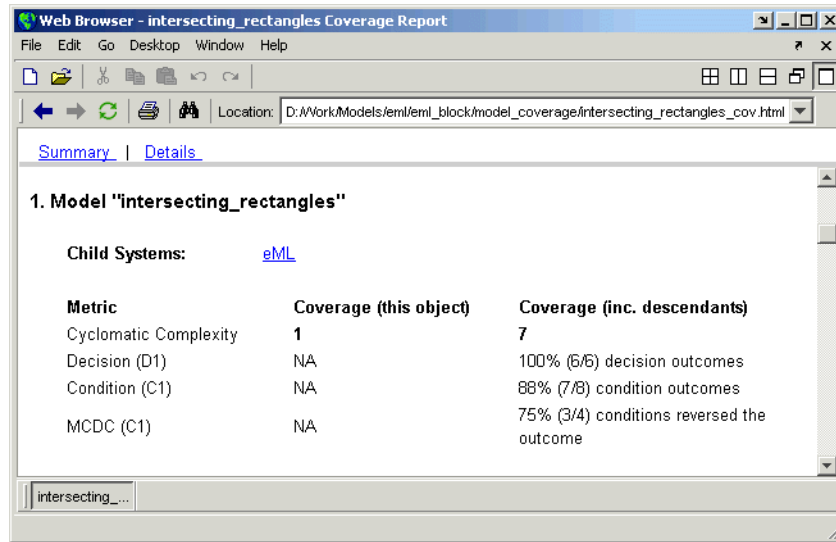
- The MCDC coverage tables for decision lines 22 and 25 each list two cases of decision reversal for each condition, for a total of four possible reversals. Only the decision reversal for a change in the evaluation of the condition $\text{right1} < \text{left2}$ of line 25 from true to false did not occur during simulation. This means that three of four, or 75% of the possible reversal cases were tested for during simulation, for a coverage of 75%.

Model Coverage for the Embedded MATLAB Function Block and the Model

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



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



Customizing Model Advisor

Model Advisor is a tool that runs a set of checks and tasks on a Simulink model or subsystem to uncover conditions and configuration settings that result in inaccurate or inefficient simulation or code generation. For more information about Model Advisor, see “Consulting Model Advisor” in the Simulink documentation.

Simulink Verification and Validation provides an API that allows you to customize the behavior of Model Advisor by defining your own custom tasks and checks, and writing your own callback functions. This chapter describes how to customize Model Advisor, covering the following topics:

The Customization Process (p. 6-3)	Describes how to create the customization file
Demo and Code Example (p. 6-4)	Describes how to run a demo that shows how to customize Model Advisor
Creating Callback Functions for Checks (p. 6-5)	Describes check callback functions and how to create them.
Defining Custom Checks (p. 6-13)	Describes the properties of custom checks and how to define them
Defining Custom Tasks (p. 6-19)	Describes the properties of custom tasks and how to define them
Defining a Process Callback Function (p. 6-22)	Describes process callback functions and how to create them

Formatting Model Advisor Outputs
(p. 6-25)

Describes the properties of custom formatted outputs and how to define them

Registering Custom Checks and
Tasks (p. 6-29)

Describes how to register custom checks and tasks in Model Advisor

The Customization Process

To customize Model Advisor, create an M-file called `sl_customization.m` and include this file on your MATLAB path. The M-file should contain a set of functions for registering and defining custom checks and tasks. Follow these guidelines:

Function	Description	When Required
<code>sl_customization()</code>	Registers custom checks and tasks with the Simulink customization manager at startup (see “Registering Custom Checks and Tasks” on page 6-29)	Required for all Model Advisor customizations
One or more check definition functions	Defines all custom checks (see “Defining Custom Checks” on page 6-13)	Required for custom checks
One or more task definition functions	Defines all custom tasks (see “Defining Custom Tasks” on page 6-19)	Required only for custom tasks
Check callback functions	Defines the actions of the custom checks (see “Creating Callback Functions for Checks” on page 6-5)	Required for custom checks. You must write one callback function for each custom check.
One process callback function	Specifies actions to be performed at different stages of Model Advisor execution (see “Defining a Process Callback Function” on page 6-22)	Optional

Demo and Code Example

Simulink Verification and Validation provides a demo that shows how to customize Model Advisor by adding three custom checks, a custom task for grouping the checks, and a process callback function. The demo also provides the source code of the `sl_customization.m` file that executes the customizations. The following sections present excerpts from this source code to illustrate how to write functions for customizing Model Advisor.

To run the demo:

- 1 Type `slvndemo_md1adv` at the MATLAB command line.
- 2 Follow the online instructions.

Creating Callback Functions for Checks

In this section...

“About Check Callback Functions” on page 6-5

“Simple Check Callback Function” on page 6-5

“Detailed Check Callback Function” on page 6-6

“Check Callback Function with Hyperlinked Results” on page 6-8

About Check Callback Functions

A callback function specifies the actions a check performs on a model or subsystem. You must create a callback function for each custom check so that Model Advisor can execute the function when the check is selected by a user. There are several styles of callback functions:

- “Simple Check Callback Function” on page 6-5
- “Detailed Check Callback Function” on page 6-6
- “Check Callback Function with Hyperlinked Results” on page 6-8

All styles of check callback functions provide one or more return arguments for displaying the results after executing the check. In some cases, return arguments are strings or cell arrays of strings that support embedded HTML tags for text formatting. It is recommended that you use the Model Advisor Formatting API to format your outputs, as described in “Formatting Model Advisor Outputs” on page 6-25, and limit HTML tags to be compatible with alternate output formats.

Simple Check Callback Function

Use the simple callback function to return a simple status, perhaps to indicate whether the model passed or failed the check, or to provide a recommendation for correcting an issue. The keyword for the simple callback function is `StyleOne`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-13).

The simple callback function takes the following arguments:

Argument	I/O Type	Description
system	Input	Path to the model or subsystem analyzed by Model Advisor.
result	Output	MATLAB string that supports Model Advisor Formatting API calls and embedded HTML tags for text formatting.

Here is an example of a simple callback function for a custom check that looks for models that do not use white as the background color for their Simulink windows:

```
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

if strcmp(get_param(bdroot(system), 'ScreenColor'), 'white')
    result = '<font color="#008000">Passed</font>';
    mdladvObj.setCheckResultStatus(true); % set to pass
else
    result = [...
        'It is recommended to select a Simulink window screen '...'
        'color of white to ensure a readable and printable model. '...'
        'Click <a href="matlab: set_param(bdroot, 'ScreenColor', 'white')">'...'
        'here</a> to change screen color to white.'];
    mdladvObj.setCheckResultStatus(false); % set to fail
end
```

Detailed Check Callback Function

Use the detailed check callback function to return and organize results as strings in a layered, hierarchical fashion. The function provides two output arguments that allow you to associate text descriptions with one or more paragraphs of detail. The keyword for the simple callback function is `StyleTwo`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-13).

The detailed callback function takes the following arguments:

Argument	I/O Type	Description
system	Input	Path to the model or system analyzed by Model Advisor.
ResultDescription	Output	Cell array of MATLAB strings that supports Model Advisor Formatting API calls and embedded HTML tags for text formatting. 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} = '<p>Check Simulation optimization settings:';
    if strcmp(get_param(model, 'BlockReduction'), 'off');
        ResultDetails{end+1} = {...
            'It is recommended to turn on Block reduction optimization option.'};
        mdladvObj.setCheckResultStatus(false); % set to fail
    else
        ResultDetails{end+1} = {'<font color="#008000">Passed</font>'};
    end
end
```

```

end

% Check code generation optimization setting
ResultDescription{end+1} = '<p>Check code generation optimization settings:';
ResultDetails{end+1} = {};
if strcmp(get_param(model,'LocalBlockOutputs'),'off');
    ResultDetails{end}{end+1} = ...
        'It is recommended to turn on Enable local block outputs option.<BR>';
    mdladvObj.setCheckResultStatus(false); % set to fail
end
if strcmp(get_param(model,'BufferReuse'),'off');
    ResultDetails{end}{end+1} = ...
        'It is recommended to turn on Reuse block outputs option.<BR>';
    mdladvObj.setCheckResultStatus(false); % set to fail
end
if isempty(ResultDetails{end})
    ResultDetails{end}{end+1} = '<font color="#008000">Passed</font>';
end
end

```

Check Callback Function with Hyperlinked Results

This callback function automatically displays hyperlinks for every object returned by the check to make it easy to locate problem areas in your model or subsystem. The keyword for this type of callback function is `StyleThree`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-13).

This callback function takes the following arguments:

Argument	I/O Type	Description
system	Input	Path to the model or system analyzed by Model Advisor.

Argument	I/O Type	Description
ResultDescription	Output	Cell array of MATLAB strings that supports Model Advisor Formatting API calls and 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.

Model Advisor automatically concatenates each string from ResultDescription with the corresponding array of objects from ResultDetails. 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, Model Advisor displays the target object highlighted in your Simulink model. Here 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.

```
function [ResultDescription, ResultDetails] = SampleStyleThreeCallback(system)
ResultDescription = {};
ResultDetails = {};

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object
mdladvObj.setCheckResultStatus(true); % init result status to pass

% find all blocks inside current system
allBlks = find_system(system);

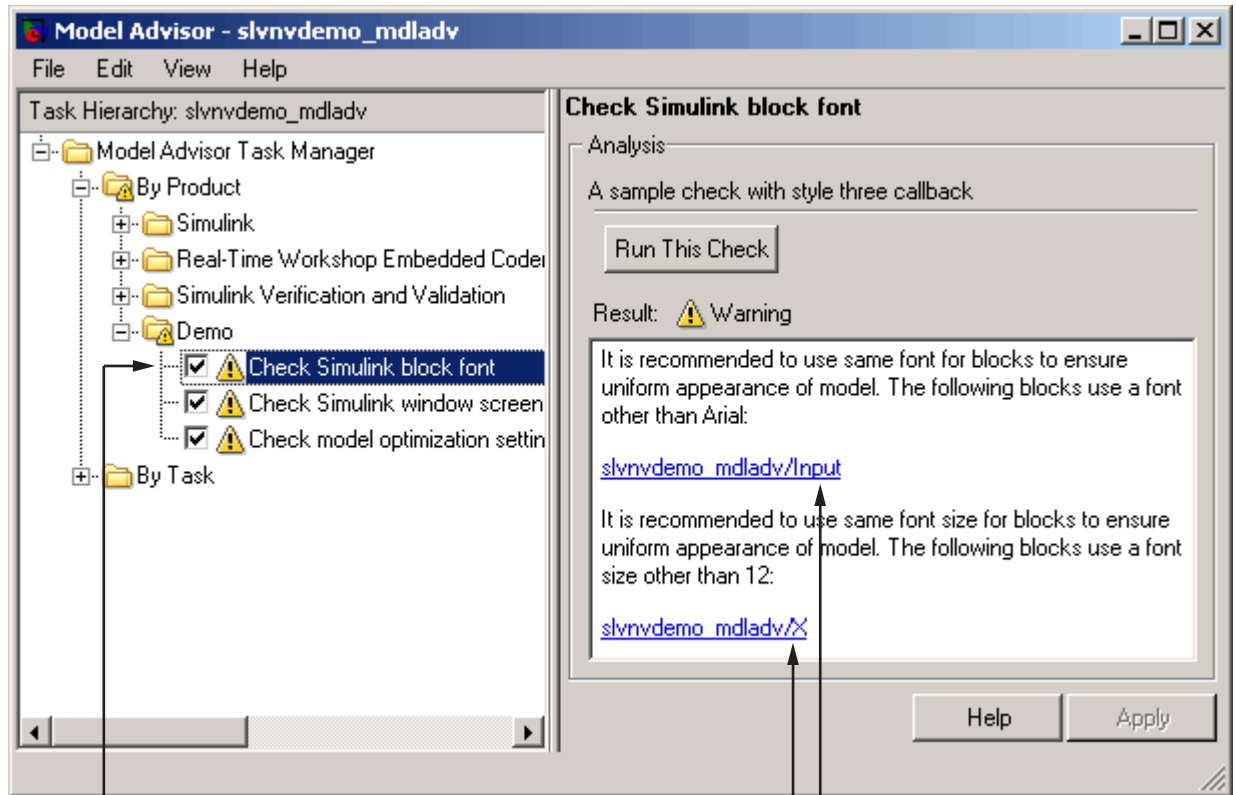
% block diagram doesn't have font property
% get blocks inside current system that have font property
allBlks = setdiff(allBlks, {system});
```

```
% find regular font name blocks
regularBlks = find_system(allBlks,'FontName','Arial');

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ['<p>It is recommended to use same font for ',...
        'blocks to ensure uniform appearance of model. '...
        'The following blocks use a font other than Arial: '];
    ResultDetails{end+1}      = searchResult;
    mdladvObj.setCheckResultStatus(false); % set to fail
else
    ResultDescription{end+1} = '<p>All block font names are identical.';
    ResultDetails{end+1}     = {};
end

% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',12);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = [...
        '<p>It is recommended to use same font size for blocks '...
        'to ensure uniform appearance of model. '...
        'The following blocks use a font size other than 12: '];
    ResultDetails{end+1}      = searchResult;
    mdladvObj.setCheckResultStatus(false); % set to fail
else
    ResultDescription{end+1} = '<p>All block font sizes are identical.';
    ResultDetails{end+1}     = {};
end
```

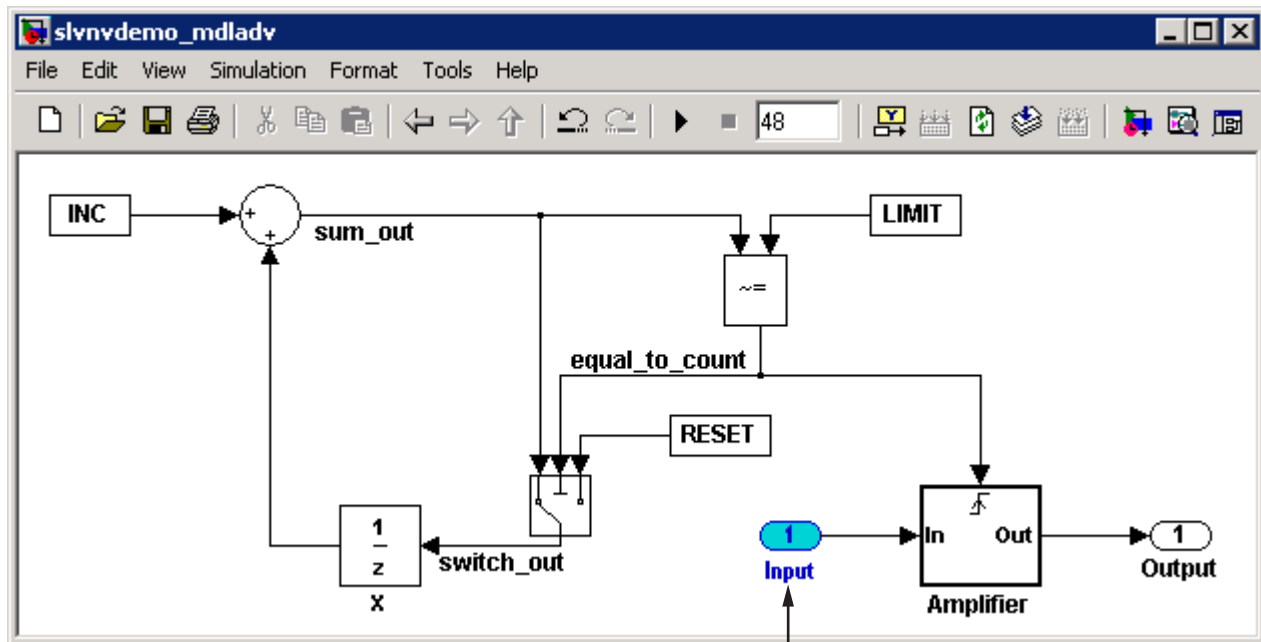
After running the check associated with this callback function, Model Advisor displays the results as follows:



Custom check selected

Result, contains hyperlinks to blocks in the model

Clicking the first hyperlink, to the input `slvndemo_md1adv/Input`, displays the Simulink model with Input highlighted, similar to the following:



Highlighted object

Defining Custom Checks

In this section...

“About Custom Checks” on page 6-13

“Properties of Model Advisor Checks” on page 6-13

“How Visible, Enable, and Value Properties Interact” on page 6-17

“Code Example: Check Definition Function” on page 6-17

About Custom Checks

You define custom checks in one or more functions that specify the properties of each instance of the class `Simulink.MdlAdvisorCheck`. You must define one instance of this class for each custom check you want to add to Model Advisor and register the custom check as described in “Registering Custom Checks and Tasks” on page 6-29. The sections that follow describe how to define custom checks.

Properties of Model Advisor Checks

The following table describes the properties of the `Simulink.MdlAdvisorCheck` class:

Property	Data Type	Default Value	Description
Title	String	'' (null string)	Name of the check as it should appear in Model Advisor.

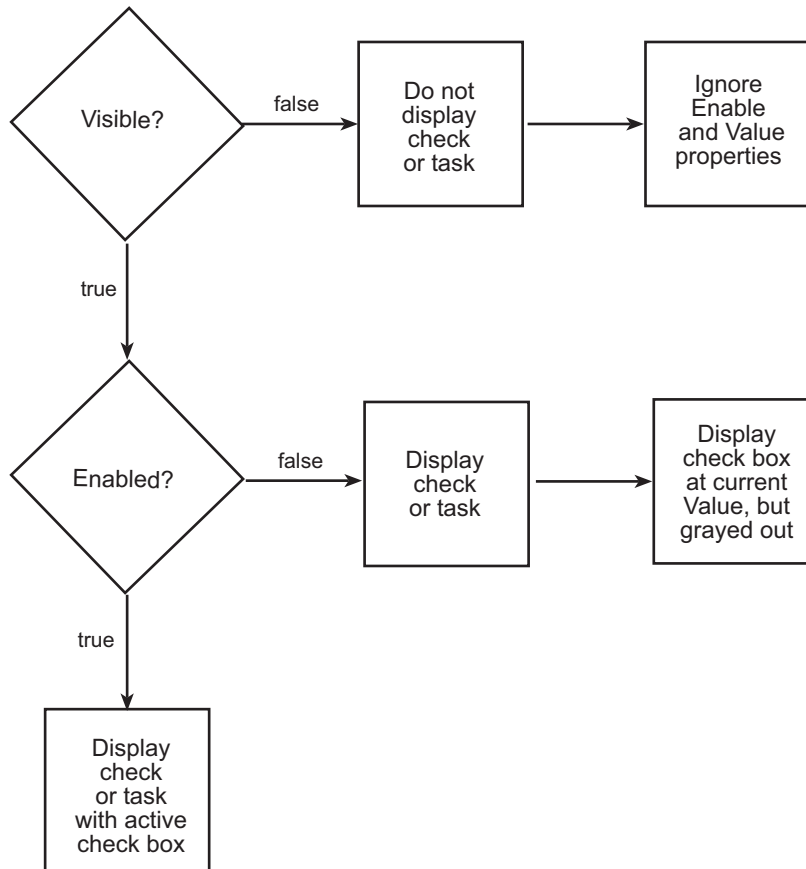
Property	Data Type	Default Value	Description
TitleID	String	' ' (null string)	Permanent, unique identifier for the check. Typically, TitleID is set to the check Title string, but the value of TitleID remains constant even if the Title of the check changes. Tasks should refer to checks by TitleID (the permanent identifier), not by Title.
TitleTips	String	' ' (null string)	Description of the check, which Model Advisor displays in its right pane when you view details about the check.
CallbackHandle	Function handle	[] (empty handle)	Handle to the callback function for the check.
CallbackContext	Enumeration	'None '	Context for checking the model or subsystem: <ul style="list-style-type: none">• None = No special requirements• PostCompile = Model must be compiled

Property	Data Type	Default Value	Description
CallbackStyle	Enumeration	'StyleOne'	Type of callback function: <ul style="list-style-type: none"> • StyleOne = simple check callback function • StyleTwo = detailed check callback function • StyleThree = check callback function with hyperlinked results
Group	String	'' (null string)	Mechanism for grouping checks hierarchically into categories. Model Advisor displays a folder in its left pane for each group name that you specify. To display nested folders, separate multiple names with vertical bars, as in this example: MyCompany MyDepartment
Visible	Boolean	true	Show or hide check? <ul style="list-style-type: none"> • true = Display check in Model Advisor • false = Hide check
Enable	Boolean	true	Can user enable and disable check? <ul style="list-style-type: none"> • true = Display check box control • false = Hide check box control

Property	Data Type	Default Value	Description
Value	Boolean	true	<p>Initial status:</p> <ul style="list-style-type: none"> • true = Check is enabled • false = Check is disabled
LicenseName	Cell array	{ } (empty cell array)	<p>Cell array of names of product licenses required to enable the check. Model Advisor does not display the check if license requirements are not met.</p> <hr/> <p>Tip To find the correct text for license strings, type <code>help license</code> at the MATLAB command line.</p> <hr/>
Result	Cell array	{ } (empty cell array)	<p>Cell array used for storing the results returned by the callback function referenced by <code>CallbackHandle</code>.</p> <hr/> <p>Tip To set the icon associated with the check, use the <code>Simulink.ModelAdvisor.setCheckResultStatus</code> class.</p> <hr/>

How Visible, Enable, and Value Properties Interact

Typically, you modify the behavior of Visible, Enable, and Value properties in a process callback function (see “Defining a Process Callback Function” on page 6-22). The following chart illustrates how these properties interact:



Code Example: Check Definition Function

Here is an example of a function that defines the custom checks associated with the callback functions described in “Creating Callback Functions for Checks” on page 6-5. The check definition function returns a cell array of custom checks to be added to Model Advisor.

```
function recordCellArray = defineModelAdvisorChecks
recordCellArray = {};

% --- sample check 1
rec = Simulink.MdlAdvisorCheck;
rec.Title = 'Check Simulink block font';
rec.TitleID = 'Check Simulink block font';
rec.TitleTips = 'A sample check with style three callback';
rec.CallbackHandle = @SampleStyleThreeCallback;
rec.CallbackContext = 'None';
rec.CallbackStyle = 'StyleThree';
rec.Group = 'Demo';
% add current record into recordCellArray
recordCellArray{end+1} = rec;

% --- sample check 2
rec = Simulink.MdlAdvisorCheck;
rec.Title = 'Check Simulink window screen color';
rec.TitleID = 'Check Simulink window screen color';
rec.TitleTips = 'A sample check with style one callback';
rec.CallbackHandle = @SampleStyleOneCallback;
rec.CallbackContext = 'None';
rec.CallbackStyle = 'StyleOne';
rec.Group = 'Demo';
% add current record into recordCellArray
recordCellArray{end+1} = rec;

% --- sample check 3
rec = Simulink.MdlAdvisorCheck;
rec.Title = 'Check model optimization settings';
rec.TitleID = 'Check model optimization settings';
rec.TitleTips = 'A sample check with style two callback';
rec.CallbackHandle = @SampleStyleTwoCallback;
rec.CallbackContext = 'None';
rec.CallbackStyle = 'StyleTwo';
rec.Group = 'Demo';
% add current record into recordCellArray
recordCellArray{end+1} = rec;
```

Defining Custom Tasks

In this section...

“About Custom Tasks” on page 6-19

“Properties of Model Advisor Tasks” on page 6-19

“How Visible, Enable, and Value Properties Interact for Tasks” on page 6-20

“Code Example: Task Definition Function” on page 6-21

About Custom Tasks

Tasks are used to group checks in Model Advisor by functionality or usage. You define custom tasks in one or more functions that specify the properties of each instance of the class `Simulink.MdlAdvisorTask`. You must define one instance of this class for each custom task you want to add to Model Advisor and register the custom task as described in “Registering Custom Checks and Tasks” on page 6-29. The sections that follow describe how to define custom tasks.

Properties of Model Advisor Tasks

The following table describes the properties of the `Simulink.MdlAdvisorTask` class:

Property	Data Type	Default Value	Description
Title	String	'' (null string)	Name of the task as it should appear in Model Advisor
TitleID	String	'' (null string)	Permanent, unique identifier for the task. Typically, TitleID is set to the task Title string, but the value of TitleID remains constant even if the Title of the task changes.

Property	Data Type	Default Value	Description
TitleTips	String	' ' (null string)	Tooltip that displays help text when you move your mouse over the task.
Visible	Boolean	true	Show or hide task? <ul style="list-style-type: none"> • true = Display task in Model Advisor • false = Hide task
Enable	Boolean	true	Can user enable and disable task? <ul style="list-style-type: none"> • true = Display check box control for task • false = Hide check box control for task
Value	Boolean	true	Initial status: <ul style="list-style-type: none"> • true = Task is enabled • false = Task is disabled
CheckTitleIDs	Cell array	{ } (empty cell array)	Cell array of TitleIDs of checks associated with this task.

How Visible, Enable, and Value Properties Interact for Tasks

These properties interact the same way for tasks as for checks (see “How Visible, Enable, and Value Properties Interact” on page 6-17).

Code Example: Task Definition Function

Here is an example of a task definition function that associates the task with the checks defined in “Code Example: Check Definition Function” on page 6-17. The task definition function returns a cell array of custom tasks to be added to Model Advisor.

```
function taskCellArray = defineModelAdvisorTasks
taskCellArray = {};

% create a sample task
task = Simulink.MdlAdvisorTask;
task.Title = 'Demo task';
task.TitleID = 'Demo task';
task.TitleTips = 'Contains demo checks';
task.CheckTitleIDs = {'Check Simulink block font',...
    'Check Simulink window screen color',...
    'Check model optimization settings'};
% add current task into taskCellArray
taskCellArray{end+1} = task;
```

Defining a Process Callback Function

In this section...
“About Process Callback Functions” on page 6-22
“Process Callback Function Arguments” on page 6-22
“Code Example: Process Callback Function” on page 6-23

About Process Callback Functions

The process callback function is an optional function that lets you modify the appearance of checks and tasks in Model Advisor, and process check results at run-time. The process callback function specifies actions to be performed at different stages of Model Advisor execution:

- **configure stage:** Model Advisor executes `configure` actions at startup, after all checks and tasks have been initialized. At this stage, you can specify actions to customize how 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:** 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 Custom Checks and Tasks” on page 6-29. The sections that follow provide mode information about defining your own process callback functions.

Process Callback Function Arguments

The process callback function takes the following arguments:

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

Code Example: Process Callback Function

Here is an example of a process callback function that specifies actions in the `configure` stage to enable only the custom checks assigned to the Demo group in “Code Example: Check Definition Function” on page 6-17. In the `process_results` stage, this function pops up an informative dialog box for checks that do not pass.

```
function [checkCellArray taskCellArray] = ...
    ModelAdvisorProcessFunction(stage, system, checkCellArray, taskCellArray)
switch stage
case 'configure'
    for i=1:length(checkCellArray)
```

```
        % disable all checks that do not belong to Demo group
        if ~(strcmp(checkCellArray{i}.Group, 'Demo'))
            checkCellArray{i}.Enable = false;
            checkCellArray{i}.Value = false;
        end
    end
end
case 'process_results'
    for i=1:length(checkCellArray)
        % print message if check does not pass
        if checkCellArray{i}.Selected && ...
            (strcmp(checkCellArray{i}.TitleID, ...
                'Check Simulink window screen color'))
            if ischar(checkCellArray{i}.Result) && ...
                isempty(strfind(checkCellArray{i}.Result, 'Passed'));
                disp('Example message from Model Advisor Process callback:');
                disp(checkCellArray{i}.Result);
            end
        end
    end
end
end
```

Formatting Model Advisor Outputs

In this section...
“What Is the Model Advisor Formatting API?” on page 6-25
“Formatting Text” on page 6-25
“Formatting Lists” on page 6-26
“Formatting Tables” on page 6-27
“Formatting Paragraphs” on page 6-27
“Code Example: Model Advisor Formatted Output” on page 6-28

What Is the Model Advisor Formatting API?

You can use the Model Advisor Formatting API to produce formatted outputs in Model Advisor. The following constructors of the `ModelAdvisor` class provide the ability to format the output. For more information on each constructor and associated methods, click the link in the Constructor column.

Constructor	Description
<code>ModelAdvisor.Text</code>	Formats element text.
<code>ModelAdvisor.Paragraph</code>	Combines elements into paragraph format.
<code>ModelAdvisor.List</code>	Creates a list of elements.
<code>ModelAdvisor.LineBreak</code>	Adds a line break between elements.
<code>ModelAdvisor.Table</code>	Creates a table.
<code>ModelAdvisor.Image</code>	Adds an image to the output.

Formatting Text

Text is the simplest form of output, but you can format text in many different ways. The default text formatting is:

- Empty
- Default color (black)

- Unformatted (that is, 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, you can achieve this using the syntax

```
ModelAdvisor.Text(content, {attributes})
```

When you want multiple types of formatting, you must build the text, as shown in the next example:

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' to ensure uniform appearance of model.');
```



```
result = [t1, t2, t3, t4, t5];
```

You can add ASCII and Extended ASCII characters using the MATLAB `char` command. See the `ModelAdvisor.Text` constructor reference for more information.

Formatting Lists

You can create two types of lists, numbered and bulleted. The default list formatting is bulleted. Use the `ModelAdvisor.List` constructor to create and format lists (see `ModelAdvisor.List`). You can create lists with indented subsections, formatted as either numbered or bulleted, as shown in the next example:

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass', 'bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass', 'bold'}));
```



```
topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1', {'keyword', 'bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2', {'keyword', 'bold'}), subList]);
```

Formatting Tables

The default table formatting is:

- Default color (black)
- Left justified
- Bold title, row, and column headings

You can change table formatting using the `ModelAdvisor.Table` constructor (see `ModelAdvisor.Table`). The following example code creates a subtable within a table, as shown in the figure:

```
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);
```

Table 1		
Table 2		
Header 1	Header 2	Header 3

Formatting Paragraphs

Paragraphs need to be handled explicitly because most markup languages do not support line breaks. The default paragraph formatting is:

- Empty
- Default color (black)

- Unformatted, that is, not bold, italicized, underlined, linked, subscripted, or superscripted
- Aligned left

If you want to change paragraph formatting, use the `ModelAdvisor.Paragraph` constructor (see `ModelAdvisor.Paragraph`).

Code Example: Model Advisor Formatted Output

The following is the example from “Simple Check Callback Function” on page 6-5, reformatted using the Model Advisor Formatting API.

```
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
if strcmp(get_param(bdroot(system), 'ScreenColor'),'white')
    result = ModelAdvisor.Text('Passed',{ 'pass' });
    mdladvObj.setCheckResultStatus(true);
else
    msg1 = ModelAdvisor.Text(...
        ['It is recommended to select a Simulink window screen color'...
        ' of white to ensure a readable and printable model. Click ']);
    msg2 = ModelAdvisor.Text('here');
    msg2.setHyperlink('matlab: set_param(bdroot, 'ScreenColor','white')');
    msg3 = ModelAdvisor.Text(' to change screen color to white. ');
    result = [msg1, msg2, msg3];
    mdladvObj.setCheckResultStatus(false);
end
```

Registering Custom Checks and Tasks

In this section...

“About Custom Checks and Tasks Registration” on page 6-29

“Methods for Registering Custom Checks and Tasks” on page 6-30

“Code Example: Methods for Registering Custom Checks and Tasks” on page 6-30

About Custom Checks and Tasks Registration

To register checks and tasks in Model Advisor, you must create the function `sl_customization()` in the `sl_customization.m` file on your MATLAB path. This function accepts one argument, a handle to an object called `Simulink.CustomizationManager`, as in this example:

```
function sl_customization(cm)
```

The customization manager object includes methods for registering custom checks and tasks. You should use these methods to register customizations specific to your application, as described and demonstrated in the sections that follow.

Simulink reads `sl_customization.m` files when it starts. If you subsequently change the contents of your customization file, update your environment by performing these tasks:

- 1 Close Model Advisor if you have previously launched it.
- 2 Enter the following command at the MATLAB command line:

```
sl_refresh_customizations
```

- 3 If you previously launched Model Advisor, reinitialize it either by removing the `slprj` folder from your working directory or by restarting your model.
- 4 Restart Model Advisor.

Methods for Registering Custom Checks and Tasks

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

- `addModelAdvisorCheckFcn (@checkDefinitionFcn)`

Adds the checks specified by the check definition function to the **By Product** folder of Model Advisor unless otherwise specified using `Simulink.MdlAdvisorTask`. The `checkDefinitionFcn` argument is a handle to the function that defines all custom checks to be added to Model Advisor as instances of the `Simulink.MdlAdvisorCheck` class (see “Defining Custom Checks” on page 6-13).
- `addModelAdvisorTaskFcn (@taskDefinitionFcn)`

Adds the tasks specified by the task definition function to the **By Product** folder of Model Advisor unless otherwise specified using `Simulink.MdlAdvisorTask`. The `taskDefinitionFcn` argument is a handle to the function that defines all custom tasks to be added to Model Advisor as instances of the `Simulink.MdlAdvisorTask` class (see “Defining Custom Tasks” on page 6-19).
- `addModelAdvisorProcessFcn (@modelAdvisorProcessFcn)`

Adds the process callback function for Model Advisor (see “Defining a Process Callback Function” on page 6-22).

Code Example: Methods for Registering Custom Checks and Tasks

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

```
function sl_customization(cm)

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

% register custom tasks
cm.addModelAdvisorTaskFcn(@defineModelAdvisorTasks);

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


Functions — By Category

Requirements Management
Interface (p. 7-2)

Model Coverage (p. 7-3)

Model Advisor Formatting API
(p. 7-4)

Access Requirements Management
Interface

Configure and execute model
coverage tests; store and report test
results

Format Model Advisor outputs

Requirements Management Interface

rmi	Requirements Management Interface API
rminav	Start Requirements Management Interface

Model Coverage

<code>conditioninfo</code>	Display condition coverage information for model object
<code>cv.cvdatagroup</code>	Group together multiple cvdata objects
<code>cv.cvtestgroup</code>	Group together multiple cvtest objects
<code>cvexit</code>	Exit model coverage environment
<code>cvhtml</code>	Produce HTML report from model coverage objects in memory
<code>cvload</code>	Load coverage tests and results stored in file
<code>cvmodelview</code>	Display model coverage results with model coloring
<code>cvsave</code>	Save coverage tests and results to file
<code>cvsim</code>	Simulate and return model coverage results for test objects
<code>cvsimref</code>	Simulate and return model coverage results for referenced models
<code>cvtest</code>	Create model coverage test specification object
<code>decisioninfo</code>	Display decision coverage information for model object
<code>mcdcinfo</code>	Display modified condition/decision coverage information for model object
<code>sigrangeinfo</code>	Display signal range coverage information for model object
<code>tableinfo</code>	Display lookup table coverage information for model object

Model Advisor Formatting API

`ModelAdvisor.Image`

Include image in Model Advisor output

`ModelAdvisor.LineBreak`

Insert line break

`ModelAdvisor.List`

Create list class

`ModelAdvisor.Paragraph`

Create and format paragraph

`ModelAdvisor.Table`

Create table class

`ModelAdvisor.Text`

Create Model Advisor text output

Functions — Alphabetical List

conditioninfo

Purpose Display condition coverage information for model object

Syntax

```
coverage = conditioninfo(cvdo, object)
coverage = conditioninfo(cvdo, object, ignore_descendants)
[coverage, description] = conditioninfo(cvdo, object)
```

Description `coverage = conditioninfo(cvdo, object)` returns condition coverage results from the cvdata object `cvdo` for the model component specified by `object`. See “Specifying a Model Object” on page 8-3 for more information about the `object` argument. The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — the number of condition outcomes satisfied for `object`
- `total_outcomes` — the total number of condition outcomes for `object`

Note `coverage` is empty if `cvdo` does not contain condition coverage results for `object`.

`coverage = conditioninfo(cvdo, object, ignore_descendants)` returns condition coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, description] = conditioninfo(cvdo, object)` returns condition coverage results and textual descriptions of each condition in `object`. `description` is a structure array containing the following fields:

- `text` — string describing a condition or the block port to which it applies
- `trueCnts` — number of times the condition was true in a simulation
- `falseCnts` — number of times the condition was false in a simulation

Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
BlockPath	Full path to a Simulink model or block
BlockHandle	Handle to a Simulink model or block
slObj	Handle to a Simulink API object
sfID	Stateflow ID
sfObj	Handle to a Stateflow API object
{BlockPath, sfID}	Cell array with the path to a Stateflow block and the ID of an object contained in that chart
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable condition coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.condition = 1;
```

conditioninfo

```
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the condition coverage results for the Logic block (in the Gain subsystem) and determine its percentage of condition outcomes covered.

```
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');  
cov = conditioninfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

See Also

decisioninfo, mdcinfo

Purpose Group together multiple cvdata objects

Description Instances of this class contain a collection of cvdata objects. For more information, see “Extracting Results from cv.cvdatagroup” on page 5-51.

Property Summary

Name	Description
name	Name of the cv.cvdatagroup object.

Method Summary

Name	Description
allNames	Get all model names associated with cv.cvdatagroup object.
get	Get cvdata objects.

Properties

name

Description

Name of the cv.cvdatagroup object.

Data Type

string

Access

RW

Methods

allNames

Purpose

Get all model names associated with cv.cvdatagroup object.

Syntax

allNames

Description

Returns a cell array of strings identifying all model names associated with a cv.cvdatagroup object.

cv.cvdatagroup

get

Purpose

Get cvdata objects.

Syntax

```
get(modelName)
```

Arguments

modelName

String specifying the name of a model whose cvdata object is to be returned.

Description

Returns the cvdata object that corresponds to modelName.

See Also

cvsimref

Purpose Group together multiple cvtest objects

Description Instances of this class contain a collection of cvtest objects. For more information, see “Creating a Test Group with cv.cvtestgroup” on page 5-50.

Property Summary

Name	Description
name	Name of the cv.cvtestgroup object.

Method Summary

Name	Description
add	Add cvtest objects.
allNames	Get all model names associated with cv.cvtestgroup object.
get	Get cvtest objects.

Properties

name

Description

Name of the cv.cvtestgroup object.

Data Type

string

Access

RW

Methods

add

Purpose

Add cvtest objects.

Syntax

add(cvto1, cvto2, ...)

cv.cvtestgroup

Arguments

cvto1

String specifying the name of a cvtest object to be added to the cv.cvtestgroup object.

cvto2

String specifying the name of another cvtest object to be added to the cv.cvtestgroup object.

Description

Adds cvtest objects to a cv.cvtestgroup object.

allNames

Purpose

Get all model names associated with cv.cvtestgroup object.

Syntax

allNames

Description

Returns a cell array of strings identifying all model names associated with a cv.cvtestgroup object.

get

Purpose

Get cvtest objects.

Syntax

get(modelName)

Arguments

modelName

String specifying the name of a model whose cvtest object is to be returned.

Description

Returns the cvtest object that corresponds to modelName.

See Also

cvsimref, cvtest

Purpose Exit model coverage environment

Syntax `cvexit`

Description `cvexit` exits the model coverage environment. Issuing this command causes the Model Coverage Tool to close the Coverage Display window and remove coloring from a block diagram that displays its model coverage results.

cvhtml

Purpose Produce HTML report from model coverage objects in memory

Syntax

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

Description Use the `cvhtml` command to produce an HTML report from `cvdata` objects you produce when you run a model coverage test in simulation.

Note The model must be open when using the `cvhtml` command to generate its coverage report.

`cvhtml(file, cvdo)` creates an HTML report of the coverage results in the `cvdata` object `cvdo`, which is written to the file `file` in the current MATLAB directory.

`cvhtml(file, cvdo1, cvdo2,...)` creates a combined report of several `cvdata` objects. The results from each object are displayed in a separate column of the HTML report. Each `cvdata` object must correspond to the same root model or subsystem, or the function produces errors.

`cvhtml(file, cvdo1, cvdo2,..., options)` creates a combined report of several `cvdata` objects using the report options specified by the options string. The table in “Report Options” on page 8-11 lists available options and their default settings. To enable an option, set it equal to 1 (e.g., `'-hTR=1'`); to disable an option, set it equal to 0 (e.g., `'-bRG=0'`). To specify multiple report options, list individual options in a single options string separated by commas or spaces (e.g., `'-hTR=1 -bRG=0 -scm=0'`).

`cvhtml(file, cvdo1, cvdo2,..., options, detail)` creates a combined report of several `cvdata` objects and specifies the detail level of the report with the value of `detail`, an integer between 0 and 3.

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

Report Options

The following table summarizes the report options that you can specify using cvhtml. See “Settings” on page 5-16 under the “Report Tab” section in the Simulink Verification and Validation User’s Guide for more information.

Option	Description	Default Setting
-aTS	Include each test in the model summary	on
-bRG	Produce bar graphs in the model summary	on
-bTC	Use two color bar graphs (red, blue)	off
-hTR	Display hit/count ratio in the model summary	off
-nFC	Do not report fully covered model objects	off
-scm	Include cyclomatic complexity numbers in summary	on
-bcm	Include cyclomatic complexity numbers in block details	on

cvload

Purpose Load coverage tests and results stored in file

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

Description The `cvload` command loads into memory the coverage tests and results stored in a file by the `cvsave` command.

`[cvtos, cvdos] = cvload(filename)` loads the tests and data stored in the text file `filename.cvt`. The `cvtest` objects that are successfully loaded are returned in `cvtos`, a cell array of `cvtest` objects. The `cvdata` objects that are successfully loaded are returned in `cvdos`, a cell array of `cvdata` objects. `cvdos` has the same size as `cvtos`, but can contain empty elements if a particular test has no results.

`[cvtos, cvdos] = cvload(filename, restoretotal)` restores the cumulative results from prior runs if `restoretotal` is 1. If `restoretotal` is unspecified or 0, the model's cumulative results are cleared.

cvload Special Considerations

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

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

Purpose Display model coverage results with model coloring

Syntax `cvmodelview(cvdo)`

Description `cvmodelview(cvdo)` displays coverage results from the `cvdata` object `cvdo` by coloring the Simulink model (see “Displaying Model Coverage with Model Coloring” on page 5-37).

Example The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
data = cvsim(testObj)
```

Afterward, issue the following command to display the model coverage results by coloring the block diagram.

```
cvmodelview(data)
```

cvsave

Purpose

Save coverage tests and results to file

Syntax

```
cvsave(filename, model)
cvsave(filename, cvto1, cvto2, ...)
cvsave(filename, cvdo1, cvdo2, ...)
```

Description

Save the coverage tests and results from simulations to a file with the function `cvsave`.

`cvsave(filename, model)` saves all the tests (`cvtest` objects) and results (`cvdata` objects) in memory related to the model `model` in the text file `filename.cvt`.

`cvsave(filename, cvto1, cvto2, ...)` saves the tests in the `cvtest` objects `cvto1`, `cvto2`, ... in the text file `filename.cvt`. Information about the referenced models is also saved.

`cvsave(filename, cvdo1, cvdo2, ...)` saves the tests, test results, and referenced models' structure for `cvdata` objects `cvdo1`, `cvdo2`, ... to the text file `filename.cvt`.

Purpose	Simulate and return model coverage results for test objects
Syntax	<pre>cvdo = cvsim(cvto) [cvdo,t,x,y] = cvsim(cvto) [cvdo,t,x,y] = cvsim(cvto, timespan, options) [cvdo,t,x,y] = cvsim(cvto, label, setupcmd) [cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)</pre>
Description	You simulate a test specification object (a cvtest object) with the cvsim command.

Note You do not have to enable model coverage reporting for the model to use the cvsim command.

`cvdo = cvsim(cvto)` executes the cvtest object cvto by starting a simulation run for the corresponding model. The results are returned in the cvdata object cvdo.

`[cvdo,t,x,y] = cvsim(cvto)` returns the time vector t, matrix of state values x, and matrix of output values y from the simulation. Refer to the sim command in the Simulink documentation for descriptions of the parameters t, x, and y.

`[cvdo,t,x,y] = cvsim(cvto, timespan, options)` returns the time vector t, matrix of state values x, and matrix of output values y from the simulation, and overrides default simulation values with the values for timespan and options. Refer to the sim command in the *Simulink Reference* for descriptions of the parameters t, x, y, timespan, and options.

`[cvdo,t,x,y] = cvsim(cvto, label, setupcmd)` creates the cvtest object cvto and simulates it in one command. The arguments label and setupcmd are passed directly to the cvtest command, which creates the cvtest object cvto.

cvsim

`[cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)` executes the cvtest objects `cvto1`, `cvto2`, ... and returns the results in the set of cvdata objects `cdvo1`, `cvdo2`,

Purpose	Simulate and return model coverage results for referenced models
Syntax	<pre>cvdg = cvsimref(topmodelName) cvdg = cvsimref(topmodelName, cvtg) [cvdg,t,x,y] = cvsimref(topmodelName, cvtg) [cvdg,t,x,y] = cvsimref(topmodelName, cvtg, timespan, options) [cvdg1, cvdg2, ...] = cvsimref(topmodelName, cvtg1, cvtg2, ...)</pre>
Description	Use the <code>cvsimref</code> function to record coverage for referenced models in a hierarchy. For more information, see “Model Coverage for Referenced Models” on page 5-47.

Note You do not have to enable model coverage reporting for any of the models in a model hierarchy to use the `cvsimref` command.

`cvdg = cvsimref(topmodelName)` simulates the top model that `topmodelName` specifies, collects model coverage data, and returns the results in the `cv.cvdagroup` object `cvdg`.

`cvdg = cvsimref(topmodelName, cvtg)` executes the `cv.cvtestgroup` object `cvtg` by starting a simulation run for the corresponding top model, `topmodelName`. The results are returned in the `cv.cvdagroup` object `cvdg`.

`[cvdg,t,x,y] = cvsimref(topmodelName, cvtg)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation. Refer to the `sim` function in the *Simulink Reference* for descriptions of the parameters `t`, `x`, and `y`.

`[cvdg,t,x,y] = cvsimref(topmodelName, cvtg, timespan, options)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation, and overrides default simulation values with the values for `timespan` and `options`. Refer to the `sim`

cvsimref

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

```
[cvdg1, cvdg2, ...] = cvsimref(topModelName, cvtg1, cvtg2,  
...) executes the cv.cvtestgroup objects cvtg1, cvtg2, ... and  
returns the results in the set of cv.cvdatabgroup objects cdvg1, cdvg2,  
....
```

See Also

`cv.cvdatabgroup`, `cv.cvtestgroup`

Purpose Create model coverage test specification object

Syntax

```

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

Description The cvtest command creates a test specification object, that you simulate with the cvsim command.

cvto = cvtest(root) creates a test object with the handle cvto. root is the name of, or a handle to, a Simulink model or a subsystem of a model. Only the specified model or subsystem and its descendants are subject to model coverage testing.

cvto = cvtest(root, label) creates a test object with the label label, which is used for reporting results.

cvto = cvtest(root, label, setupcmd) creates a test object with the setup command setupcmd and labels it with label. The setup command is executed in the base MATLAB workspace just prior to running the instrumented simulation. This command is useful for loading data prior to a test.

A test object has the following structure.

Field	Description
id	Read-only internal data-dictionary ID
modelcov	Read-only internal data-dictionary ID
rootPath	Name of the system or subsystem instrumented for analysis
label	String used when reporting results

Field	Description
setupCmd	Command executed in the base workspace just prior to simulation.
settings.condition	Set to 1 if condition coverage is desired
settings.decision	Set to 1 if decision coverage is desired
settings.mcdc	Set to 1 if MC/DC coverage is desired
settings.sigrange	Set to 1 if signal range coverage is desired
settings.tableExec	Set to 1 if lookup table coverage is desired

Purpose Display decision coverage information for model object

Syntax

```
coverage = decisioninfo(cvdo, object)
coverage = decisioninfo(cvdo, object, ignore_descendants)
[coverage, description] = decisioninfo(cvdo, object)
```

Description `coverage = decisioninfo(cvdo, object)` returns decision coverage results from the cvdata object `cvdo` for the model component specified by `object`. See “Specifying a Model Object” on page 8-22 for more information about the object argument. The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — the number of decision outcomes satisfied for `object`
- `total_outcomes` — the total number of decision outcomes for `object`

Note `coverage` is empty if `cvdo` does not contain decision coverage results for `object`.

`coverage = decisioninfo(cvdo, object, ignore_descendants)` returns decision coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, description] = decisioninfo(cvdo, object)` returns decision coverage results and textual descriptions of decision points associated with `object`. `description` is a structure array containing the following fields:

- `decision.text` — string describing a decision point, e.g., 'U > LL'
- `decision.outcome.text` — string describing a decision outcome, i.e., 'true' or 'false'
- `decision.outcome.executionCount` — number of times a decision outcome occurred in a simulation

Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
BlockPath	Full path to a Simulink model or block
BlockHandle	Handle to a Simulink model or block
sObj	Handle to a Simulink API object
sfID	Stateflow ID
sfObj	Handle to a Stateflow API object
{BlockPath, sfID}	Cell array with the path to a Stateflow block and the ID of an object contained in that chart
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable decision coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
testObj.settings.decision = 1;
```

```
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the decision coverage results for the Saturation block and determine its percentage of decision outcomes covered.

```
blk_handle = get_param([mdl, '/Saturation'], 'Handle');  
cov = decisioninfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

See Also

conditioninfo, mcdcinfo

mcdcinfo

Purpose Display modified condition/decision coverage information for model object

Syntax

```
coverage = mcdcinfo(cvdo, object)
coverage = mcdcinfo(cvdo, object, ignore_descendants)
[coverage, description] = mcdcinfo(cvdo, object)
```

Description `coverage = mcdcinfo(cvdo, object)` returns modified condition/decision coverage results from the cvdata object `cvdo` for the model component specified by `object`. See “Specifying a Model Object” on page 8-25 for more information about the `object` argument. The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — the number of condition/decision outcomes satisfied for `object`
- `total_outcomes` — the total number of condition/decision outcomes for `object`

Note `coverage` is empty if `cvdo` does not contain modified condition/decision coverage results for `object`.

`coverage = mcdcinfo(cvdo, object, ignore_descendants)` returns modified condition/decision coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, description] = mcdcinfo(cvdo, object)` returns modified condition/decision coverage results and textual descriptions of each condition/decision in `object`. `description` is a structure array containing the following fields:

- `text` — string denoting whether the condition/decision is associated with a block output or Stateflow transition

- `condition.text` — string describing a condition/decision or the block port to which it applies
- `condition.achieved` — logical array indicating whether a condition case has been fully covered
- `condition.trueRslt` — string representing a condition case expression that produces a true result
- `condition.falseRslt` — string representing a condition case expression that produces a false result

See “MC/DC Analysis Table” on page 5-25 for more information about the data contained in these fields.

Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
BlockPath	Full path to a Simulink model or block
BlockHandle	Handle to a Simulink model or block
slObj	Handle to a Simulink API object
sfID	Stateflow ID
sfObj	Handle to a Stateflow API object
{BlockPath, sfID}	Cell array with the path to a Stateflow block and the ID of an object contained in that chart

Object Specification	Description
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable modified condition/decision coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
testObj.settings.mcdc = 1;  
data = cvsims(testObj)
```

Afterward, issue the following commands to retrieve the modified condition/decision coverage results for the Logic block (in the Gain subsystem) and determine its percentage of condition/decision outcomes covered.

```
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');  
cov = mcdcinfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

See Also

`conditioninfo`, `decisioninfo`

Purpose Include image in Model Advisor output

Syntax `object = ModelAdvisor.Image`

Arguments `object`
A variable representing the image object created.

Description Specify an image to appear in the Model Advisor output. Model Advisor supports many image formats, including but not limited to jpg, bmp, and gif.

Method Summary

Name	Description
“setHyperlink” on page 8-27	Specify hyperlink location
“setImageSource” on page 8-27	Specify image location

Methods

setHyperlink

Purpose

Specify hyperlink location

Syntax

`setHyperlink(url)`

Arguments

url

A string that specifies the location of the link.

Description

Specifies the location of the hyperlink.

setImageSource

Purpose

Specify image location

Syntax

`setImageSource(source)`

ModelAdvisor.Image

Arguments

source

A string specifying the location of the image.

Description

Specifies the location of the image.

Example

```
report_image = ModelAdvisor.Image;  
report_image = ModelAdvisor.Image; report_image.setImageSource(...  
    'http://www.mathworks.com/access/helpdesk/help/techdoc/learn_matlab/p09.gif');
```

See Also

ModelAdvisor.LineBreak, ModelAdvisor.List,
ModelAdvisor.Paragraph, ModelAdvisor.Table, ModelAdvisor.Text

Purpose Insert line break

Syntax `line_break_obj = ModelAdvisor.LineBreak`

Arguments `line_break_obj`
A variable representing the line break object created.

Description Use instances of this class to insert line breaks in Model Advisor outputs.

Example

```
report_paragraph = ModelAdvisor.Paragraph;
report_text = ModelAdvisor.Text('Model Advisor', {'bold'});
report_text.setItalic(true);

report_text2 = ModelAdvisor.Text('Check Report', {'bold'});

line_break = ModelAdvisor.LineBreak;

report_paragraph.addItem([report_text line_break report_text2]);
```

See Also `ModelAdvisor.Image`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`,
`ModelAdvisor.Table`, `ModelAdvisor.Text`

ModelAdvisor.List

Purpose Create list class

Syntax `list = ModelAdvisor.List`

Arguments `list`
A variable representing the list object created.

Description Use instances of this class to create list formatted outputs. Creates a new list object.

Method Summary

Name	Description
“addItem” on page 8-30	Add list item
“setType” on page 8-30	Specify list type

Methods

addItem

Purpose

Add list item

Syntax

`addItem(element)`

Arguments

element

Element, cell array of elements, or string to be added. When a cell array of elements is added, they form different rows in the list.

Description

This method adds items to the list.

setType

Purpose

Specify list type

Syntax

`setType(listType)`

Arguments

listType

String specifying type of list, either numbered or bulleted.

Description

This method specifies the type of list created.

Example

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));

topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1',{'keyword','bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2',{'keyword','bold'}), subList]);
```

See Also

ModelAdvisor.Image, ModelAdvisor.LineBreak,
ModelAdvisor.Paragraph, ModelAdvisor.Table, ModelAdvisor.Text

ModelAdvisor.Paragraph

Purpose Create and format paragraph

Syntax `para_obj = ModelAdvisor.Paragraph`

Arguments `para_obj`
A variable representing the paragraph object created.

Description Creates and formats a paragraph.

Method Summary

Name	Description
“setAlign” on page 8-32	Specify paragraph alignment
“addItem” on page 8-33	Add paragraph element

Methods

setAlign

Purpose

Specify paragraph alignment

Syntax

`setAlign(alignment)`

Arguments

alignment

A string that specifies the alignment of the text. Possible alignments include:

left

Align left

right

Align right

center

Align center

Description

Specifies the paragraph alignment. The default is left.

addItem

Purpose

Add paragraph element

Syntax

```
addItem(element)
```

Arguments

element

A string, element, or cell array of elements to add to the paragraph.

Description

Adds an element to the paragraph.

Example

```
report_paragraph = ModelAdvisor.Paragraph;  
report_paragraph.setAlign('center');  
  
report_text = ModelAdvisor.Text('Magic Square', {'bold'});  
report_text.setItalic(true);  
  
report_image = ModelAdvisor.Image;  
report_image = ModelAdvisor.Image; report_image.setImageSource(...  
    'http://www.mathworks.com/access/helpdesk/help/techdoc/learn_matlab/p09.gif');  
  
line_break = ModelAdvisor.LineBreak;  
  
report_paragraph.addItem([report_text line_break line_break report_image]);
```

See Also

ModelAdvisor.Image, ModelAdvisor.LineBreak, ModelAdvisor.List,
ModelAdvisor.Table, ModelAdvisor.Text

ModelAdvisor.Table

Purpose Create table class

Syntax `table = ModelAdvisor.Table(row, column)`

Arguments

table
A variable representing the table object created.

row
An integer specifying the number of rows the table contains.

column
An integer specifying the number of columns the table contains.

Description Use instances of this class to create and format a table. Specify the number of rows and columns in a table, excluding the table title and table heading row.

Method Summary

Name	Description
“getEntry” on page 8-35	Get cell contents
“setColHeading” on page 8-35	Specify table column title
“setColHeadingAlign” on page 8-36	Specify column title alignment
“setColWidth” on page 8-36	Specify column widths
“setEntry” on page 8-37	Add cell to table
“setEntryAlign” on page 8-37	Specify cell alignment
“setHeading” on page 8-38	Specify table title
“setHeadingAlign” on page 8-39	Specify table title alignment
“setRowHeading” on page 8-39	Specify table row title
“setRowHeadingAlign” on page 8-40	Specify row title alignment

Methods

getEntry

Purpose

Get cell contents

Syntax

```
content = getEntry(row, column)
```

Arguments

row

An integer specifying the row.

column

An integer specifying the column.

content

An element object or object array specifying the content of the table entries.

Description

Gets the contents of a specified cell.

setColHeading

Purpose

Specify table column title

Syntax

```
setColHeading(column, heading)
```

Arguments

column

An integer specifying column number.

heading

A string, element object, or object array specifying the table column title.

Description

Specifies the table column title.

setColHeadingAlign

Purpose

Specify column title alignment

Syntax

```
setColHeadingAlign(column, alignment)
```

Arguments

column

An integer specifying column number.

alignment

A string specifying the cell alignment. Possible values are:

left

Align left

right

Align right

center

Align center

Description

Specifies the alignment of the column headings.

setColWidth

Purpose

Specify column widths

Syntax

```
setColWidth(column, width)
```

Arguments

column

An integer specifying column number.

width

An integer or array of integers specifying the column widths, relative to the entire table width.

Description

Specifies the table column widths relative to the entire table width. If column widths are [1 2 3], the second column is twice the width of the first column, and the third column is three times the width of the first column. Unspecified columns have a default width of 1. For example:

```
setColWidth(1, 1);  
setColWidth(3, 2);
```

specifies [1 1 2] column widths.

setEntry

Purpose

Add cell to table

Syntax

```
setEntry(row, column, string)  
setEntry(row, column, content)
```

Arguments

row

An integer specifying the row.

column

An integer specifying the column.

string

A string representing the contents of the entry

content

An element object or object array specifying the content of the table entries.

Description

Add a cell entry to a table.

setEntryAlign

Purpose

Specify cell alignment

ModelAdvisor.Table

Syntax

`setEntryAlign(row, column, alignment)`

Arguments

row

An integer specifying row number.

column

An integer specifying column number.

alignment

A string specifying the cell alignment. Possible values are:

`left`

Align left

`right`

Align right

`center`

Align center

Description

Specifies the alignment of the table cells.

setHeading

Purpose

Specify table title

Syntax

`setHeading(title)`

Arguments

title

A string, element object, or object array, specifying the table title.

Description

Specifies the table title, which is the first line of the table.

setHeadingAlign

Purpose

Specify table title alignment

Syntax

`setHeadingAlign(alignment)`

Arguments

alignment

A string specifying the cell alignment. Possible values are:

`left`

Align left

`right`

Align right

`center`

Align center

Description

Specifies the alignment of table titles.

setRowHeading

Purpose

Specify table row title

Syntax

`setRowHeading(row, heading)`

Arguments

row

An integer specifying row number.

heading

A string, element object, or object array specifying the table row title.

Description

Specifies the table row title.

ModelAdvisor.Table

setRowHeadingAlign

Purpose

Specify row title alignment

Syntax

```
setRowHeadingAlign(row, alignment)
```

Arguments

row

An integer specifying row number.

alignment

A string specifying the cell alignment. Possible values are:

left

Align left

right

Align right

center

Align center

Description

Specifies the alignment of row titles.

Example

```
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);
```

See Also

ModelAdvisor.Image, ModelAdvisor.LineBreak, ModelAdvisor.List,
ModelAdvisor.Paragraph, ModelAdvisor.Text

Purpose

Create Model Advisor text output

Syntax

```
text = ModelAdvisor.Text(content, {attribute})
```

Arguments

text

A variable representing the text object created.

content

A string specifying the content.

attribute

A string specifying the formatting of the content. If no attribute is specified, the output text has default coloring with no formatting implemented. Possible formatting options include:

bold

Text is bold.

italic

Text is italic.

underlined

Text is underlined.

normal

Text is default color.

pass

Text is green.

warn

Text is yellow.

fail

Text is red.

keyword

Text is blue.

subscript

Text is subscripted.

ModelAdvisor.Text

superscript
Text is superscripted.

retainspacereturn
Text retains spacing and returns.

Description

Use instances of this constructor to create formatted text for Model Advisor outputs. You can implement the `ModelAdvisor.Text` constructor with or without an *attribute* value. If *content* is empty, empty text is output.

Method Summary

Name	Description
“setBold” on page 8-42	Bold text
“setColor” on page 8-43	Color text
“setHyperlink” on page 8-43	Hyperlink text
“setItalic” on page 8-44	Italic text
“setRetainSpaceReturn” on page 8-44	Retain spacing and returns in text
“setSubscript” on page 8-44	Subscripted text
“setSuperscript” on page 8-45	Superscripted text
“setUnderlined” on page 8-45	Underlined text

Methods

setBold

Purpose

Bold text

Syntax

```
setBold(mode)
```

Arguments

mode

A Boolean value indicating bold formatting of text, either on (`true`) or off (`false`).

Description

This method makes text bold.

setColor**Purpose**

Color text

Syntax

```
setColor(color)
```

Arguments

color

An enumerated string specifying the color of the text. Possible formatting options include:

normal

Text is default color.

pass

Text is green.

warn

Text is yellow.

fail

Text is red.

keyword

Text is blue.

Description

This method colors text.

setHyperlink**Purpose**

Hyperlink text

Syntax

```
setHyperlink(url)
```

Arguments

url

A string that specifies the location of the link.

Description

This method hyperlinks text to the specified URL.

setItalic

Purpose

Italic text

Syntax

`setItalic(mode)`

Arguments

mode

A Boolean value indicating italicized formatting of text, either on (true) or off (false).

Description

This method italicizes text.

setRetainSpaceReturn

Purpose

Retain spacing and returns

Syntax

`setRetainSpaceReturn(mode)`

Arguments

mode

A Boolean value indicating whether to preserve space and return formatting of text, either on (true) or off (false).

Description

This method retains spaces and carriage returns in text.

setSubscript

Purpose

Subscript text

Syntax

`setSubscript(mode)`

Arguments

mode

A Boolean value indicating subscripted formatting of text, either on (true) or off (false).

Description

This method subscripts text.

setSuperscript**Purpose**

Superscript text

Syntax

`setSuperscript(mode)`

Arguments

mode

A Boolean value indicating superscripted formatting of text, either on (true) or off (false).

Description

This method superscripts text.

setUnderlined**Purpose**

Underline text

Syntax

`setUnderlined(mode)`

Arguments

mode

A Boolean value indicating underlined formatting of text, either on (true) or off (false).

Description

This method underlines text.

ModelAdvisor.Text

Example

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' to ensure uniform appearance of model.');
```



```
result = [t1, t2, t3, t4, t5];
```

See Also

ModelAdvisor.Image, ModelAdvisor.LineBreak, ModelAdvisor.List,
ModelAdvisor.Paragraph, ModelAdvisor.Table

Purpose

Requirements Management Interface API

Syntax

```
rmi setup
rmi setupdoors
rmi register linktypename
rmi unregister linktypename
rmi linktypelist

reqlinks = rmi('createempty')
reqlinks = rmi('get', object)
rmi('set', object, reqlinks)
rmi('cat', object, reqlinks)
cnt = rmi('count', object)
rmi('clearall', object)

cmdstr = rmi('navcmd', object)
[cmdstr, titlestr] = rmi('navcmd', object)
guidstr = rmi('guidget', object)
object = rmi('guidlookup', model, guidstr)
rmi('highlightModel', object)
rmi('unhighlightModel', object)
rmi('view', object, index)
dialog = rmi('edit', object)
rmi('objCopy', object)
```

Description

Use the rmi command to interact programmatically with the Requirements Management Interface (RMI).

- “RMI Setup” on page 8-47
- “Requirement Link Management” on page 8-48
- “Navigation and Display Options” on page 8-48

RMI Setup

rmi setup configures the RMI for use with your computer and installs the DOORS interface, if needed. See “Configuring the Requirements

Management Interface” on page 2-4 for more information about using this command to set up the RMI.

`rmi register linktypename` registers the custom link type specified by the M-file function `linktypename`. See “Linking to Custom Types of Requirements Documents” on page 2-28 for more information.

`rmi unregister linktypename` removes the custom link type specified by the M-file function `linktypename`.

`rmi linktypelist` displays a list of the currently registered link types. The list indicates whether each link type is built-in or custom and provides the path to the M-file function used for its registration.

Requirement Link Management

`reqlinks = rmi('createempty')` creates an empty instance of the requirement links data structure. See “Requirement Links Data Structure” on page 8-49 for more information.

`reqlinks = rmi('get', object)` returns the requirement links data structure for `object`. `object` is the name or handle of a Simulink or Stateflow object with which requirements can be associated.

`rmi('set', object, reqlinks)` sets the requirement links data structure `reqlinks` to `object`.

`rmi('cat', object, reqlinks)` appends the requirement links data structure `reqlinks` to the end of the existing structure associated with `object`. If no structure exists, the RMI sets `reqlinks` to `object`.

`cnt = rmi('count', object)` returns the number of requirement links associated with `object`.

`rmi('clearall', object)` removes the requirement links data structure associated with `object`, deleting its requirements.

Navigation and Display Options

`cmdstr = rmi('navcmd', object)` returns the MATLAB command string used to navigate to `object`. `object` is the name or handle of a Simulink or Stateflow object with which requirements can be

associated. See “Navigating to Simulink from External Documents” on page 2-42 for more information.

`[cmdstr, titlestr] = rmi('navcmd', object)` returns the MATLAB command string `cmdstr` and the title string `titlestr` that provides descriptive text for `object`.

`guidstr = rmi('gidget', object)` returns the globally unique identifier for `object`. A globally unique identifier is created for `object` if it lacks one. See “Providing Unique Object Identifiers” on page 2-42 for more information.

`object = rmi('guidlookup', model, guidstr)` returns the object name in `model` that has the globally unique identifier specified by `guidstr`.

`rmi('highlightModel', object)` highlights all of the objects in the parent model of `object` that have requirement links.

`rmi('unhighlightModel', object)` removes highlighting of objects in the parent model of `object` that have requirement links.

`rmi('view', object, index)` accesses the requirement numbered `index` in the requirements document associated with `object`. `index` is an integer that represents the n th requirement linked to `object`.

`dialog = rmi('edit', object)` displays the Requirements dialog box for `object` and returns the handle of the dialog box.

`rmi('objCopy', object)` resets the globally unique identifier for `object`, preserving its requirement links.

Requirement Links Data Structure

Requirement links are represented using a MATLAB structure array with the following fields:

- `doc` — a string identifying the requirements document, equivalent to the **Document** field of the Requirements dialog box.
- `id` — a string defining a particular location in the requirements document. The first character in the string specifies the type of

identifier that follows. Valid characters that can appear at the beginning of the string are

Character	Identifier	Example
?	Search text, the first occurrence of which is located in the requirements document	'?Requirement 1'
@	Named item, such as a bookmark in a Word document or an anchor in an HTML document	'@my_req'
#	Page or item number	'#21'
>	Line number	'>3156'
\$	Worksheet range in a spreadsheet	'\$A2:C5'

- **linked** — a Boolean value specifying whether the requirement link is accessible for report generation and highlighting. The default value is 1 (true), specifying that the RMI can highlight the model object and include its requirement link in generated reports.
- **description** — a string describing the requirement, equivalent to the **Description** field of the Requirements dialog box.
- **keywords** — an optional string supplementing description, equivalent to the **User tag** field of the Requirements dialog box.
- **reqsys** — a string identifying the link type registration name. This field displays 'other' for built-in link types.

Purpose	Start Requirements Management Interface
Syntax	<code>rminav</code>
Description	<p><code>rminav</code> starts the Requirements Management Interface Navigator window.</p> <p>If you specified <code>reqsys = 'OTHERS'</code> in the MATLAB file <code>reqmgropts.m</code>, the standard version of the Requirements Management Interface Navigator window opens. You can associate requirements documents written in HTML, Microsoft Word, or Microsoft Excel with Simulink models, Stateflow diagrams, and MATLAB M-files.</p> <p>If you specified <code>reqsys = 'DOORS'</code> in <code>reqmgropts.m</code>, the DOORS version of the Requirements Management Interface Navigator window opens. You can associate DOORS requirements with Simulink models, Stateflow diagrams, and MATLAB M-files.</p> <p>To associate DOORS requirements with MATLAB objects, you must start MATLAB with the <code>/automation</code> option.</p>

sigrangeinfo

Purpose Display signal range coverage information for model object

Syntax
`[min, max] = sigrangeinfo(cvdo, object)`
`[min, max] = sigrangeinfo(cvdo, object, portID)`

Description `[min, max] = sigrangeinfo(cvdo, object)` returns the minimum and maximum signal values output by the model component object within the cvdata object cvdo. See “Specifying a Model Object” on page 8-52 for more information about the object argument. If object outputs a vector, min and max are also vectors.

`[min, max] = sigrangeinfo(cvdo, object, portID)` returns the minimum and maximum signal values associated with the output port portID of the Simulink block object.

Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
BlockPath	Full path to a Simulink model or block
BlockHandle	Handle to a Simulink model or block
sObj	Handle to a Simulink API object
sfID	Stateflow ID
sfObj	Handle to a Stateflow API object
{BlockPath, sfID}	Cell array with the path to a Stateflow block and the ID of an object contained in that chart

Object Specification	Description
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable signal range coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.sigrange = 1;
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the signal range for the `Product` block.

```
blk_handle = get_param([mdl, '/Product'], 'Handle');
[minVal, maxVal] = sigrangeinfo(data, blk_handle)
```

tableinfo

Purpose

Display lookup table coverage information for model object

Syntax

```
coverage = tableinfo(cvdo, object)
coverage = tableinfo(cvdo, object, ignore_descendants)
[coverage, exeCounts] = tableinfo(cvdo, object)
[coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)
```

Description

`coverage = tableinfo(cvdo, object)` returns lookup table coverage results from the cvdata object `cvdo` for the model component specified by `object`. `object` is the full path or handle to a Simulink lookup table block or a model containing such a block. The value of `coverage` is a two-element vector of form `[covered_intervals total_intervals]`, the elements of which are defined as follows:

- `covered_intervals` — the number of interpolation/extrapolation intervals satisfied for `object`
- `total_intervals` — the total number of interpolation/extrapolation intervals for `object`

Note `coverage` is empty if `cvdo` does not contain lookup table coverage results for `object`.

`coverage = tableinfo(cvdo, object, ignore_descendants)` returns lookup table coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, exeCounts] = tableinfo(cvdo, object)` returns lookup table coverage results and the execution count for each interpolation/extrapolation interval in the lookup table block specified by `object`. `exeCounts` is an array having the same dimensionality as the lookup table block; however, its size has been extended to allow for the lookup table extrapolation intervals.

`[coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)` returns lookup table coverage results, the execution count for

each interpolation/extrapolation interval, and the execution counts for breakpoint equality. `brkEquality` is a cell array containing vectors that identify the number of times in a simulation the lookup table block input was equivalent to a breakpoint value. Each vector represents the breakpoints along a different lookup table dimension.

Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable lookup table coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
testObj.settings.tableExec = 1;  
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the lookup table coverage results for the Gain Table block (in the Gain subsystem) and determine its percentage of interpolation/extrapolation intervals covered.

```
blk_handle = get_param([mdl, '/Gain/Gain Table'], 'Handle');  
cov = tableinfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

tableinfo

Blocks — Alphabetical List

System Requirements

Purpose List system requirements in Simulink diagrams

Library Simulink Verification and Validation

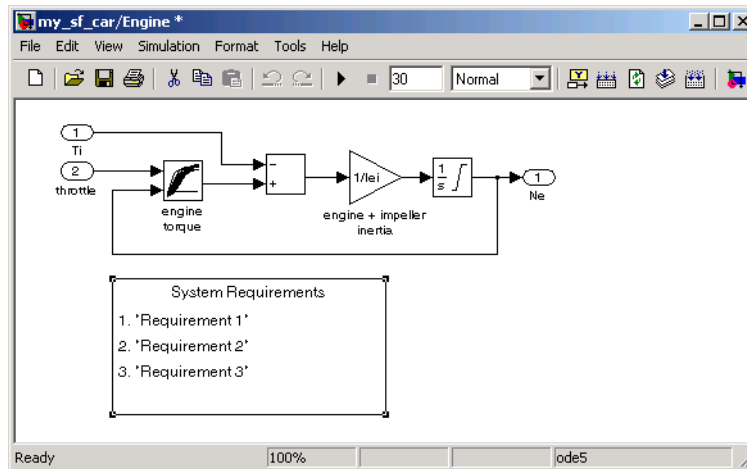
Description



The System Requirements block lists all the system requirements associated with the model or subsystem depicted in the current diagram. It does not list requirements associated with individual blocks in the diagram.

You can place this block anywhere in a diagram. It is not connected to other Simulink blocks. You cannot have more than one System Requirements block in a diagram.

When you drag the System Requirements block from the library browser into your Simulink diagram, it is automatically populated with the system requirements, as shown.



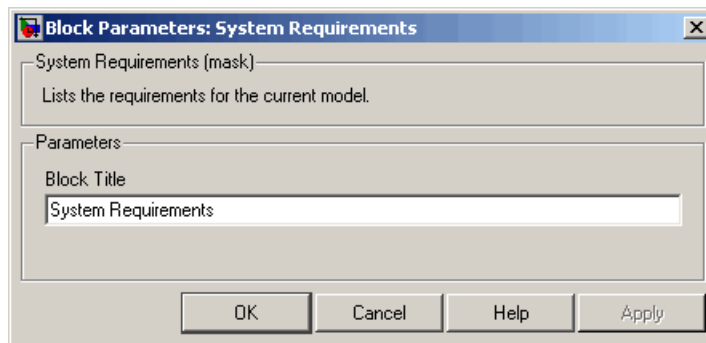
Each of the listed requirements is an active link to the actual requirements document. When you double-click on a requirement name, the associated requirements document opens in its editor window, scrolled to the target location.

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

For more information on using the System Requirements block, see “Displaying the System Requirements in a Diagram” on page 2-50.

Dialog Box and Parameters

To access the Block Parameters dialog box for the System Requirements block, right-click on the System Requirements block and, from the resulting pop-up menu, select **Mask Parameters**. The Block Parameters dialog box opens, as shown.



The Block Parameters dialog box for the System Requirements block contains one parameter.

Block Title

The title of the system requirements list in the diagram. The default title is System Requirements. You can type a customized title, for example, Engine Requirements.

System Requirements

Model Advisor Checks

Simulink Verification and Validation Checks (p. 10-2) Describes Model Advisor checks for verification and validation

Simulink Verification and Validation Checks

In this section...

“Modeling Standards Overview” on page 10-3

“DO-178B Checks Overview” on page 10-3

“Check safety-related optimization settings” on page 10-3

“Check safety-related diagnostic settings for solvers” on page 10-7

“Check safety-related diagnostic settings for sample time” on page 10-9

“Check safety-related diagnostic settings for signal data” on page 10-11

“Check safety-related diagnostic settings for parameters” on page 10-14

“Check safety-related diagnostic settings for data used for debugging” on page 10-16

“Check safety-related diagnostic settings for data store memory” on page 10-17

“Check safety-related diagnostic settings for type conversions” on page 10-19

“Check safety-related diagnostic settings for signal connectivity” on page 10-20

“Check safety-related diagnostic settings for bus connectivity” on page 10-22

“Check safety-related diagnostic settings that apply to function-call connectivity” on page 10-24

“Check safety-related diagnostic settings for compatibility” on page 10-25

“Check safety-related diagnostic setting for model referencing” on page 10-26

“Check safety-related model referencing settings” on page 10-29

“Check safety-related code generation settings” on page 10-31

“Check for proper usage of For Iterator blocks” on page 10-37

“Check for proper usage of While Iterator blocks” on page 10-37

“Display model version information” on page 10-38

In this section...

“Check for proper usage of blocks that compute absolute values” on page 10-39

“Check for proper usage of Relational Operator blocks” on page 10-40

Modeling Standards Overview

Modeling standards checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements or MathWorks Automotive Advisory Board (MAAB) modeling guidelines.

See Also

- Consulting Model Advisor
- Simulink Checks
- Real-Time Workshop Checks

DO-178B Checks Overview

DO-178B checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements.

See Also

- Consulting Model Advisor
- Simulink Checks
- Real-Time Workshop Checks

Check safety-related optimization settings

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-critical application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Results and Recommended Actions

Condition	Recommended Action
<p>Block reduction optimization is on. This optimization can remove blocks from generated code, resulting in requirements with no associated code and violations for traceability requirements. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Clear the Block reduction check box on the Optimization pane of the Configuration Parameters dialog box or set the parameter BlockReduction to 'off'.</p>
<p>Conditional input branch execution is on. Because the model coverage tool does not account for this optimization, the optimization can result in the tool reporting 100% model coverage while coverage for the code using the same test cases can be less than 100%. (See DO-178B, Section 6.4.4.2 – Test coverage of software structure is achieved.)</p>	<p>Clear the Conditional input branch execution check box on the Optimization pane of the Configuration Parameters dialog box or set the parameter ConditionallyExecuteInputs to off.</p>
<p>Implementation of logic signals as Boolean data is off. Strong data typing is recommended for safety-critical code. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards, DO-178B, Section 6.3.2e – Low-level requirements conform to standards, and MISRA-C 2004, Rule 12.6.)</p>	<p>Select Implement logic signals as boolean data (vs. double) on the Optimization pane of the Configuration Parameters dialog box or set the parameter BooleanDataType to on.</p>

Condition	Recommended Action
<p>The model includes blocks that depend on elapsed or absolute time and is configured to minimize the amount of memory allocated for the timers. Such a configuration limits the number of days the application can execute before a timer overflow occurs. Many aerospace products are powered on continuously and timers should not assume a limited lifespan. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 12.11.)</p>	<p>Set Application lifespan (days) on the Optimization pane of the Configuration Parameters dialog box or the parameter <code>LifeSpan</code> to <code>inf</code>.</p>
<p>The optimization that ignores integer downcasts in folded expressions is on. This optimization can remove blocks that typecast data from generated code, resulting in requirements with no associated code and violations for traceability requirements. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 10.1.)</p>	<p>Clear the Ignore integer downcasts in folded expressions check box on the Optimization pane of the Configuration Parameters dialog box or set the parameter <code>EnforceIntegerDowncast</code> to <code>off</code>.</p>
<p>The optimization that suppresses the generation of initialization code for root-level inports and outports that are set to zero is on. For safety-critical code, you should explicitly initialize all variables. (See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA-C 2004, Rule 9.1.)</p>	<p>Clear the Remove root level I/O zero initialization check box on the Optimization pane of the Configuration Parameters dialog box or set the parameter <code>ZeroExternalMemoryAtStartup</code> to <code>on</code>. Alternatively, integrate external, hand-written code that initializes all I/O variables to zero explicitly.</p>

Condition	Recommended Action
<p>The optimization that suppresses the generation of initialization code for internal work structures, such as block states and block outputs that are set to zero, is on. For safety-critical code, you should explicitly initialize all variables. (See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA-C 2004, Rule 9.1.)</p>	<p>Clear the Remove internal state zero initialization check box on the Optimization pane of the Configuration Parameters dialog box or set the parameter <code>ZeroInternalMemoryAtStartup</code> to on. Alternatively, integrate external, hand-written code that initializes all state variables to zero explicitly.</p>
<p>The optimization that suppresses generation of code resulting from floating-point to integer conversions that wrap out-of-range values is off. You must avoid overflows for safety-critical code. When this optimization is off and your model includes blocks that disable the Saturate on overflow parameter, the code generator wraps out-of-range values for those blocks. This can result in unreachable, and therefore, The optimization that specifies whether to generate code that guards against division by zero for fixed-point data is on. You must avoid division-by-zero exceptions in safety-critical code. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 21.1.)</p>	<p>Select Remove code from floating-point to integer conversions that wraps out-of-range values on the Optimization pane of the Configuration Parameters dialog box or set the parameter <code>EfficientFloat2IntCast</code> to on.</p> <p>Clear the Remove code that protects against division arithmetic exceptions check box on the Optimization pane of the Configuration Parameters dialog box or set the parameter <code>NoFixptDivByZeroProtection</code> to off.</p>

See Also

- Optimization Pane in the Simulink graphical user interface documentation
- Optimizing a Model for Code Generation in the Real-Time Workshop documentation

- Tips for Optimizing the Generated Code in the Real-Time Workshop Embedded Coder documentation

Check safety-related diagnostic settings for solvers

Check model configuration for diagnostic settings that apply to solvers and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to solvers are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic for detecting automatic breakage of algebraic loops is set to none or warning. The breaking of algebraic loops can affect the predictability of the order of block execution. For safety-critical applications, a model developer needs to know when such breaks occur. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Algebraic loop on the Diagnostics > Solver pane of the Configuration Parameters dialog box or the parameter <code>AlgebraicLoopMsg</code> to error. Consider breaking such loops explicitly with Unit Delay blocks to ensure that execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.</p>
<p>The diagnostic for detecting automatic breakage of algebraic loops for Model blocks, atomic subsystems, and enabled subsystems is set to none or warning. The breaking of algebraic loops can affect the predictability of the order of block execution. For safety-critical applications, a model developer needs to know when such breaks occur. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Minimize algebraic loop on the Diagnostics > Solver pane of the Configuration Parameters dialog box or the parameter <code>ArtificialAlgebraicLoopMsg</code> to error. Consider breaking such loops explicitly with Unit Delay blocks to ensure that execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.</p>

Condition	Recommended Action
<p>The diagnostic for detecting potential conflict in block execution order is set to none or warning. For safety-critical applications, block execution order must be predictable. A model developer needs to know when conflicting block priorities exist. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Block priority violation on the Diagnostics > Solver pane of the Configuration Parameters dialog box or the parameter BlockPriorityViolationMsg to error.</p>
<p>The diagnostic for detecting whether a model contains an S-function that has not been specified explicitly to inherit sample time is set to none or warning. These settings can result in unpredictable behavior. A model developer needs to know when such an S-function exists in a model so it can be modified to produce predictable behavior. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Unspecified inheritability of sample times on the Diagnostics > Solver pane of the Configuration Parameters dialog box or the parameter UnknownTs1nhSupMsg to error.</p>

Condition	Recommended Action
<p>The diagnostic for detecting whether Simulink automatically modifies the solver, step size, or simulation stop time is set to none or warning. Such changes can affect the operation of generated code. For safety-critical applications, it is better to detect such changes so a model developer can explicitly set the parameters to known values. (See DO-178B, Section 6.3.3e – Software architecture conforms to</p>	<p>Set Automatic solver parameter selection on the Diagnostics > Solver pane of the Configuration Parameters dialog box or the parameter SolverPrmCheckMsg to error.</p>
<p>The diagnostic for detecting when a name is used for more than one state in the model is set to none. State names within a model should be unique. For safety-critical applications, it is better to detect name clashes so a model developer can correct them. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set State name clash on the Diagnostics > Solver pane of the Configuration Parameters dialog box or the parameter StateNameClashWarn to warning.</p>

See Also

- Diagnostics Pane: Solver in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

Check safety-related diagnostic settings for sample time

Check model configuration for diagnostic settings that apply to sample time and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to sample times are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic for detecting when a source block, such as a Sine Wave block, inherits a sample time (specified as -1) is set to none or warning. The use of inherited sample times for a source block can result in unpredictable execution rates for the source block and blocks connected to it. For safety-critical applications, source blocks should have explicit sample times to prevent incorrect execution sequencing. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Source block specifies -1 sample time on the Diagnostics > Sample Time pane of the Configuration Parameters dialog box or the parameter <code>InheritedTsInSrcMsg</code> to error.</p>
<p>The diagnostic for detecting whether the input for a discrete block, such as the Unit Delay block, is a continuous signal is set to none or warning. Signals with continuous sample times should not be used for embedded real-time code. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Discrete used as continuous on the Diagnostics > Sample Time pane of the Configuration Parameters dialog box or the parameter <code>DiscreteInheritContinuousMsg</code> to error.</p>
<p>The diagnostic for detecting invalid rate transitions between two blocks operating in multitasking mode is set to none or warning. Such rate transitions should not be used for embedded real-time code. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Multitask rate transition on the Diagnostics > Sample Time pane of the Configuration Parameters dialog box or the parameter <code>MultiTaskRateTransMsg</code> to error.</p>

Condition	Recommended Action
<p>The diagnostic for detecting subsystems that can cause data corruption or nondeterministic behavior is set to none or warning. This diagnostic detects whether conditionally executed multirate subsystems (enabled, triggered, or function call subsystems) operate in multitasking mode. Such subsystems can corrupt data and behave unpredictably in real-time environments that allow preemption. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Multitask conditionally executed subsystem on the Diagnostics > Sample Time pane of the Configuration Parameters dialog box or the parameter <code>MultiTaskCondExecSysMsg</code> to error.</p>
<p>The diagnostic for checking sample time consistency between a Signal Specification block and the connected destination block is set to none or warning. An over-specified sample time can result in an unpredictable execution rate. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Enforce sample times specified by Signal Specification blocks on the Diagnostics > Sample Time pane of the Configuration Parameters dialog box or the parameter <code>SigSpecEnsureSampleTimeMsg</code> to error.</p>

See Also

- Diagnostics Pane: Sample Time in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

Check safety-related diagnostic settings for signal data

Check model configuration for diagnostic settings that apply to signal data and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal data are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that specifies how Simulink resolves signals associated with Simulink.Signal objects in the MATLAB workspace is set to Explicit and implicit or Explicit and warn implicit. For safety-critical applications, model developers should be required to define signal resolution explicitly. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Signal resolution on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter SignalResolutionControl to Explicit only. This provides predictable operation by requiring users to define each signal and block setting that must resolve to Simulink.Signal objects in the workspace.</p>
<p>The Product block diagnostic that detects a singular matrix while inverting one of its inputs in matrix multiplication mode is set to none or warning. Division by a singular matrix can result in numeric exceptions when executing generated code. This is not acceptable in safety-critical systems. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g - Algorithms are accurate, and MISRA-C 2004, Rule 21.1.)</p>	<p>Set Division by singular matrix on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter CheckMatrixSingularityMsg to error.</p>

Condition	Recommended Action
<p>The diagnostic that detects when Simulink cannot infer the data type of a signal during data type propagation is set to none or warning. For safety-critical applications, model developers must ensure that all data types are specified correctly. (See DO-178B, Section 6.3.1e - High-level requirements conform to standards, DO-178B and Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set Underspecified data types on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter UnderSpecifiedDataTypeMsg to error.</p>
<p>The diagnostic that detects whether the value of a signal or parameter is too large to be represented by the signal or parameter's data type is set to none or warning. Undetected numeric overflows can result in incorrect and unsafe application behavior. (See DO-178B, Section 6.3.1g - Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 21.1.)</p>	<p>Set Detect overflow on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter IntegerOverflowMsg to error.</p>

Condition	Recommended Action
<p>The diagnostic that detects when the value of a block output signal is Inf or NaN at the current time step is set to none or warning. When this type of block output signal condition occurs, numeric exceptions can result, and numeric exceptions are not acceptable in safety-critical applications. (See DO-178B, Section 6.3.1g - Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 21.1.)</p>	<p>Set Inf or NaN block output on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter SignalInfNanChecking to error.</p>
<p>The diagnostic that detects Simulink object names that begin with rt is set to none or warning. This diagnostic prevents name clashes with generated signal names that have an rt prefix. (See DO-178B, Section 6.3.1e - High-level requirements conform to standards, DO-178B and Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set "rt" prefix for identifiers on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter RTPrefix to error.</p>

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

Check safety-related diagnostic settings for parameters

Check model configuration for diagnostic settings that apply to parameters and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to parameters are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects when a parameter downcast occurs is set to none or warning. A downcast to a lower signal range can result in numeric overflows of parameters, resulting in incorrect and unsafe behavior. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 21.1.)</p>	<p>Set Detect downcast on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter ParameterDowncastMsg to error.</p>
<p>The diagnostic that detects when a parameter underflow occurs is set to none or warning. When the data type of a parameter does not have sufficient resolution, the parameter value is zero instead of the specified value. This can lead to incorrect operation of generated code. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 21.1.)</p>	<p>Set Detect underflow on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter ParameterUnderflowMsg to error.</p>
<p>The diagnostic that detects when a parameter overflow occurs is set to none or warning. Numeric overflows can result in incorrect and unsafe application behavior and should be detected and corrected in safety-critical applications. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rule 21.1.)</p>	<p>Set Detect overflow on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter ParameterOverflowMsg to error.</p>

Condition	Recommended Action
<p>The diagnostic that detects when a parameter loses precision is set to none or warning. Not detecting such errors can result in a parameter being set to an incorrect value in the generated code. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rules 10.1, 10.2, 10.3, and 10.4.)</p>	<p>Set Detect precision loss on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter <code>ParameterPrecisionLossMsg</code> to error.</p>
<p>The diagnostic that detects when an expression with tunable variables is reduced to its numerical equivalent is set to none or warning. This can result in a tunable parameter unexpectedly not being tunable in generated code. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)</p>	<p>Set Detect loss of tunability on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter <code>ParameterTunabilityLossMsg</code> to error.</p>

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

Check safety-related diagnostic settings for data used for debugging

Check model configuration for diagnostic settings that apply to data used for debugging and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to debugging are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that enables model verification blocks is set to Use local settings or Enable all. Such blocks should be disabled because they are assertion blocks, which are for verification only. Model developers should not use assertions in embedded code. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards, DO-178B and Section 6.3.2e – Low-level requirements conform to standards.)	Set Model Verification block enabling on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter AssertControl to Disable All.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

Check safety-related diagnostic settings for data store memory

Check model configuration for diagnostic settings that apply to data store memory and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to data store memory are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects whether the model attempts to read data from a data store in which it has not stored data in the current time step is set to a value other than Enable all as errors. Reading data before it is written can result in use of stale data or data that is not initialized. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Detect read before write on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter ReadBeforeWriteMsg to Enable all as errors.</p>
<p>The diagnostic that detects whether the model attempts to store data in a data store after previously reading data from it in the current time step is set to a value other than Enable all as errors. Writing data after it is read can result in use of stale or incorrect data. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Detect write after read on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter WriteAfterReadMsg to Enable all as errors.</p>
<p>The diagnostic that detects whether the model attempts to store data in a data store twice in succession in the current time step is set to a value other than Enable all as errors. Writing data twice in one time step can result in unpredictable data. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Detect write after write on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter WriteAfterWriteMsg to Enable all as errors.</p>
<p>The diagnostic that detects when one task reads data from a Data Store Memory block to which another task writes data is set to none or warning. Reading or writing data in different tasks in multitask mode can result in corrupted or unpredictable data. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Multitask data store on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or the parameter MultiTaskDSMMsg to error.</p>

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

Check safety-related diagnostic settings for type conversions

Check model configuration for diagnostic settings that apply to type conversions and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to type conversions are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects Data Type Conversion blocks used where no type conversion is necessary is set to none. Simulink might remove unnecessary Data Type Conversion blocks from generated code. This can result in requirements with no corresponding code. The removal of such blocks need to be detected so model developers can remove the unnecessary blocks explicitly. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)</p>	<p>Set Unnecessary type conversions on the Diagnostics > Type Conversion pane of the Configuration Parameters dialog box or the parameter <code>UnnecessaryDatatypeConvMsg</code> to warning.</p>

Condition	Recommended Action
<p>The diagnostic that detects vector-to-matrix or matrix-to-vector conversions at block inputs is set to none or warning. When Simulink automatically converts between vector and matrix dimensions, unintended operations or unpredictable behavior can occur. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)</p>	<p>Set Vector/matrix block input conversion on the Diagnostics > Type Conversion pane of the Configuration Parameters dialog box or the parameter <code>VectorMatrixConversionMsg</code> to error.</p>
<p>The diagnostic that detects when a 32-bit integer value is converted to a floating-point value is set to none. This type of conversion can result in a loss of precision due to truncation of the least significant bits for large integer values. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA-C 2004, Rules 10.1, 10.2, 10.3, and 10.4.)</p>	<p>Set 32-bit integer to single precision float conversion on the Diagnostics > Type Conversion pane of the Configuration Parameters dialog box or the parameter <code>Int32ToFloatConvMsg</code> to warning.</p>

See Also

- Diagnostics Pane: Type Conversion in the Simulink graphical user interface documentation
- Data Type Conversion block in the Simulink reference documentation

Check safety-related diagnostic settings for signal connectivity

Check model configuration for diagnostic settings that apply to signal connectivity and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal connectivity are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects virtual signals that have a common source signal but different labels is set to none or warning. This diagnostic pertains to virtual signals only and has no effect on generated code. However, signal label mismatches can lead to confusion during model reviews. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards and DO-178B, Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set Signal label mismatch on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter SignalLabelMismatchMsg to error.</p>
<p>The diagnostic that detects when the model contains a block with an unconnected input signal is set to none or warning. This must be detected because code is not generated for unconnected block inputs. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards and DO-178B, Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set Unconnected block input ports on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter UnconnectedInputMsg to error.</p>

Condition	Recommended Action
<p>The diagnostic that detects when the model contains a block with an unconnected output signal is set to none or warning. This must be detected because dead code can result from unconnected block output signals. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards and DO-178B, Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set Unconnected block output ports on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter UnconnectedOutputMsg to error.</p>
<p>The diagnostic that detects unconnected signal lines and unmatched Goto or From blocks is set to none or warning. This must be detected because code is not generated for unconnected lines. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards and DO-178B, Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set Unconnected line on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter UnconnectedLineMsg to error.</p>

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- Signal Basics in the Simulink documentation

Check safety-related diagnostic settings for bus connectivity

Check model configuration for diagnostic settings that apply to bus connectivity and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to bus connectivity are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects whether a Model block's root Output block is connected to a bus but does not specify a bus object is set to none or warning. For a bus signal to cross a model boundary, the signal must be defined as a bus object to ensure compatibility with higher level models that use a model as a reference model. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Unspecified bus object at root Output block on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter <code>RootOutputRequireBusObject</code> to error.</p>
<p>The diagnostic that detects whether the name of a bus element matches the name specified by the corresponding bus object is set to none or warning. This diagnostic prevents the use of incompatible buses in a bus-capable block such that the output names are inconsistent. (See DO-178B, Section 6.3.3b - Software architecture is consistent.)</p>	<p>Set Element name mismatch on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter <code>BusObjectLabelMismatch</code> to error.</p>
<p>The diagnostic that detects when some blocks treat a signal as a mux/vector, while other blocks treat the signal as a bus, is set to none or warning. When Simulink automatically converts a muxed signal to a bus, it is possible for an unintended operation or unpredictable behavior to occur. (See DO-178B, Section 6.3.3b - Software architecture is consistent.)</p>	<p>Set Mux blocks used to create bus signals on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter <code>StrictBusMsg</code> to error. You can use the Model Advisor or the <code>s1_replace_mux</code> utility function to replace all Mux blocks used as bus creators with a Bus Creator block.</p>

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- `Simulink.Bus` in the Simulink reference documentation

Check safety-related diagnostic settings that apply to function-call connectivity

Check model configuration for diagnostic settings that apply to function-call connectivity and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to function-call connectivity are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects incorrect use of a function-call subsystem is set to none or warning. If this condition is undetected, incorrect code might be generated. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set Invalid function-call connection on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter <code>InvalidFcnCallConMsg</code> to error.
The diagnostic that specifies whether Simulink has to compute a function-call subsystem's inputs directly or indirectly while executing the subsystem is set to Use local settings or Disable all. This diagnostic detects unpredictable data coupling between a function-call subsystem and the subsystem's inputs. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set Context-dependent inputs on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or the parameter <code>FcnCallInpInsideContextMsg</code> to Enable all.

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation

Check safety-related diagnostic settings for compatibility

Check model configuration for diagnostic settings that affect compatibility and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to compatibility are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects when a block has not been upgraded to use features of the current release is set to none or warning. An S-function written for an earlier version might not be compatible with the current version and generated code could operate incorrectly. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set S-function upgrades needed on the Diagnostics > Compatibility pane of the Configuration Parameters dialog box or the parameter SFcnCompatibilityMsg to error.</p>
<p>The Check undefined subsystem initial output diagnostic is off. This diagnostic specifies whether Simulink displays a warning if the model contains a conditionally executed subsystem in which a block with a specified initial condition drives an Outport block with an undefined initial condition. A conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic. (See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA-C 2004, Rule 9.1.)</p>	<p>Set Check undefined subsystem initial output on the Diagnostics > Compatibility pane of the Configuration Parameters dialog box or the parameter CheckSSInitialOutputMsg to on.</p>

Condition	Recommended Action
<p>The diagnostic that detects potential initial output differences from earlier releases is off. An output of a conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic. (See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA-C 2004, Rule 9.1.)</p>	<p>Set Check preactivation output of execution context on the Diagnostics > Compatibility pane of the Configuration Parameters dialog box or the parameter <code>CheckExecutionContextPreStartOutputMsg</code> to on.</p>
<p>The diagnostic that detects potential output differences from earlier releases is off. An output of a conditionally executed subsystem could have an output that is not initialized and feeds into a block with a tunable parameter. If undetected, this condition can cause the behavior of the downstream block to be nondeterministic. (See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA-C 2004, Rule 9.1.)</p>	<p>Set Check runtime output of execution context on the Diagnostics > Compatibility pane of the Configuration Parameters dialog box or the parameter <code>CheckExecutionContextRuntimeOutputMsg</code> to on.</p>

See Also

- Diagnosing Simulation Errors
- Diagnostics Pane: Compatibility

Check safety-related diagnostic setting for model referencing

Check model configuration for diagnostic settings that apply to model referencing and can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to model referencing are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects a mismatch between the version of the model that creates or refreshes a Model block and the referenced model's current version is set to none or warning. The detection occurs during load and update operations. Get the latest version of the referenced model from the software configuration management system, rather than using an older version. Using an older version can produce incorrect simulation results and mismatches between simulation and target code operation. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Model block version mismatch on the Diagnostics > Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>ModelReferenceVersionMismatchMessage</code> to error.</p>
<p>The diagnostic that detects port and parameter mismatches during model loading and updating is set to none or warning. If undetected, such mismatches can lead to incorrect simulation results because the parent and referenced models have different interfaces. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Port and parameter mismatch on the Diagnostics > Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>ModelReferenceIOMismatchMessage</code> to error.</p>

Condition	Recommended Action
<p>The Model configuration mismatch diagnostic is set to none or error. This diagnostic checks whether the configuration parameters of a model referenced by the current model match the current model's configuration parameters or are inappropriate for a referenced model. Some diagnostics for referenced models are not supported in simulation mode. Setting this diagnostic to error can prevent simulations from running. Some differences in configurations can lead to incorrect simulation results and mismatches between simulation and target code generation. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Model configuration mismatch on the Diagnostics > Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>ModelReferenceCSMismatchMessage</code> to warning.</p>

Condition	Recommended Action
<p>The diagnostic that detects invalid internal connections to the current model's root-level Inport and Outport blocks is set to none or warning. When this condition is detected, Simulink might automatically insert hidden blocks into the model to correct the condition. The hidden blocks can result in generated code that has no traceable requirements. Setting the diagnostic to error forces model developers to correct the referenced models manually. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Invalid root Inport/Outport block connection on the Diagnostics > Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>ModelReferenceIOMessage</code> to error.</p>
<p>The diagnostic that detects whether To Workspace or Scope blocks are logging data in a referenced model is set to none or warning. Because To Workspace and Scope blocks are for debugging and not for embedded code, it is best to detect the condition so model developers can correct it. (See DO-178B, Section 6.3.1d – High-level requirements are verifiable and DO-178B, Section 6.3.2d – Low-level requirements are verifiable.)</p>	<p>Set Unsupported data logging on the Diagnostics > Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>ModelReferenceDataLoggingMessage</code> to error.</p>

See Also

- Diagnosing Simulation Errors
- Diagnostics Pane: Model Referencing

Check safety-related model referencing settings

Check model configuration for model referencing settings that can impact safety.

Description

This check verifies that model configuration parameters for model referencing are set optimally for generating code for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The referenced model is configured such that its target is rebuilt whenever you update, simulate, or generate code for the model, or if Simulink detects any changes in known dependencies. These configuration settings can result in unnecessary regeneration of the code, resulting in changing only the date of the file and slowing down the build process when using model references. (See DO-178B, Section 6.3.1b – High-level requirements are accurate and consistent and DO-178B, Section 6.3.2b – Low-level requirements are accurate and consistent.)</p>	<p>Set Rebuild options on the Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>UpdateModelReferenceTargets</code> to <code>Never</code> or <code>If any changes detected</code>.</p>
<p>The diagnostic that detects whether a target needs to be rebuilt is set to <code>None</code> or <code>Warn</code> if targets require rebuild. For safety-critical applications, an error should alert model developers that the parent and referenced models are inconsistent. This diagnostic parameter is available only if Rebuild options is set to <code>Never</code>. (See DO-178B, Section 6.3.1b – High-level requirements are accurate and consistent and DO-178B, Section 6.3.2b – Low-level requirements are accurate and consistent.)</p>	<p>Set Never rebuild targets diagnostic on the Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>CheckModelReferenceTargetMessage</code> to <code>Error</code> if targets require rebuild.</p>

Condition	Recommended Action
<p>The ability to pass scalar root input by value is on. This capability should be off because scalar values can change during a time step and result in unpredictable data. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Pass scalar root inputs by value on the Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>ModelReferencePassRootInputsByReference</code> to off.</p>
<p>The model is configured to minimize algebraic loop occurrences. This is incompatible with the recommended setting of Single output/update function for embedded systems code. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Minimize algebraic loop occurrences on the Model Referencing pane of the Configuration Parameters dialog box or the parameter <code>ModelReferenceMinAlgLoopOccurrences</code> to off.</p>

See Also

- Model Dependencies in the Simulink documentation
- Model Referencing Pane in the Simulink graphical user interface documentation

Check safety-related code generation settings

Check model configuration for code generation settings that can impact safety.

Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-critical application.

Results and Recommended Actions

Condition	Recommended Action
<p>The option to include comments in the generated code is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Include comments on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or the parameter <code>GenerateComments</code> to on.</p>
<p>The option to include comments that describe the code for blocks is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Simulink block comments on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or the parameter <code>SimulinkBlockComments</code> to on.</p>
<p>The option to include comments that describe the code for blocks eliminated from a model is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Show eliminated blocks on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or the parameter <code>ShowEliminatedStatement</code> to on.</p>
<p>The option to include the names of parameter variables and source blocks as comments in the model parameter structure declaration in <code>model_prm.h</code> is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Verbose comments for SimulinkGlobal storage class on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or the parameter <code>ForceParamTrailComments</code> to on.</p>
<p>The option to include requirement descriptions assigned to Simulink blocks as comments is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Requirements in block comments on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or the parameter <code>ReqInCode</code> to on.</p>

Condition	Recommended Action
<p>The option to generate nonfinite data and operations is on. Support for nonfinite numbers is inappropriate for real-time safety-critical systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set Support: non-finite numbers on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter SupportNonFinite to off.</p>
<p>The option to generate and maintain integer counters for absolute and elapsed time is on. Support for absolute time is inappropriate for real-time safety-critical systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set Support: absolute time on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter SupportAbsoluteTime to off.</p>
<p>The option to generate code for blocks that use continuous time is on. Support for continuous time is inappropriate for real-time safety-critical systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set Support: continuous time on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter SupportContinuousTime to off.</p>
<p>The option to generate code for noninlined S-functions is on. This option requires support of nonfinite numbers, which is inappropriate for real-time safety-critical systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set Support: non-inlined S-functions on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter SupportNonInlinedSFcns to off.</p>

Condition	Recommended Action
<p>The option to generate model function calls compatible with the main program module of the GRT target is on. This option is inappropriate for real-time safety-critical systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set GRT compatible call interface on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter <code>GRTInterface</code> to off.</p>
<p>The option to generate the <code>model_update</code> function is off. Having a single call to the output and update functions simplifies the interface to the real-time operating system (RTOS) and simplifies verification of the generated code. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set Single output/update function on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter <code>CombineOutputUpdateFcns</code> to on.</p>
<p>The option to generate the <code>model_terminate</code> function is on. This function deallocates dynamic memory, which is not appropriate for real-time safety-critical systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set Terminate function required on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter <code>IncludeMdlTerminateFcn</code> to off.</p>

Condition	Recommended Action
<p>The option to log or monitor error status is off. If you do not select this option, Real-Time Workshop generates extra code that might not be reachable for testing. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set Suppress error status in real-time model data structure on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter SuppressErrorStatus to on.</p>
<p>MAT-file logging is enabled. This option adds extra code for logging test points to a MAT-file, which is not supported by embedded targets. Use this option only in test harnesses. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set MAT-file logging on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or the parameter MatFileLogging to on.</p>
<p>The option that specifies the style for parenthesis usage is set to Minimum (Rely on C/C++ operators precedence) or to Nominal (Optimize for readability). For safety-critical applications, explicitly specify precedence with parentheses. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer, DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer, and MISRA-C 2004, Rule 12.1.)</p>	<p>Set Parenthesis level on the Real-Time Workshop > Code pane of the Configuration Parameters dialog box or the parameter ParenthesisStyle to Maximum (Specify precedence with parentheses).</p>
<p>The option that specifies whether to preserve operand order is off. This option increases the traceability of the generated code. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Preserve operand order in expression on the Real-Time Workshop > Code pane of the Configuration Parameters dialog box or the parameter PreserveExpressionOrder to on.</p>

Condition	Recommended Action
<p>The option that specifies whether to preserve empty primary condition expressions in <code>if</code> statements is off. This option increases the traceability of the generated code. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Preserve condition expression in if statement on the Real-Time Workshop > Code pane of the Configuration Parameters dialog box or the parameter <code>PreserveIfCondition</code> to on.</p>
<p>The minimum number of characters specified for generating name mangling strings is less than four. You can use this option to minimize the likelihood that parameter and signal names will change during code generation when the model changes. This assists with minimizing code differences between file versions, decreasing the effort to perform code reviews. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set Minimum mangle length on the Real-Time Workshop > Symbols pane of the Configuration Parameters dialog box or the parameter <code>MangleLength</code> to a value of 4 or greater.</p>
<p>The maximum number of characters specified for generated function, type definition, and variable names is less than 31. You should set this option based on the maximum value allowed by the compiler being used. The longer the length can be, the easier it is to trace identifiers to the model. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards and DO-178B, Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set Maximum identifier length on the Real-Time Workshop > Symbols pane of the Configuration Parameters dialog box or the parameter <code>MaxIdLength</code> to a value of 31 or greater.</p>

See Also

- Real-Time Workshop Pane: Comments in the Real-Time Workshop reference documentation
- Real-Time Workshop Pane: Symbols in the Real-Time Workshop reference documentation

- Real-Time Workshop Pane: Interface in the Real-Time Workshop reference documentation
- Real-Time Workshop Pane: Code Style in the Real-Time Workshop Embedded Coder reference documentation

Check for proper usage of For Iterator blocks

Check for For Iterator blocks that have variable loops.

Description

This check verifies that a model does not use variable loops with For Iterator blocks.

Results and Recommended Actions

Condition	Recommended Action
The model combines the use of variable iteration values with a For Iterator block. The use of variable for loops can lead to unpredictable execution time and, in the case of external iteration variables, infinite loops. (See DO-178B Section 6.3.1e – High-level requirements conform to standards, DO-178B Section 6.3.2e – Low-level requirements conform to standards, MISRA-C 2004, Rule 13.6.)	<p>To avoid the use of variable for loops, do one of the following</p> <ul style="list-style-type: none"> • Set the Iteration limit source parameter of the For Iterator block to <code>internal</code>. • If the Iteration limit source parameter of the For Iterator block must be <code>external</code>, use a Constant, Probe, or Width block as the source. • Avoid selecting the Set next i (iteration variable) externally parameter of the For Iterator block.

See Also

For Iterator Subsystem block

Check for proper usage of While Iterator blocks

Check for While Iterator blocks that cause infinite loops.

Description

This check verifies that a model does not include infinite loops with While Iterator blocks.

Results and Recommended Actions

Condition	Recommended Action
<p>The model combines the use of a While Iterator block with an unlimited number of iterations. An unlimited number of iterations can lead to infinite loops in real-time code, which can lead to execution time overruns. (See DO-178B Section 6.3.1e – High-level requirements conform to standards, DO-178B Section 6.3.2e – Low-level requirements conform to standards, MISRA-C 2004, Rule 21.1)</p>	<p>To avoid infinite loops:</p> <ul style="list-style-type: none"> • Set the Maximum number of iterations parameter of the While Iterator block to a positive integer value. • Consider selecting the Show iteration number port parameter of the While Iterator block and observe the iteration value during simulation to determine whether the maximum number of iterations is being reached. If the loop reaches the maximum number of iterations, verify whether the output values of the While Iterator block are correct.

See Also

While Iterator Subsystem block

Display model version information

Display model version information in report.

Description

This check displays the following information for the current model:

- Version number
- Author
- Date
- Model checksum

See Also

Validate Generated Code in the Real-Time Workshop documentation

Check for proper usage of blocks that compute absolute values

Check for absolute value blocks that have unreachable code or produce overflows.

Description

This check verifies whether the model includes a block that attempts to compute the absolute value of a Boolean or unsigned integer value.

Results and Recommended Actions

Condition	Recommended Action
<p>The model includes a block that:</p> <ul style="list-style-type: none"> • Computes an absolute value and the input signal of the block is a Boolean value or an unsigned integer. Use of Boolean and unsigned data types result in code that is unreachable and cannot be tested. • Computes an absolute value of a signed integer and Saturate on integer overflow is not selected for that block. Taking the absolute value of full scale negative integer value results in an overflow. <p>(See DO-178B Section 6.3.1d – High-level requirements are verifiable, DO-178B Section 6.3.2d – Low-level requirements are verifiable, DO-178B Section 6.3.1g – Algorithms are accurate, DO-178B Section 6.3.2g – Algorithms are accurate, MISRA-C 2004, Rule 14.1, and MISRA-C 2004, Rule 21.1.)</p>	<ul style="list-style-type: none"> • To avoid unreachable code, change the input to the Absolute Value block to a signed input type. • To avoid overflows, select the Saturate on integer overflow check box of the Absolute Value block.

See Also

Abs block

Check for proper usage of Relational Operator blocks

Check for relational operator blocks that compare data types or equate floating-point types.

Description

This check verifies that a model does not use the == or ~= operator with a relational operator block to compare floating-point signals.

Results and Recommended Actions

Condition	Recommended Action
The model includes a relational operator block that uses the == or ~= operator to compare floating-point signals. Because of floating-point precision issues, the use of these operators on floating-point signals is unreliable. (See DO-178B Section 6.3.1g – Algorithms are accurate, DO-178B Section 6.3.2g – Algorithms are accurate, MISRA-C 2004, Rule 12.6, and MISRA-C 2004, Rule 13.3.)	Change the data type of the signal or rework the model to eliminate the need to use the relational operator block with the == or ~= operator.

See Also

- Relational Operator block
- Compare To Constant block
- Compare To Zero block
- Detect Change block

Examples

Use this list to find examples in the documentation.

Requirements Management Interface

- “Adding Requirement Links to an Object” on page 2-8
- “Viewing Requirements Documents” on page 2-13
- “Making Selection-Based Links” on page 2-22
- “Creating a Custom Link Requirement Type” on page 2-32
- “Viewing Objects with Requirement Links” on page 2-45
- “Generating a Requirements Report” on page 2-48
- “Displaying the System Requirements in a Diagram” on page 2-50
- “Including Requirements with Generated Code” on page 2-56

Requirements Management Interface (DOORS Version)

- “Linking Objects to DOORS Requirements” on page 3-8
- “Synchronizing a Model with DOORS” on page 3-14
- “Navigating from Simulink to DOORS” on page 3-27
- “Navigating from DOORS to Simulink” on page 3-29

Verification Manager

- “Opening the Verification Manager” on page 4-7
- “Enabling and Disabling Model Verification Blocks with the Verification Manager” on page 4-15
- “Using Enabling and Disabling Tools in the Verification Manager” on page 4-20
- “Managing Verification Requirements” on page 4-24

Model Coverage

- “Details Report Section” on page 5-22
- “Decisions Analyzed Table” on page 5-24
- “Conditions Analyzed Table” on page 5-24
- “MC/DC Analysis Table” on page 5-25

“N-Dimensional Lookup Table Report” on page 5-27

“Signal Range Analysis Report” on page 5-33

“Displaying Model Coverage with Model Coloring” on page 5-37

“Creating a Model with Embedded MATLAB Function Block Decisions”
on page 5-53

“Understanding Embedded MATLAB Function Block Model Coverage”
on page 5-56

A

- adding links to requirements 2-8
- adding requirements 2-5
- Assert block appearance 4-19

C

- categorical lists of functions 7-1
- changing links to requirements 2-11
- closing Signal Builder Requirements pane 4-13
- colored diagram model coverage display 5-36
 - enabling 5-36
- condition coverage
 - Embedded MATLAB Function blocks 5-63
 - statements in Embedded MATLAB Function block 5-53
- configuring MATLAB
 - for DOORS version 3-6
- cv.cvdatagroup function
 - reference 7-3 8-5
- cv.cvtestgroup function
 - reference 7-3 8-7
- cv.exit function
 - reference 7-3 8-9
- cv.html function
 - model coverage 5-44
 - reference 7-3 8-10
- cv.load function
 - model coverage 5-45
 - reference 7-3 8-12
- cv.save function
 - model coverage 5-44
 - reference 7-3 8-14
- cv.sim function
 - model coverage 5-43
 - reference 7-3 8-15
- cv.simref function
 - reference 7-3 8-17
- cv.test function
 - model coverage 5-41

reference 7-3 8-19

D

- decision coverage
 - Embedded MATLAB Function blocks 5-63
 - statements in Embedded MATLAB Function blocks 5-52
- demos
 - simcovdemo model coverage demo 5-8
- disabling Model Verification blocks across test groups 4-20
- DOORS
 - additional installation for 3-3
 - starting 3-6
- DOORS Requirements Management Interface
 - block type descriptions 3-17
 - definition for object 3-14
 - from Simulink to DOORS 3-27
 - hierarchical numbers 3-17
 - naming of surrogate exported module 3-17
 - object identifiers 3-17
 - opening the object in Simulink, Stateflow, or MATLAB 3-31
 - overview 3-2
 - saving formal modules 3-20
 - starting MATLAB for 3-6
 - synchronizing models with DOORS 3-14
 - synchronizing objects with DOORS formal module 3-14
 - viewing model elements with requirements 3-25
 - viewing requirements 3-25

E

- Embedded MATLAB Function blocks
 - condition coverage 5-63
 - condition coverage statements 5-53
 - decision coverage 5-63

- decision coverage statements 5-52
- MCDC coverage 5-64
- MCDC coverage statements 5-53
- model coverage 5-52
- model coverage example 5-53
- types of model coverage 5-52

enabling Model Verification blocks across test groups 4-20

F

functions

- categorical lists of 7-1
- cv.cvdatagroup 7-3 8-5
- cv.cvtestgroup 7-3 8-7
- cvexit 7-3 8-9
- cvhtml 7-3 8-10
- cvload 7-3 8-12
- cvsave 7-3 8-14
- cvsim 7-3 8-15
- cvsimref 7-3 8-17
- cvtest 7-3 8-19
- Model Advisor formatting API 7-4
- model coverage 7-3
- rminav 7-2 8-51
- start old Requirements Management Interface 7-2

I

icons for Model Verification blocks in Verification Manager 4-16

installing DOORS 3-3

L

linking model objects to requirements 2-8

Lookup Table block in model coverage report 5-27

Lookup Table model coverage

- n-dimensional 5-32
- three-dimensional example 5-29

two-dimensional example 5-27

M

MCDC coverage

- Embedded MATLAB Function blocks 5-64
- statements in Embedded MATLAB Function blocks 5-53

MCDC table

- condition cases 5-25

Model Advisor formatting API functions 7-4

model coverage

- colored Simulink diagram display 5-36
- colored simulink diagram example 5-37
- commands in MATLAB 5-41
- Embedded MATLAB Function blocks 5-52
- enabling colored diagram display 5-36
- enabling colored simulink diagram display 5-36
- HTML settings 5-16
- introduction 5-3
- Lookup Table block report 5-27
- MCDC table 5-25
- n-dimensional Lookup Table 5-32
- settings in dialog 5-12
- signal range analysis report 5-33
- Summary report section 5-21
- three-dimensional Lookup Table example 5-29
- two-dimensional Lookup Table 5-27
- understanding report 5-21

model coverage demo

- simcovdemo 5-8

model coverage functions 7-3

- cvhtml 5-44
- cvload 5-45
- cvsave 5-44
- cvsim 5-43
- cvtest 5-41

Model Verification blocks

- block appearance 4-17
- disabling for test groups 4-15
- enabling for test groups 4-15
- icons 4-16
- parameter settings 4-3
- using individually 4-2
- models
 - running test cases 5-8
- modifying requirements 2-5

O

- objects
 - linking model objects to requirements 2-8
 - viewing objects with requirements 2-45
- old Requirements Management Interface 7-2
- opening a Signal Builder block 4-9
- operating system requirements 1-3

P

- parameters for Model Verification blocks 4-3

R

- report
 - model coverage HTML options 5-16
 - understanding model coverage report 5-21
- requirements
 - adding 2-5
 - adding to test groups 4-25
 - for Model Verification block settings 4-24
 - for Requirements Management Interface for DOORS 3-2
 - in generated code 2-56
 - linking to model objects 2-8
 - modifying 2-5
 - viewing 2-5
 - viewing for test groups 4-27
 - viewing objects with 2-45
- requirements documents

- editing 2-13
- viewing 2-13
- requirements links
 - editing 2-11
- Requirements Management Interface
 - overview 2-3
- Requirements Management Interface for DOORS
 - block type descriptions 3-17
 - definition of object in DOORS 3-14
 - from Simulink to DOORS 3-27
 - hierarchical numbers 3-17
 - naming of surrogate exported modules 3-17
 - object identifiers 3-17
 - opening the object in Simulink or Stateflow 3-31
 - overview 3-2
 - saving formal modules 3-20
 - starting 3-6
 - starting MATLAB for 3-6
 - synchronizing models with DOORS 3-14
 - synchronizing objects with DOORS formal module 3-14
 - viewing model elements with requirements 3-25
 - viewing requirements 3-25
- Requirements pane for Verification Manager 4-24
- rminav function
 - reference 7-2 8-51

S

- Signal Builder block
 - opening 4-9
- Signal Builder dialog box
 - closing Verification Manager Requirements pane 4-13
- signal range analysis report in model
 - coverage 5-33
- simcovdemo

- model coverage demo 5-8
- starting DOORS 3-6
- starting MATLAB for DOORS 3-6
- starting Requirements Management Interface for DOORS 3-6
- Summary section of model coverage report 5-21
- synchronizing models with DOORS 3-14
- system requirements 1-3
 - MATLAB 1-3
 - Microsoft Excel 1-3
 - Microsoft Word 1-3
 - operating system 1-3
 - Simulink 1-3
 - Stateflow 1-3
 - Telelogic DOORS 1-3

T

- test case commands 5-8
- test groups
 - adding requirements 4-25
 - disabling Model Verification blocks 4-15
 - enabling Model Verification blocks 4-15

- Model Verification blocks enabled across 4-20

V

- verification blocks
 - example of use 4-2
 - icons 4-16
 - requirements for test groups 4-24
 - stopping simulation 4-4
- Verification Manager
 - closing Requirements pane 4-13
 - disabling Model Verification blocks for test groups 4-15
 - enabled/disabled block appearance 4-17
 - enabling Model Verification blocks for test groups 4-15
 - flat display 4-15
 - hierarchical display 4-15
 - icons for Model Verification blocks 4-16
 - opening 4-7
 - Requirements pane 4-24
- viewing objects with requirements 2-45
- viewing requirements 2-5