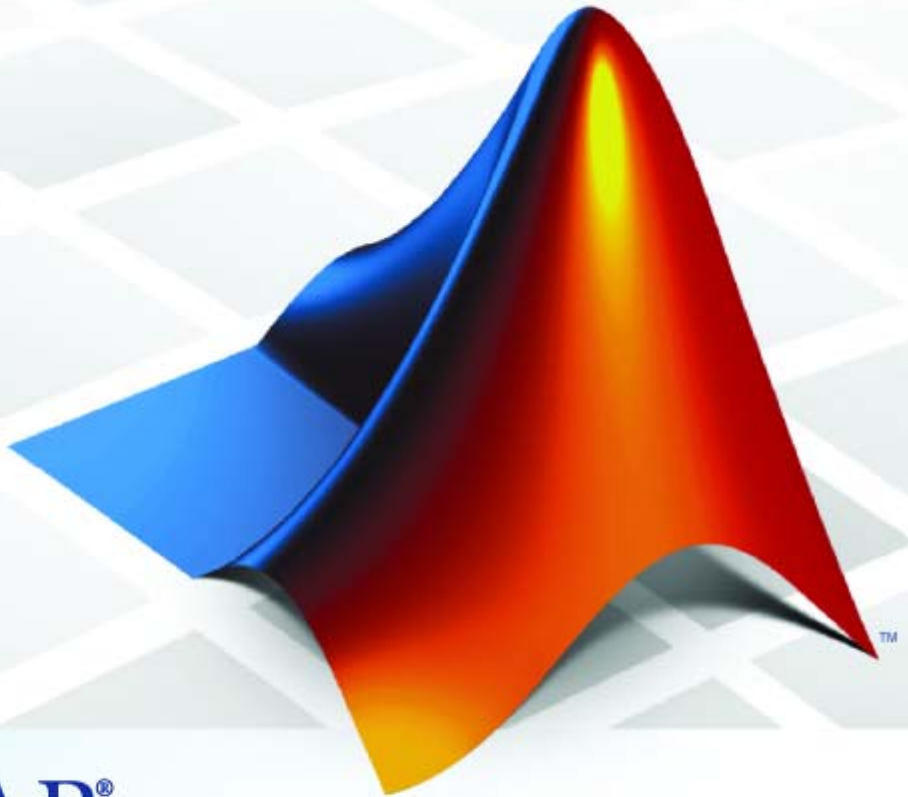


Simulink® Verification and Validation™ 2 User's Guide



MATLAB®
& SIMULINK®

How to Contact The MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Simulink® Verification and Validation™ User's Guide

© COPYRIGHT 2004–2009 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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)
March 2008	Online only	Revised for Version 2.3 (Release 2008a)
October 2008	Online only	Revised for Version 2.4 (Release 2008b)
March 2009	Online only	Revised for Version 2.5 (Release 2009a)
September 2009	Online only	Revised for Version 2.6 (Release 2009b)

Getting Started

1

Product Overview	1-2
System Requirements	1-3
Operating System Requirements	1-3
Product Requirements	1-3
Organization of This User's Guide	1-5

Managing Model Requirements

2

About the Requirements Management Interface	2-2
About Requirements Documents	2-3
Requirements Document Types	2-3
Locations Within Requirements Documents	2-3
Linking a Simulink Object to a Location in a Requirements Document	2-6
What Is Selection-Based Linking?	2-6
Creating a Requirements Document in Microsoft Word ...	2-6
Linking from a Simulink Object to a Selected Item in a Requirements Document	2-7
Customizing Selection-Based Linking	2-8
Linking from a Simulink Object to a Specified Location in a Requirements Document	2-9
Creating a Requirements Document in a Microsoft® Excel Spreadsheet	2-12
Adding Requirement Links to Multiple Objects Simultaneously	2-12

Linking a Signal Builder Block to a Requirement	2-13
Resolving the Document Path	2-15
Viewing Simulink Objects That Have Requirements	
Links	2-17
Highlighting Objects with Requirements in the Model Editor	2-17
Highlighting Objects with Linked Requirements from Model Explorer	2-18
Deleting Requirement Links from Simulink Objects ..	2-19
Deleting a Single Link from a Simulink Object	2-19
Deleting All Links from a Simulink Object	2-19
Deleting Links from Multiple Simulink Objects	2-19
Creating Requirements in Linked Libraries	2-21
Creating a Requirements Report	2-22
Creating the Default Requirements Report	2-22
Customizing a Requirements Report	2-26
Navigating from Requirements Documents to Simulink Objects	2-30
Configuring the RMI to Insert Navigation Controls	2-30
Enabling ActiveX Controls	2-30
Creating Navigation Controls in Requirements Documents	2-31
Troubleshooting Simulink Navigation Controls in Microsoft Office 2007	2-32
Linking to Custom Types of Requirements	
Documents	2-40
Built-In Link Types	2-40
Why Create a Custom Link Type?	2-40
Custom Link Type Registration	2-41
Link Properties	2-42
Link Type Properties	2-42
Creating a Custom Link Requirement Type	2-44
Navigating to Simulink Objects from External Documents	2-53

Using the System Requirements Block in a Model	2-56
About the System Requirements Block	2-56
Adding the System Requirements Block	2-56
Renaming the System Requirements Block	2-57
Including Requirements Information with Generated Code	2-59

Managing Requirements with DOORS Software

3

Why Use DOORS Requirements with Simulink Objects?	3-2
Configuring the Requirements Management Interface for DOORS Software	3-4
Before You Begin	3-4
Installing DOORS Software	3-4
Manually Installing Additional Files for DOORS Software	3-4
Upgrading DOORS Software	3-5
Linking Simulink Objects to DOORS Requirements . . .	3-7
Creating DOORS Requirements	3-7
Creating One-Way Links from Simulink Objects to DOORS Requirements	3-8
Navigating from a Simulink Object to a DOORS Requirement	3-10
Synchronizing a Simulink Model to a DOORS Surrogate Module	3-12
What Is a Surrogate Module?	3-12
What Is Synchronization?	3-12
Advantages of Synchronization	3-14
Synchronizing a Simulink Model to Create a Surrogate Module	3-14
Customizing the Synchronization	3-16
Updating the Surrogate Module to Reflect Model Changes	3-22

Navigating Using the Surrogate Module	3-25
Viewing Simulink Objects with Requirements	3-28
Viewing Objects with Requirements in the Model Editor ..	3-28
Viewing Objects with Requirements in the Model Explorer	3-28
Creating Requirements Reports	3-30
About Requirements Reports	3-30
Creating a Default Requirements Report for a Model	3-30
Customizing a Requirements Report with Links to DOORS Requirements	3-31
Creating Two-Way Links Between Requirements and Simulink Objects	3-35
Creating Two-Way Links	3-35
Navigating Two-Way Links	3-36

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

Introduction to Model Coverage	5-2
What Is Model Coverage?	5-2
How Model Coverage Works	5-2
Simulink Optimizations and Model Coverage	5-2
Types of Model Coverage	5-3
Blocks That Receive Model Coverage	5-8
Analyzing Model Coverage	5-11
Model Coverage Analysis Workflow	5-11
Creating and Running Test Cases	5-11
Model Coverage Reporting Options	5-16
Coverage Settings Dialog Box	5-16
Coverage Tab	5-18
Results Tab	5-21
Report Tab	5-23
Options Tab	5-27
Understanding Model Coverage Reports	5-30
Types of Coverage Reports	5-30
Model Coverage Reports	5-31
Model Summary Reports	5-61
Model Reference Coverage Reports	5-62
External M-File Coverage Reports	5-62
Subsystem Coverage Reports	5-66
Colored Simulink Diagram Coverage Display	5-69
How Model Coverage Highlighting Works	5-69
Enabling the Colored Diagram Display	5-69
Displaying Model Coverage with Model Coloring	5-70
Accessing Coverage Information for Colored Blocks	5-72
Using Model Coverage Commands	5-74
About Model Coverage Commands	5-74
Creating Tests with cvtest	5-74
Running Tests with cvsim	5-76
Producing HTML Reports with cvhtml	5-77
Saving Test Runs to a File with cvsave	5-78

Loading Stored Coverage Test Results with cvload	5-79
Coverage Script Example	5-79
Using Model Coverage Commands for Referenced	
Models	5-81
Introduction	5-81
Creating a Test Group with cv.cvtestgroup	5-84
Running Tests with cvsimref	5-84
Extracting Results from cv.cvdatagroup	5-85
Model Coverage for Embedded MATLAB Function	
Blocks	5-87
Types of Model Coverage in Embedded MATLAB Function	
Blocks	5-87
Creating a Model with Embedded MATLAB Function Block	
Decisions	5-88
Understanding Embedded MATLAB Function Block Model	
Coverage	5-92

Customizing the Model Advisor

Overview of the Model Advisor

6

Why Use and Customize the Model Advisor?	6-2
About the Model Advisor	6-2
Customizing the Model Advisor	6-2
Customizing and Using the Model Advisor Workflow ..	6-4
Before Customizing the Model Advisor	6-5

Authoring Checks Workflow	7-2
Customization File Overview	7-3
Register Checks and Process Callbacks	7-6
Create sl_customization Function	7-6
Registering Checks and Process Callbacks	7-6
Defining Startup and Post-Execution Actions Using Process Callback Functions	7-8
Defining Custom Checks	7-11
About Custom Checks	7-11
Contents of Check Definitions	7-11
Displaying and Enabling Checks	7-13
Defining Where Custom Checks Appear	7-14
Model Advisor Code Example: Check Definition Function	7-15
Defining Check Input Parameters	7-16
Defining Model Advisor Result Explorer Views	7-18
Defining Check Actions	7-19
Creating Callback Functions and Results	7-22
About Callback Functions	7-22
Common Utilities for Authoring Checks	7-23
Simple Check Callback Function	7-23
Detailed Check Callback Function	7-31
Check Callback Function with Hyperlinked Results	7-33
Action Callback Function	7-37
Formatting Model Advisor Results	7-38

Creating Custom Configurations by Organizing Checks and Folders

Overview of Creating Custom Configurations	8-2
---	-----

About Creating Custom Configurations	8-2
Creating Custom Configurations Workflow	8-2
Using the Model Advisor Configuration Editor Versus Customization File	8-3
Organizing Checks and Folders Using the Model	
Advisor Configuration Editor	8-4
Overview of the Model Advisor Configuration Editor	8-4
Starting the Model Advisor Configuration Editor	8-9
How To Organize Checks and Folders Using the Model Advisor Configuration Editor	8-10
Organizing Checks and Folders Within a Customization	
File	8-12
Customization File Overview	8-12
Register Tasks and Folders	8-13
Defining Custom Tasks	8-15
Defining Custom Folders	8-18
Demo and Code Example	8-20
Verifying and Using Custom Configurations	8-22
Updating the Environment to Include Your sl_customization File	8-22
Verifying Custom Configurations	8-22

Deploying Custom Configurations

9

Overview of Deploying Custom Configurations	9-2
About Deploying Custom Configurations	9-2
Deploying Custom Configurations Workflow	9-2
How to Deploy Custom Configurations	9-3
Loading and Setting the Default Configuration	9-4

Function Reference

10

Requirements Management Interface	10-2
Model Coverage	10-3
Model Advisor Customization API	10-5
Model Advisor Result Template API	10-7
Model Advisor Formatting API	10-8

Class Reference

11

Model Coverage	11-2
Model Advisor Customization API	11-3
Model Advisor Result Template API	11-4
Model Advisor Formatting API	11-5

Alphabetical List

12

Block Reference

13

Model Advisor Checks

14

Simulink® Verification and Validation Checks	14-2
Simulink® Verification and Validation Checks Overview ..	14-2
Modeling Standards Checks Overview	14-3
DO-178B Checks	14-4
DO-178B Checks Overview	14-5
Check safety-related optimization settings	14-6
Check safety-related diagnostic settings for solvers	14-10
Check safety-related diagnostic settings for sample time ..	14-13
Check safety-related diagnostic settings for signal data ..	14-16
Check safety-related diagnostic settings for parameters ..	14-19
Check safety-related diagnostic settings for data used for debugging	14-22
Check safety-related diagnostic settings for data store memory	14-24
Check safety-related diagnostic settings for type conversions	14-26
Check safety-related diagnostic settings for signal connectivity	14-28
Check safety-related diagnostic settings for bus connectivity	14-30
Check safety-related diagnostic settings that apply to function-call connectivity	14-32
Check safety-related diagnostic settings for compatibility	14-34
Check safety-related diagnostic settings for model initialization	14-36

Check safety-related diagnostic settings for model referencing	14-38
Check safety-related model referencing settings	14-41
Check safety-related code generation settings	14-43
Check safety-related diagnostic settings for saving	14-50
Check for model objects that do not link to requirements ..	14-52
Check for proper usage of Math blocks	14-53
Check for proper usage of For Iterator blocks	14-54
Check for proper usage of While Iterator blocks	14-55
Display model version information	14-57
Check for proper usage of blocks that compute absolute values	14-58
Check for proper usage of Relational Operator blocks	14-60
IEC 61508 Checks	14-62
IEC 61508 Checks Overview	14-62
Display model metrics and complexity report	14-64
Check for unconnected objects	14-65
Check for fully defined interface	14-66
Check for questionable constructs	14-68
Check usage of Stateflow constructs	14-70
Check for model objects that do not link to requirements ..	14-73
Display configuration management data	14-74
Check usage of Simulink constructs	14-75
MathWorks Automotive Advisory Board Checks	14-79
MathWorks Automotive Advisory Board Checks	
Overview	14-81
Check for difference in font and font sizes	14-82
Check transition orientations in flow charts	14-84
Check for display of nondefault block attributes	14-85
Check for proper labeling on signal lines	14-86
Check for propagated labels on signal lines	14-88
Check default transition placement in Stateflow charts ..	14-90
Check setting Stateflow graphical function return value ..	14-91
Check for blocks not using one-based indexing	14-92
Check for invalid file names	14-94
Check for invalid model directory names	14-96
Check for blocks that are not discrete	14-97
Check for prohibited sink blocks	14-98
Check for invalid port positioning and configuration	14-99
Check for mismatches between names of ports and corresponding signals	14-101

Check whether block names do not appear below blocks ..	14-102
Check for systems that mix primitive blocks and subsystems	14-103
Check whether model has unconnected block input ports, output ports, or signal lines	14-105
Check for improperly positioned Trigger and Enable blocks	14-106
Check whether annotations have drop shadows	14-107
Check whether tunable parameters specify expressions, data type conversions, or indexing operations	14-108
Check whether Stateflow events are defined at the chart level or below	14-110
Check whether Stateflow data objects with local scope are defined at the chart level or below	14-111
Check interface signals and parameters	14-112
Check for exclusive states, default states, and substate validity	14-113
Check optimization parameters for Boolean data types ...	14-115
Check model diagnostic settings	14-116
Check the display attributes of block names	14-119
Check icon display attributes for port blocks	14-120
Check whether subsystem block names include invalid characters	14-121
Check whether Inport and Outport block names include invalid characters	14-123
Check whether signal line names include invalid characters	14-125
Check whether block names include invalid characters ...	14-127
Check Trigger and Enable block port names	14-129
Check for Simulink diagrams that have nonstandard appearance attributes	14-130
Check visibility of port block names	14-133
Check for direction of subsystem blocks	14-135
Check for proper position of constants used in Relational Operator blocks	14-136
Check for entry format in state blocks	14-137
Check for use of tunable parameters in Stateflow	14-139
Check for proper use of Switch blocks	14-140
Check for proper use of signal buses and Mux block usage	14-141
Check for mismatches between Stateflow ports and associated signal names	14-143
Check for proper scope of From and Goto blocks	14-144

Requirements Consistency Checks	14-145
Identify requirement links with missing documents	14-146
Identify requirement links that specify invalid locations within documents	14-147
Identify selection-based links having descriptions that do not match their requirements document text	14-148
Identify requirement links with inconsistent path types and preferences	14-150

Examples

A

Requirements Management Interface	A-2
Requirements Management Interface (DOORS Version)	A-2
Verification Manager	A-3
Model Coverage	A-3
Model Advisor Check	A-4
Model Advisor Organization	A-4

Index

Getting Started

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

- “Product Overview” on page 1-2
- “System Requirements” on page 1-3
- “Organization of This User’s Guide” on page 1-5

Product Overview

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

- Associate design requirements that you manage using external applications with Simulink model objects that implement the requirements
- Verify proper function of the model by monitoring model signals during extensive testing
- Validate the model, making sure that all possible model decisions are taken through testing.
- Customize the Model Advisor to analyze a model for settings that result in inaccuracies or inefficiencies.

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

System Requirements

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

Operating System Requirements

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

- Microsoft® Windows® XP and Windows Vista™
- UNIX® systems (Requirements linking to HTML and TXT documents only)

Product Requirements

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

- MATLAB®
- Simulink

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

- Stateflow

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

- Microsoft Word 2000 or later
- Microsoft® Excel® 98 or later
- IBM® Rational® DOORS® 6.0 or later

- Adobe® PDF

Organization of This User's Guide

The component tools of the Simulink Verification and Validation software are organized on the basis of workflow that you follow in certifying the correct and complete behavior of your models. This workflow 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 Requirements Management Interface with requirements in HTML, PDF, TXT, Microsoft Word, and Microsoft Excel documents. Use this feature to associate requirements with objects in Simulink models and Stateflow charts.
 - Chapter 3, “Managing Requirements with DOORS Software” — Instructions for using the DOORS software with the Requirements Management Interface. Use this feature to associate Simulink models and Stateflow charts with requirements in the DOORS software.
- 2** Verify proper performance of the model by monitoring model signals during extensive testing with model verification blocks using the Verification Manager, which is described in the following chapter:
 - Chapter 4, “Managing Model Verification Blocks” — Shows you how to use verification blocks individually in Simulink models and how to manage them as a group for testing.
- 3** Validate the model by making sure that all possible model decisions are taken through testing, by using the Model Coverage tool, which is described in the following chapter:
 - Chapter 5, “Using Model Coverage” — Shows you how to generate and interpret model coverage reports and displays for validating model decisions.
- 4** Customize the Model Advisor to analyze your model for conditions and configuration settings that result in inaccurate or inefficient simulation or code generation. You can write custom checks, tasks, and callback functions, as described in the following chapter:

- Customizing the Model Advisor on page 1 — Shows you how to define custom checks and tasks, write callback functions, and register customizations for the Model Advisor.

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

- Chapter 10, "Function Reference" — Provides a categorical list of functions used in executing and managing model coverage tests and reports from the MATLAB prompt. Automate your model coverage tests with scripts of MATLAB commands calling these functions.
- Chapter 12, "Alphabetical List" — Provides an alphabetical reference of functions used in executing and managing model coverage tests and reports from the MATLAB prompt.
- Chapter 13, "Block Reference" — Provides reference information for the Simulink Verification and Validation library, which currently contains only one block, System Requirements. This block lets you list all the requirements for a model or subsystem on its Simulink diagram.

Managing Model Requirements

- “About the Requirements Management Interface” on page 2-2
- “About Requirements Documents” on page 2-3
- “Linking a Simulink Object to a Location in a Requirements Document” on page 2-6
- “Viewing Simulink Objects That Have Requirements Links” on page 2-17
- “Deleting Requirement Links from Simulink Objects” on page 2-19
- “Creating Requirements in Linked Libraries” on page 2-21
- “Creating a Requirements Report” on page 2-22
- “Navigating from Requirements Documents to Simulink Objects” on page 2-30
- “Linking to Custom Types of Requirements Documents” on page 2-40
- “Using the System Requirements Block in a Model” on page 2-56
- “Including Requirements Information with Generated Code” on page 2-59

About the Requirements Management Interface

The Requirements Management Interface (RMI) allows you to link requirements with Simulink and Stateflow objects. Requirements links have the following attributes:

- A description of up to 255 characters.
- A requirements document path name, such as a Microsoft Word file. (The RMI supports several built-in document formats and also allows you to register custom types of requirements documents.)
- A link to a location inside the requirements document, such as bookmark, anchor, line number, cell range, and so on.

Use the RMI to:

- Associate requirements with:
 - Simulink objects, such as subsystems, blocks, and signals
 - Stateflow charts, states, transitions, boxes, and functions
- Navigate from a Simulink or Stateflow object to requirements information.
- Navigate from an embedded link in a requirements document to the corresponding Simulink or Stateflow object.
- Highlight Simulink or Stateflow objects that have requirements associated with them.
- Create reports for your Simulink model that show which objects have links to which requirements.

Note You can add requirements to a Model or Subsystem block linked from a Simulink library, but not to its contents. You cannot modify blocks in a Simulink library.

About Requirements Documents

In this section...
“Requirements Document Types” on page 2-3
“Locations Within Requirements Documents” on page 2-3

Requirements Document Types

The RMI supports the following types of requirements documents:

- Text
- HTML
- PDF
- Microsoft Word 2007 and earlier
- Microsoft Excel 2007 and earlier
- An IBM Rational DOORS database — see “Linking a Simulink Object to a Location in a Requirements Document” on page 2-6

You can also link to requirements in custom types of documents — see “Linking to Custom Types of Requirements Documents” on page 2-40

Locations Within Requirements Documents

Depending on the requirements document type, you can link to specific locations within a document using the following options in the Requirements dialog box.

Requirements Document Type	Location Options
Text	<ul style="list-style-type: none"> • Search text — Type a string in the Location text field. The RMI links to the first occurrence of the text string within the document. This search is not case sensitive. • Line number — Type a line number in the Location text field. The RMI links to the specified line.
HTML	<p>You can link only to a named anchor.</p> <p>For example, in your HTML requirements document, if you define the anchor</p> <pre data-bbox="654 725 1258 753" style="text-align: center;"> ...contents... </pre> <p>in the Location text field, enter <code>valve_timing</code> or choose the anchor name from the document index.</p>
Microsoft Word	<ul style="list-style-type: none"> • Search text — Type a string in the Location text field. The RMI links to the first occurrence of the text string within the document. This search is not case sensitive. • Named item — Type the bookmark name in the Location text field, or select the name from the document index. The RMI links to the location of that bookmark in the document. • Page/item number — Type a page number in the Location text field. The RMI links to the top of that page.

Requirements Document Type	Location Options
Microsoft Excel	<ul style="list-style-type: none"> • Search text — Type a string in the Location text field. The RMI links to the first occurrence of the text string within the workbook. This search is not case sensitive. • Named item — Type the name in the Location text field. The RMI links to that named item within the workbook. • Sheet range — Type a location in a workbook in the Location text field: <ul style="list-style-type: none"> ▪ Cell number (A1, C13) ▪ Range of cells (C5:D7) ▪ Range of cells on another worksheet (Sheet1!A1:B4) <p>The RMI links to the specified cell or cells.</p>
PDF	<ul style="list-style-type: none"> • Named item — Type the bookmark name in the Location field, or select the bookmark name in the document index. The RMI links to the location of that bookmark in the document. • Page/item number — Type a page number in the Location text field. The RMI links to the top of the page.
Web browser URL	<p>The RMI can link to a URL location. Type the URL string in the Document text field. When you click the link, the document opens in a Web browser. Similarly to HTML document types, you can explicitly specify the anchor in the Location text field.</p>

Linking a Simulink Object to a Location in a Requirements Document

In this section...

“What Is Selection-Based Linking?” on page 2-6

“Creating a Requirements Document in Microsoft Word” on page 2-6

“Linking from a Simulink Object to a Selected Item in a Requirements Document” on page 2-7

“Customizing Selection-Based Linking” on page 2-8

“Linking from a Simulink Object to a Specified Location in a Requirements Document” on page 2-9

“Creating a Requirements Document in a Microsoft® Excel Spreadsheet” on page 2-12

“Adding Requirement Links to Multiple Objects Simultaneously” on page 2-12

“Linking a Signal Builder Block to a Requirement” on page 2-13

“Resolving the Document Path” on page 2-15

What Is Selection-Based Linking?

Selection-based linking is linking a Simulink object to a selected item in a requirements document. Using selection-based linking, the RMI creates a link from the object to the requirement text.

The following tutorial takes you through the steps to link a block in a Simulink model to text in a Microsoft Word document.

Creating a Requirements Document in Microsoft Word

For this tutorial, create and save a Microsoft Word 2007 document, `requirements.docx`, with the content shown in the following graphic. Style the header lines “Transmission Requirements” and “Engine Requirements” as Heading 1.

Engine Requirements:

Engine torque should increase with throttle opening. Impeller resistance torque counteracts generated engine torque.

Transmission Requirements:

Should have five gears: Park, Reverse, D1, D2, and Lo. User cannot start engine unless the Park gear is engaged. Reverse cannot be entered from D1 unless the vehicle is not moving. Lo cannot be entered from D2 unless the speed of the vehicle is less than 15 mph.

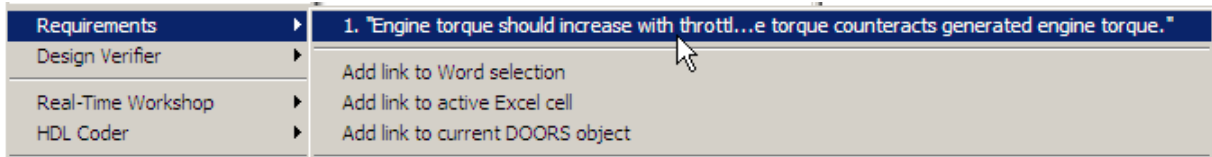
Note The requirements document for this exercise is in Microsoft Word 2007 format. If you do not have Microsoft Word 2007, you can use earlier versions of the Microsoft Word software.

Linking from a Simulink Object to a Selected Item in a Requirements Document

Open a Simulink demo model. Create a link from a block in the model to the text you selected in the requirements document:

- 1 At the MATLAB prompt, type `sf_car` to open the model.
- 2 In `requirements.docx`, select the paragraph under “Engine Requirements”.
- 3 Right-click the Engine block and select **Add link to Word selection**.
- 4 Verify the link from the Engine block by right-clicking the Engine block and selecting **Requirements**.

A shortened version of the requirements text appears at the top of the context menu.



- 5 Click the requirement text to navigate to that location in the requirements document.
- 6 Save the model as `sf_car_linking`.
- 7 The RMI inserts a bookmark into the requirements document when creating the link. Save the requirements document.

Customizing Selection-Based Linking

You can customize selection-based linking. From the Model Editor window, select **Tools > Requirements > Settings**. On the **Selection-based linking** tab, the following options are available.

Selection-Based Linking Option	Description
Document types	Specify the document types for which you want selection-based linking to be available.
Document file reference	<p>Specify how to store the requirements document. Valid options are:</p> <ul style="list-style-type: none"> • absolute path • path relative to current directory • path relative to model directory • filename only (on MATLAB path) <p>For more information, see “Resolving the Document Path” on page 2-15.</p>

Selection-Based Linking Option	Description
Modify documents to include links to models	Select this option to enable the RMI to insert navigation objects into the requirements document. By default, the RMI inserts only bookmarks (when needed) into the requirements document.
Model file reference	Specify how to locate the model path from a requirements document. Valid options are: <ul style="list-style-type: none"> • absolute • none (on MATLAB path) For more information, see “Resolving the Document Path” on page 2-15.

Linking from a Simulink Object to a Specified Location in a Requirements Document

In the following step, you create a link from another Simulink object to another requirement in the requirements document that you created. Use document indexing instead of selection-based linking.

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

The Requirements dialog box for the Engine block opens.

- 2 In the Requirements: transmission dialog box, click **New** to add a new requirement.
- 3 In the **Description** field, enter `Transmission requirements`.
- 4 In the **Document type** field, select `Microsoft Word` to search only for those file types.
- 5 Next to the **Document type** field, click **Browse**.

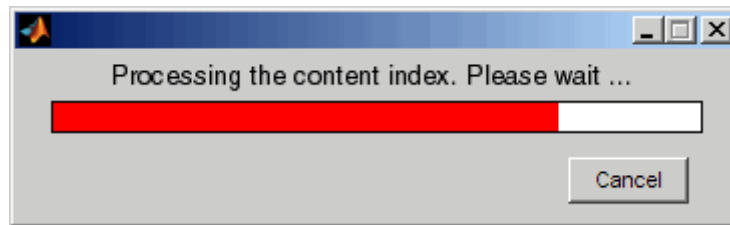
- 6 Locate the requirements document, `requirements.docx`, and select **Open**.

The file path appears in the **Document** field.

Note For information about how the RMI resolves the path to the requirements document, see “Resolving the Document Path” on page 2-15.

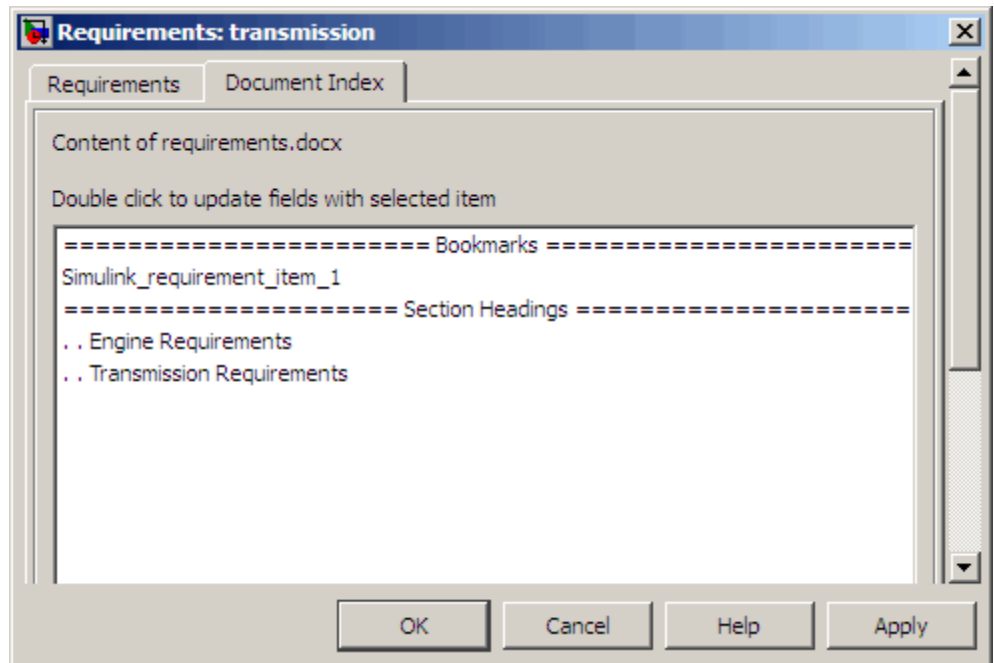
- 7 Click the **Document Index** tab to create an index of the requirements document so that you can easily access specific locations in the document.

Note After you click the **Document Index** tab, the document indexing begins immediately. You see the following status bar.



If you do not want the document index or if it takes too long to process the file, click **Cancel**. If you click **Cancel**, processing stops and the software creates a partial document index. You cannot resume the document indexing unless you restart your MATLAB session.

- 8 From the automatically generated list of headings and bookmarks in the document, select **Transmission Requirements**.



Note The document index includes the bookmark `Simulink_requirement_item_1` created during the selection-based linking tutorial in “Linking from a Simulink Object to a Selected Item in a Requirements Document” on page 2-7.

- 9 Click **Apply** to create the link from the requirements document text to the Engine block and redisplay the **Requirements** tab.
- 10 (Optional) To provide additional details about the current requirement, on the **Requirements** tab, in the **User tag** field, enter additional, descriptive text.
- 11 To verify the link from the transmission block to the requirement, right-click the transmission block and select **Requirements**.

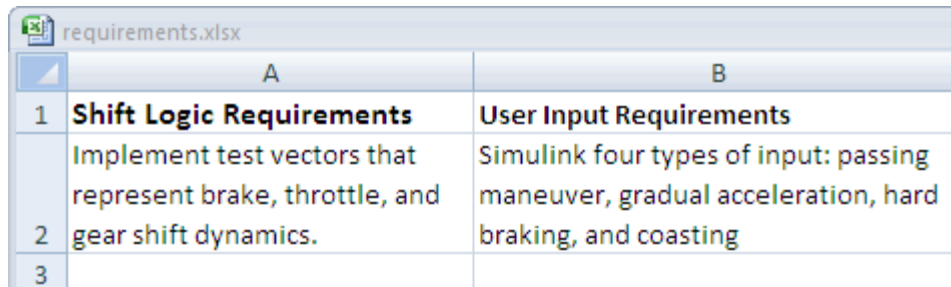
The name of the requirement in the new link is a submenu selection.



12 Save the sf_car_linking model.

Creating a Requirements Document in a Microsoft Excel Spreadsheet

For the next two tutorials, create a requirements document in a Microsoft Excel spreadsheet with the following content and save it as requirements.xlsx.



	A	B
1	Shift Logic Requirements	User Input Requirements
2	Implement test vectors that represent brake, throttle, and gear shift dynamics.	Simulink four types of input: passing maneuver, gradual acceleration, hard braking, and coasting
3		

Note The requirements document for this exercise is in Microsoft Excel 2007 format. If you do not have Microsoft Excel 2007, you can use earlier versions of the Microsoft Excel software.

Adding Requirement Links to Multiple Objects Simultaneously

You can add or delete links to requirements for a selection of multiple Simulink or Stateflow objects. In this example, you modify the requirements document to add a third requirement, and link two objects in the model to the new requirement:

- 1 Open the Microsoft Excel requirements document (requirements.xlsx).
- 2 In the sf_car_linking model, select two objects together:
 - transmission

- shift_logic

3 Right-click any of the blocks that you selected and then select **Requirements > Add Links to All**.

The Add requirements dialog box opens for those two blocks.

4 Add a new requirement called Shift logic requirements.

5 Browse to the Microsoft Excel requirements document and click **Open**.

Note For information about how the RMI resolves the path to the requirements document, see “Resolving the Document Path” on page 2-15.

6 In the **Location** drop-down list, select Sheet range.

7 Next to the **Location** drop-down list, enter A1:A2 to specify the range of cells for the Shift Logic Requirements.

8 Click **OK** to close the Add requirements dialog box.

9 In the sf_car_linking model, verify that both the shift_logic Stateflow chart and the transmission block have a new requirement, Shift logic requirements.


10 Save the sf_car_linking model.

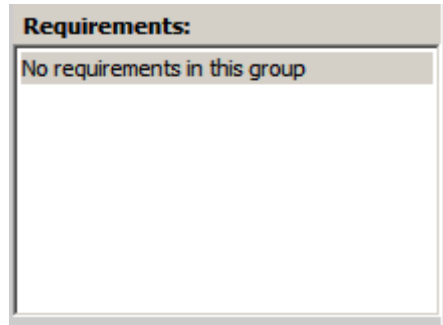
Linking a Signal Builder Block to a Requirement

The RMI can create a link from a Signal Builder block to a requirements document:

1 In the sf_car_linking model, double-click the User Inputs block.

The Signal Builder dialog box opens, displaying four groups of signals.

2 At the far-right end of the toolbar, click the **Show verification settings** button . You see a **Requirements** pane in the Signal Builder dialog box.



- 3** Place your cursor in the window, right-click, and select **Edit/Add Links**.
The Requirements dialog box opens.
- 4** Click **New**. In the **Description** field, enter User input requirements.
- 5** Browse to the Microsoft Excel requirements document and click **Open**.
- 6** In the **Location** drop-down list, select **Search text** to link to specified text in the spreadsheet.
- 7** Next to the **Location** drop-down list, enter User Input Requirements.
- 8** Click **Apply** to create the link.
- 9** To verify that the RMI creates the link, in the Model Editor, select the User Inputs block, right-click, and select **Requirements**.
- 10** Save the sf_car_linking model.

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

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

Resolving the Document Path

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

- 1** The software attempts to resolve the path relative to the current MATLAB folder.
- 2** If there is no path specification and the document is not in the current folder, the software uses the MATLAB search path to locate the file.
- 3** If the RMI cannot locate the document relative to the current folder or the MATLAB search path, the RMI resolves the path relative to the model file folder.

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

Relative (Partial) Path Example

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

Relative (No) Path Example

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

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

Absolute Path Example

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

Viewing Simulink Objects That Have Requirements Links

In this section...

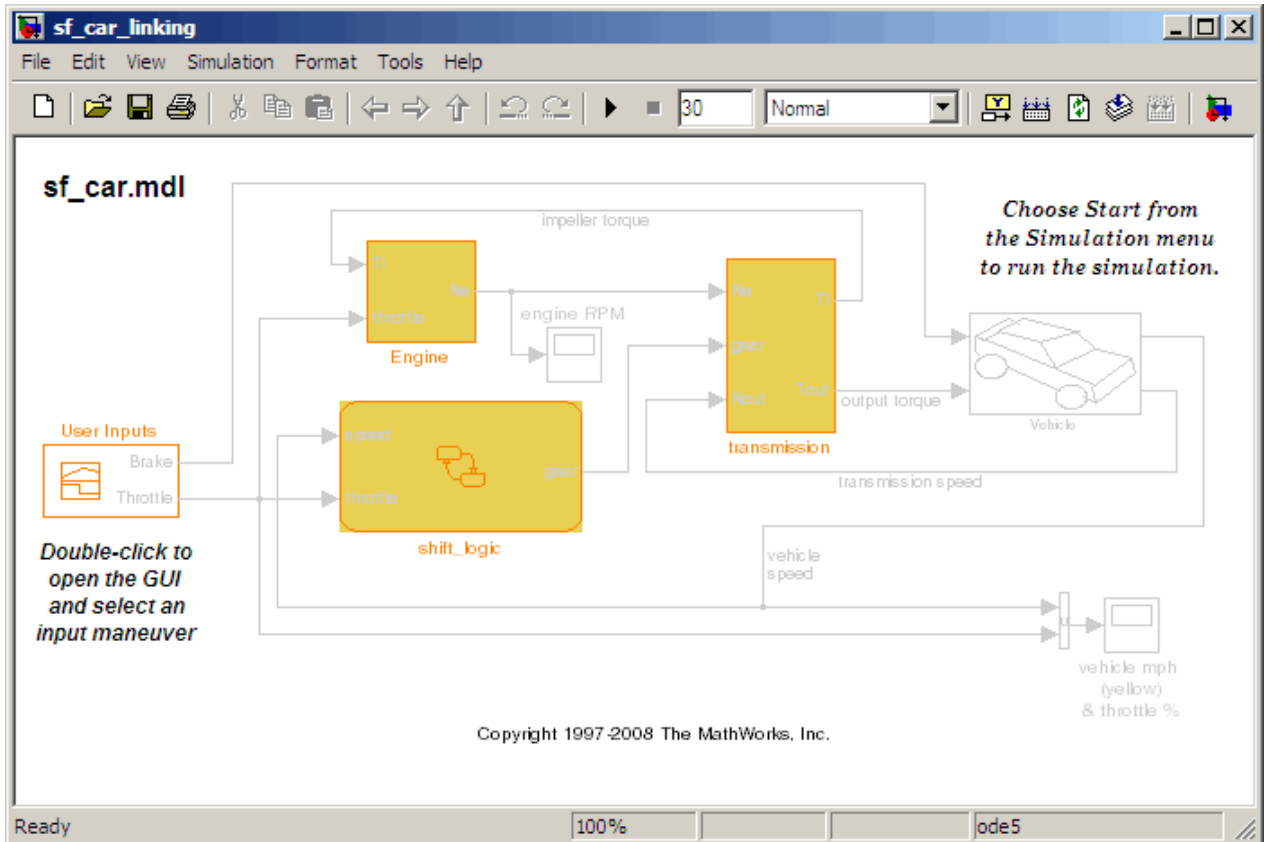
“Highlighting Objects with Requirements in the Model Editor” on page 2-17
“Highlighting Objects with Linked Requirements from Model Explorer”
on page 2-18

Highlighting Objects with Requirements in the Model Editor

To see the Simulink objects in the `sf_doors_linking` model that you linked requirements to:



- 1** Open the `sf_doors_linking` model.
- 2** To force model-wide library block updates so that all library linked blocks are up to date, select **Edit > Update Diagram**
- 3** Select **Tools > Requirements > Highlight Model**.

The Engine and transmission blocks appear highlighted.



Highlighting Objects with Linked Requirements from Model Explorer

The Model Explorer offers two ways to identify Simulink objects with linked requirements:

- Click **Display objects with requirements** () to list only those objects in the model that have links to requirements.
- Click **Highlight Items with Requirements on Model** () to open the model in the Model Editor, with the objects having linked requirements highlighted.

Deleting Requirement Links from Simulink Objects

In this section...

“Deleting a Single Link from a Simulink Object” on page 2-19

“Deleting All Links from a Simulink Object” on page 2-19

“Deleting Links from Multiple Simulink Objects” on page 2-19

Deleting a Single Link from a Simulink Object

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

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

The Requirements dialog box opens.

- 2 In the top-most window, select the link that you want to delete.
- 3 Click the **Delete** button.
- 4 Click **Apply** to confirm the deletion.

Deleting All Links from a Simulink Object

To delete all links to requirements from a Simulink object:

- 1 Right-click the object and select **Requirements > Delete All Links**.
- 2 Click the **Delete** button.
- 3 Click **Apply** to confirm the deletion.

Deleting Links from Multiple Simulink Objects

To delete all links to requirements for a group of Simulink objects:

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

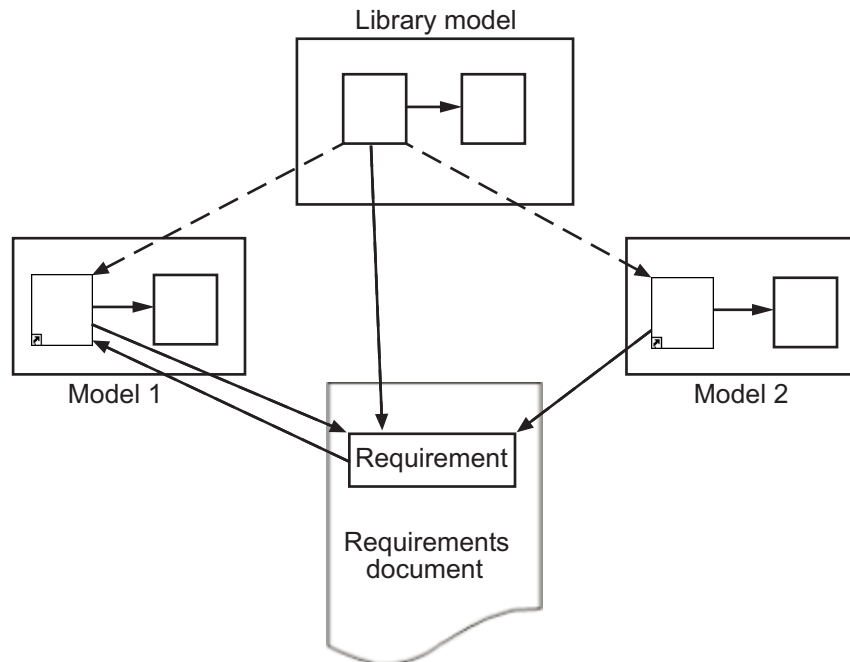
3 Click **Apply** to confirm the deletion.

Creating Requirements in Linked Libraries

You can create a requirement link for the contents of a library model. If you copy a library linked object to a Simulink model and disable the link, the software copies the contents of the linked object, including requirement links, to the model. Any navigation control in the requirements document still links to the object in the library.

You can also create a requirement link for the copy of a library linked object in a model (Model 1 in the following graphic). If you then push the change to the library, the software copies the requirement link to the library. However, any navigation control in the requirements document still links to the object in Model 1.

If you then link to that library object from another model (Model 2) and disable the link, the software copies the link to the requirement. Any navigation control in the requirements document still links to Model 1, creating the situation illustrated in the following graphic.



Creating a Requirements Report

In this section...
“Creating the Default Requirements Report” on page 2-22
“Customizing a Requirements Report” on page 2-26

Creating the Default Requirements Report

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

To generate a requirements report for the `sf_car_linking` model that you saved in “Linking a Signal Builder Block to a Requirement” on page 2-13:

- 1 Open the `sf_car_linking` model.
- 2 Select **Tools > Requirements > Generate Report**.

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

Table of Contents / List of Tables

The beginning of the report includes:

- A Table of Contents that lists the major sections of the report. There is one System section for the top-level model and one System section for each subsystem or referenced model. The final section lists the systems and subsystems that do not have any requirements.
- A List of Tables in the report, numbered by section.

Model Requirements

slemaire

17-Jun-2009 15:05:27

Table of Contents

- [1. Model Information](#)
- [2. Documents Information](#)
- [3. System - sf car linking](#)
- [4. System - Engine](#)
- [5. System - transmission](#)
- [6. Systems and subsystems with no requirements](#)

List of Tables

- 1.1. [sf car linking Version Information](#)
- 2.1. [Requirements documents linked in model](#)
- 3.1. [Blocks in sf car linking that have requirements](#)
- 3.2. [User Inputs : Passing Maneuver signal requirements](#)
- 3.3. [Objects in sf car linking that are not linked to requirements](#)
- 4.1. [sf car linking/Engine Requirements](#)
- 4.2. [Objects in sf car linking/Engine that are not linked to requirements](#)
- 5.1. [sf car linking/transmission Requirements](#)
- 5.2. [Objects in sf car linking/transmission that are not linked to requirements](#)

Model Information

The Model Information contains version information about the model for which you created the report.

Chapter 1. Model Information

Table 1.1. sf_car_linking Version Information

<i>ModelVersion</i>	1.85	<i>ConfigurationManager</i>	none
<i>Created</i>	Tue Jun 02 16:12:01 1998	<i>Creator</i>	The MathWorks Inc.
<i>LastModifiedDate</i>	Wed Jun 17 14:42:47 2009	<i>LastModifiedBy</i>	slemaire

Documents Information

The Documents Information section lists all the requirements documents to which objects in the model link to.

Chapter 2. Documents Information

Table 2.1. Requirements documents linked in model

ID	Document	Path type	Last modified	# refs.
DOC1	requirements.docx	relative	17-Jun-2009 13:24:01	1
DOC2	requirements.xlsx	absolute	17-Jun-2009 14:08:07	3
DOC3	requirements.docx	absolute	17-Jun-2009 13:24:01	1

System

The first System section lists information about the top-level model, `sf_car_linking`. It also lists information about its child objects that have requirements—the `shift_logic` chart and the Passing Maneuver signal group. The subsystems in `sf_car_linking` that have requirements—Engine and transmission—each have their own System section.

Table 3.1. Blocks in `sf_car_linking` that have requirements

Name	Requirements
<code>shift_logic</code>	Shift logic requirements DOC2 SA1:A2

Table 3.2. User Inputs : Passing Maneuver signal requirements

Description	Document	ID
User input requirements	DOC2	User Input Requirements

The first System section also lists objects in the top-level model that do not have requirements.

Table 3.3. Objects in `sf_car_linking` that are not linked to requirements

Name	Type
<code>engine RPM</code>	Scope
<code>Mux</code>	Mux
<code>Vehicle</code>	SubSystem
<code>vehicle mph (yellow) & throttle %</code>	Scope

Systems and subsystems with no requirements

The Systems and subsystems with no requirements section lists objects in the model that do not have requirements links. However, it does not list child

objects that have requirements links. In this example, that list includes the top-level model (sf_car_linking), which does not have links to requirements.

Chapter 6. Systems and subsystems with no requirements

These systems and subsystems are not linked to requirements. Their children may have links - refer to corresponding chapters above.

- ◆ sf_car_linking
- ◆ sf_car_linking/shift_logic/selection_state.calc_th
- ◆ sf_car_linking/transmission/transmission_ratio

Customizing a Requirements Report

RMI uses the Simulink® Report Generator™ software to generate the requirements report. You can customize the report using the RMI, or you can use the Simulink Report Generator software for advanced customization.

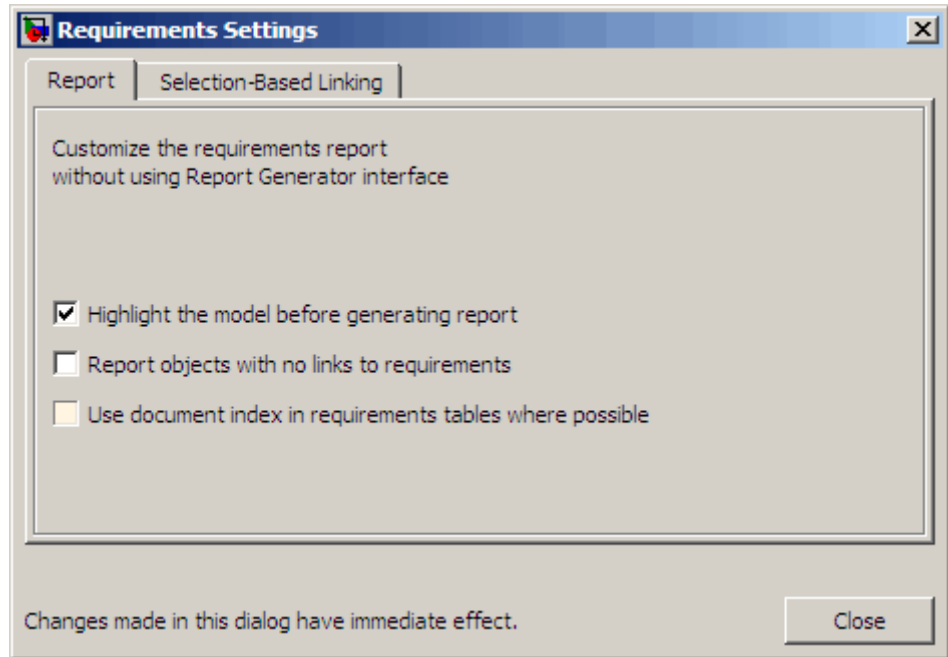
Customizing the Report Using the RMI

To customize the requirements report in the Model Editor:

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

The Requirements Settings dialog box opens.

- 2** Click the **Report** tab.



The options that you select in the Requirements Settings dialog box determine the contents of the report.

Requirements Settings Report Option	Description
Highlight the model before generating report	Highlights the Simulink objects with requirements in the Model Editor before creating the report and then highlights them in the report.
Report objects with no links to requirements	Lists Simulink objects that have no requirements.
Use document index in requirements tables where possible	Uses a document ID, if available, instead of a path name in the requirements table.

- 3 Select the options that you want and click **Close**.

Run the report to generate the requirements report for your model.

Customizing the Report Using the Simulink Report Generator Software

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

To customize the requirements report, first start the Simulink Report Generator software. At the MATLAB command prompt, enter the following command:

```
setedit requirements
```

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

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

In addition to the standard report components, Simulink Report Generator provides RMI-specific components. These components insert information about Simulink objects whether or not they have associated requirements:

- **Missing Requirements Block Loop** — Applies all child components to blocks that have no requirements
- **Missing Requirements System Loop** — Applies all child components to systems that have no requirements
- **Requirements Block Loop** — Applies all child components to blocks that have requirements
- **Requirements Documents Table** — Inserts a table that lists requirements documents
- **Requirements Signal Loop** — Applies all child components to signal groups with requirements

- **Requirements Summary Table** — Inserts a property table listing blocks that have requirements and requirements details
- **Requirements System Loop** — Applies all child components to systems with requirements
- **Requirements Table** — Inserts a table that lists system and subsystem requirements

There are several ways you can customize the requirements report:

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

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

Navigating from Requirements Documents to Simulink Objects

In this section...
“Configuring the RMI to Insert Navigation Controls” on page 2-30
“Enabling ActiveX Controls” on page 2-30
“Creating Navigation Controls in Requirements Documents” on page 2-31
“Troubleshooting Simulink Navigation Controls in Microsoft Office 2007” on page 2-32

Configuring the RMI to Insert Navigation Controls

To create links from the requirements document to Simulink objects, the RMI inserts Microsoft® ActiveX® controls into the requirements document. To initiate that, first run the RMI setup script. At the MATLAB Command Window, enter the following command:

```
rmi setup
```

This command runs a setup script that registers ActiveX® controls that the RMI inserts into requirements documents. If you enable this feature, as described in “Enabling ActiveX Controls” on page 2-30, the RMI can insert these controls into requirements documents when you create selection-based links. These controls allow you to navigate from a requirements document to the linked Simulink object.

Note If you have installed IBM Rational DOORS software on the machine, this command also invokes the corresponding setup script for the IBM Rational DOORS software. For more information, see “Configuring the Requirements Management Interface for DOORS Software” on page 3-4.

Enabling ActiveX Controls

When you use selection-based linking to create a link from a Simulink object to a requirements document, the RMI does not automatically insert a navigation object in the requirements document. (It does insert a bookmark,

when necessary, to enable the RMI to link to the correct location in the requirements document.)

To enable the RMI to insert a navigation object when creating a selection-based link:

- 1** In the Model Editor, select **Tools > Requirements > Settings**.
- 2** Select the **Selection-based linking** tab.
- 3** Select **Modify documents to include links to models** to allow the RMI to insert the navigation controls.
- 4** Click **Close** to close the Requirements Settings dialog box.


Note For more information about the selection-based linking settings, see “Customizing Selection-Based Linking” on page 2-8.

Creating Navigation Controls in Requirements Documents

If you enable the **Modify documents to include links to models** option, you can create navigation controls that link from the requirements document back to the Simulink object. Follow these steps, used for selection-based linking with the `sf_car_linking` model:

- 1** Open the requirements document `requirements.docx`.
- 2** Select the text under “Engine Requirements.”
- 3** Open the `sf_car_linking` model.
- 4** Right-click the engine RPM block and select **Requirements > Add link to Word selection**.
- 5** The RMI inserts an ActiveX control into the requirements document.

Engine Requirements

Engine torque should increase with throttle opening. Impeller resistance torque counteracts generated engine torque. 

Troubleshooting Simulink Navigation Controls in Microsoft Office 2007

- “Saving Requirements Documents to Microsoft Word 2007 Format” on page 2-32
- “Field Codes in Requirements Document” on page 2-33
- “ActiveX Control Does Not Link to Model Element” on page 2-35
- “Deleting an ActiveX Control from Microsoft® Excel 2007 file” on page 2-37

Saving Requirements Documents to Microsoft Word 2007 Format

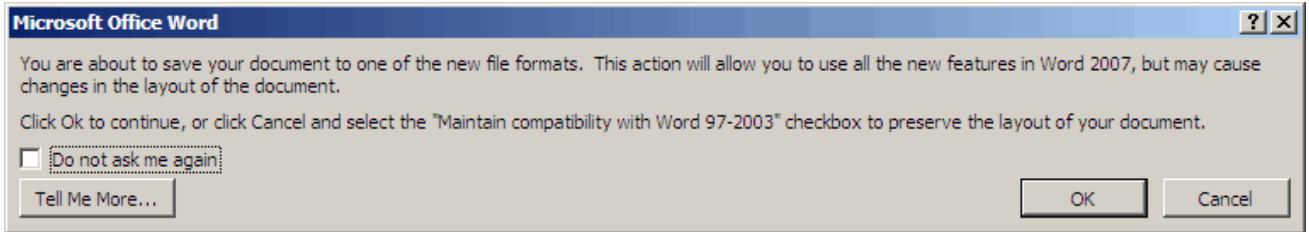
If you create a requirements document with an earlier version of Microsoft Word than Word 2007, the two-way links automatically work. If you save the document in Microsoft Word 2007 format, make sure that the two-way links continue to work:

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



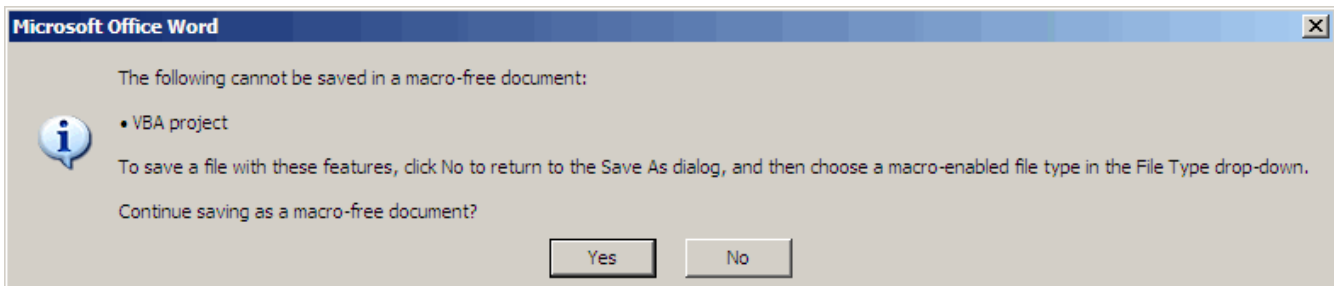
- 2** Select **Save As > Word Document**.

You see the following dialog box.



3 Click **OK**.

You then see the following dialog box.



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

Field Codes in Requirements Document

If your Microsoft Word requirements document displays the field codes in addition to, or instead of, the ActiveX icon, you must change a setting in Microsoft Word 2007.

The following graphic shows a requirements document created in Microsoft Word 2003, with the field codes displayed.

Determination of pumping efficiency{CONTROL

mwSimulink1.SLRefButton \s }



Requirement ID: REQ2

Model Element: fuelsys/fuel rate controller/Airflow calculat

Details: The airflow calculation will use a calibratib
pumping efficiency of the engine based on
manifold pressure.

The following graphic shows a requirements document created in Microsoft Word 2007, with the field codes displayed.

Primary Requirements

Requirement text. Requirement text. Requirement text.

Requirement text. Requirement text. Requirement text.

Requirement text. Requirement text. Requirement text. { CONTROL mwSimulink1.SLRefButton }

Requirement text. Requirement text. Requirement text.

Requirement text. Requirement text. Requirement text.

Requirement text. Requirement text. Requirement text.

To hide the field codes and display the ActiveX icon:

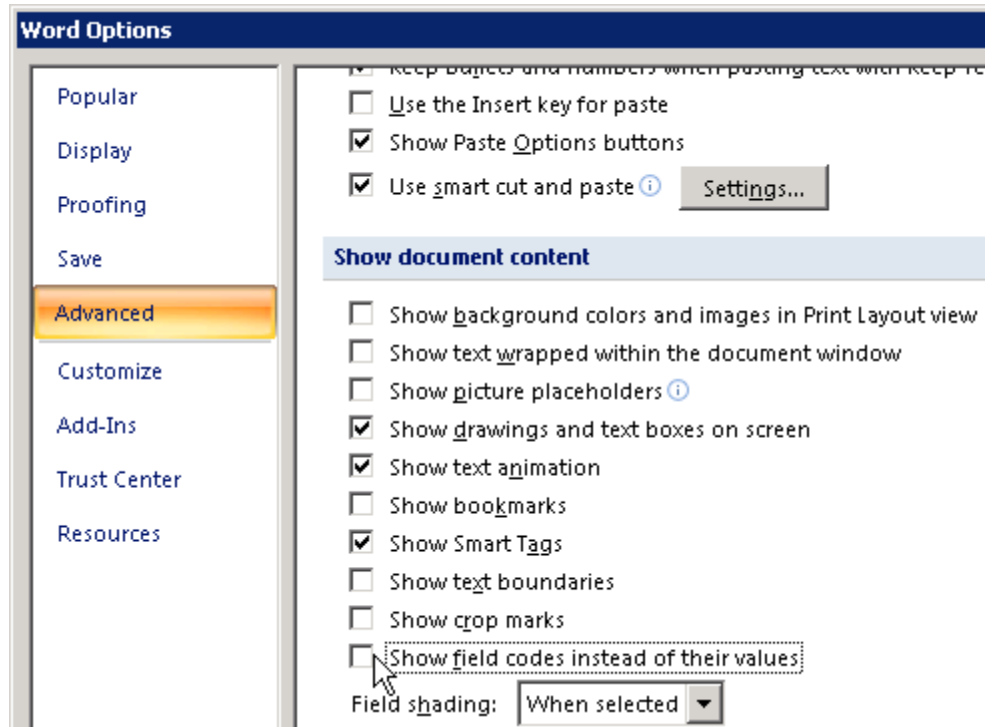
- 1** In the Microsoft Word window, in the upper-left corner, click the **Microsoft Office Button**.



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



- 3** In the left-hand portion of the pane, click **Advanced**.
- 4** In the **Advanced** pane, scroll to the **Show document content** section and clear the **Show field codes instead of their values** option.



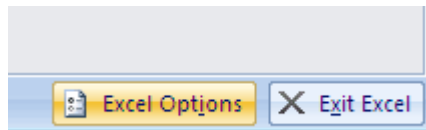
ActiveX Control Does Not Link to Model Element

If you click an ActiveX control that links to a Simulink or Stateflow object, and the object does not open, you have two options:

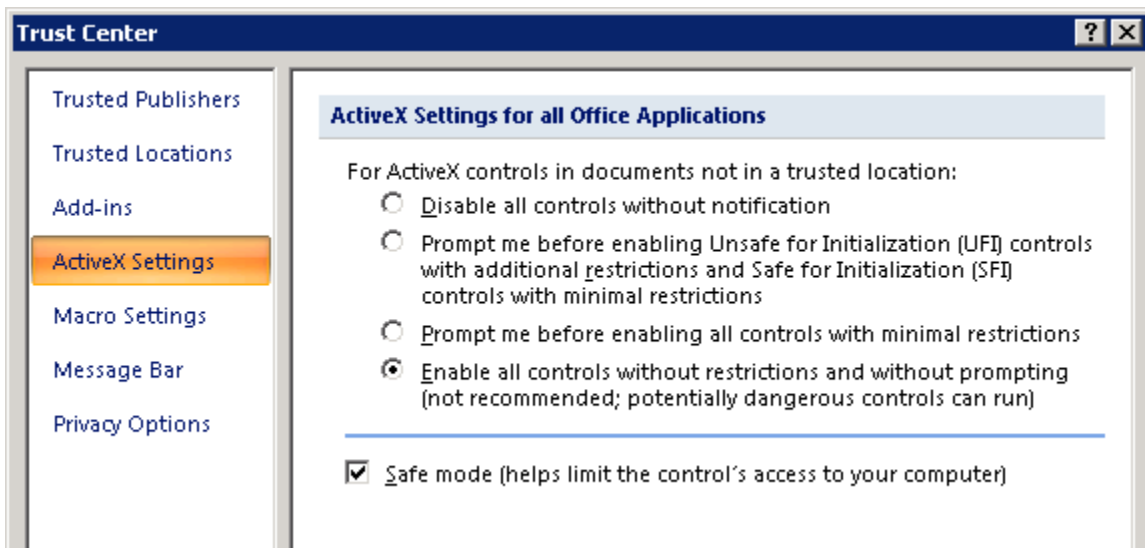
- Store your requirements documents in trusted locations, as described in the Microsoft Office 2007 documentation. The Trust Center does not check files for ActiveX controls stored in trusted locations, so you can maintain your Trust Center restrictions.
- Enable ActiveX controls:
 - 1 In the Microsoft Word or Microsoft Excel window, in the upper-left corner, click the **Microsoft Office Button**.



- 2 In the pane that opens, at the bottom, click **Word Options** or **Excel Options**, depending on which program you are running.



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



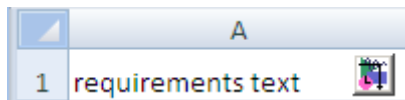
- 6 Select the setting that you want for ActiveX controls:

- **Prompt me for enabling all controls with minimum restrictions** to decide each time you click an ActiveX control if you want to enable all controls.
 - **Enable all controls without restrictions and without prompting** to enable all ActiveX controls.
- 7** Close the application and restart your computer so that the settings go into effect.

Deleting an ActiveX Control from Microsoft Excel 2007 file

To remove an ActiveX control from your Microsoft Excel 2007 file:

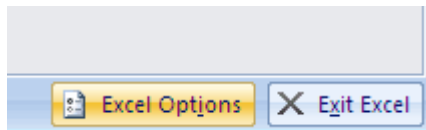
- 1** Your document has an ActiveX control in a worksheet cell.



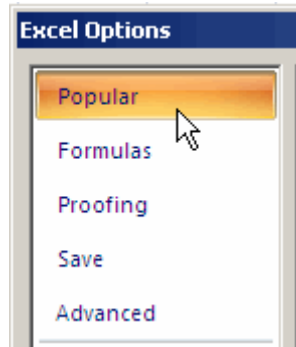
In the Microsoft Excel window, in the upper-left corner, click the **Microsoft Office Button**.



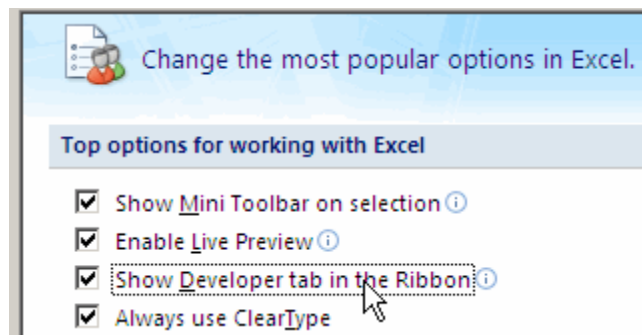
- 2** In the pane that opens, at the bottom, click **Excel Options**.



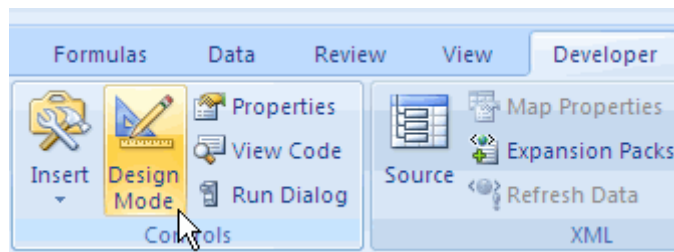
- 3** In the Excel Options dialog box, in the left-hand pane, click **Popular**.



- 4** On the **Popular** pane, in the **Top options for working with Excel** section, select **Show Developer tab in the Ribbon**.

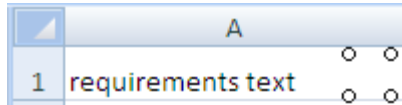


- 5** Click **OK**.
- 6** On the **Developer** tab, select **Design Mode**.



When you select **Design Mode**, the ActiveX control is no longer visible in the cell.

- 7 Click where the ActiveX control was, and you see four handles showing the location of the control.



- 8 Select **Home > Cut** to delete the control.

Linking to Custom Types of Requirements Documents

In this section...
“Built-In Link Types” on page 2-40
“Why Create a Custom Link Type?” on page 2-40
“Custom Link Type Registration” on page 2-41
“Link Properties” on page 2-42
“Link Type Properties” on page 2-42
“Creating a Custom Link Requirement Type” on page 2-44
“Navigating to Simulink Objects from External Documents” on page 2-53

Built-In Link Types

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

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

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

Why Create a Custom Link Type?

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

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

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

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

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

With custom link types, you can:

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

Custom Link Type Registration

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

```
rmi register mytargetfilename
```

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

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

```
rmi unregister mytargetfilename
```

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

Link Properties

Requirement links are the data structures, saved in the Simulink model, that identify a specific location within a document. You get and set the links on a block using the `rmi` command. The RMI encapsulates link information in 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 requirements. The document and ID fields of links uniquely identify the linked item in external documents. The RMI passes both of these strings to the navigation command when you navigate a link from the model.

Link Type Properties

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

Property	Description
Registration	The name of the M-file that creates the link type. The RMI stores this name in the Simulink model.
Label	A string to identify this link type. In the Requirements dialog box, this string appears on the Document type drop-down list for a Simulink or Stateflow object.
IsFile	A Boolean property that indicates if the linked documents are files within the computer file system. If a document is a file: <ul style="list-style-type: none">• The software uses the standard method for resolving the path.• In the Requirements dialog box, when you click Browse, the file selection dialog box opens.

Property	Description
Extensions	An array of file extensions. Use these file extensions as filter options in the Requirements dialog box when you click Browse . The file extensions infer the link type based on the document name. If you registered more than one link type for the same file extension, the link type that you registered takes first priority.
LocDelimiters	A string containing the list of supported navigation delimiters. The first character in the ID of a requirement specifies the type of identifier. For example, an identifier can refer to a specific page number (#4), a named bookmark (@my_tag), or some searchable text (?search_text). The valid location delimiters determine the possible entries in the Requirements dialog box Location drop-down list.
NavigateFcn	<p>The MATLAB callback you invoke when you click a link. The function has two input arguments: the document field and the ID field of the link:</p> <pre data-bbox="639 874 1225 899">feval(LinkType.NavigateFcn, Link.document, Link.id)</pre>
ContentsFcn	<p>The MATLAB callback you invoke when you click the Document Index tab in the Requirements dialog box. This function has a single input argument that contains the full path of the resolved function or, if the link type is not a file, the Document field contents.</p> <p>The function returns three outputs:</p> <ul data-bbox="605 1159 758 1281" style="list-style-type: none"> • Labels • Depths • Locations
BrowseFcn	The MATLAB callback you invoke when you click Browse in the Requirements dialog box. This function is not necessary when the link type is a file. The function takes no input arguments and returns a single output argument that identifies the selected document.

Creating a Custom Link Requirement Type

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

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

To create a custom link requirement type:

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

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

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

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

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

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

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

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

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

function openFileToLine(fileName, lineNum)
```

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

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

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

    % Find all the requirement items
    fList1 = 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
```

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

- 3** Create a text file with the .abc extension containing several requirement items marked by the Requirement:: string. For your convenience, an example file ships with your installation. The example file is demo_req_1.abc and resides in *matlabroot*\toolbox\slvnx\rmidemos. demo_req_1.abc contains the following content:

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

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

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

```
Description:
```

```
When the autopilot is in altitude climb
control mode, the controller maintains a
```

constant user-selectable target climb rate.

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

<END "Altitude Climb Control">

Requirement:: "Altitude Hold"

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

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.

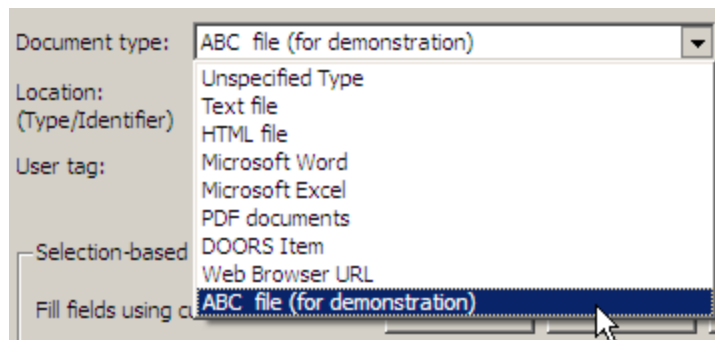
The user-selectable climb rate is always a positive number if the current altitude is above the target altitude the actual target climb rate is the negative of the user setting.

<END "Glide Slope Coupled">

- 4 Open the model `aero_dap3dof`.
- 5 Right-click the Reaction Jet Control subsystem and select **Requirements > Edit/Add Links**.

The Requirements dialog box opens.

- 6 Click **New** to add a new requirement link. The **Document type** drop-down list now contains the ABC file (for demonstration) option.



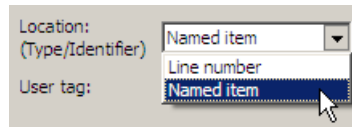
- 7 Set **Document type** to ABC file (for demonstration) and browse to the demo_req_1.abc file. The browser shows only the files with the .abc extension.
- 8 To define a particular location in the requirements document, use the **Location** field.

In this example, the rmicustabcinterface function specifies two types of location delimiters for your requirements:

- > — Line number in a file
- @ — Named item, such as a bookmark, function, or HTML anchor

Note The rmi reference page describes other types of requirements location delimiters.

The **Location** drop-down list contains these two types of location delimiters whenever you set **Document type** to ABC file (for demonstration).



- 9 Select **Line number**. Enter the number 26, which corresponds with the line number for the Altitude Hold requirement in demo_req_1.abc.
- 10 In the **Description** field, enter Altitude Hold, to identify the requirement by name.
- 11 Click **Apply**.
- 12 Verify that the Altitude Hold requirement links to the Reaction Jet Control subsystem. Right-click the subsystem and select **Requirements > 1. "Altitude Hold"**.

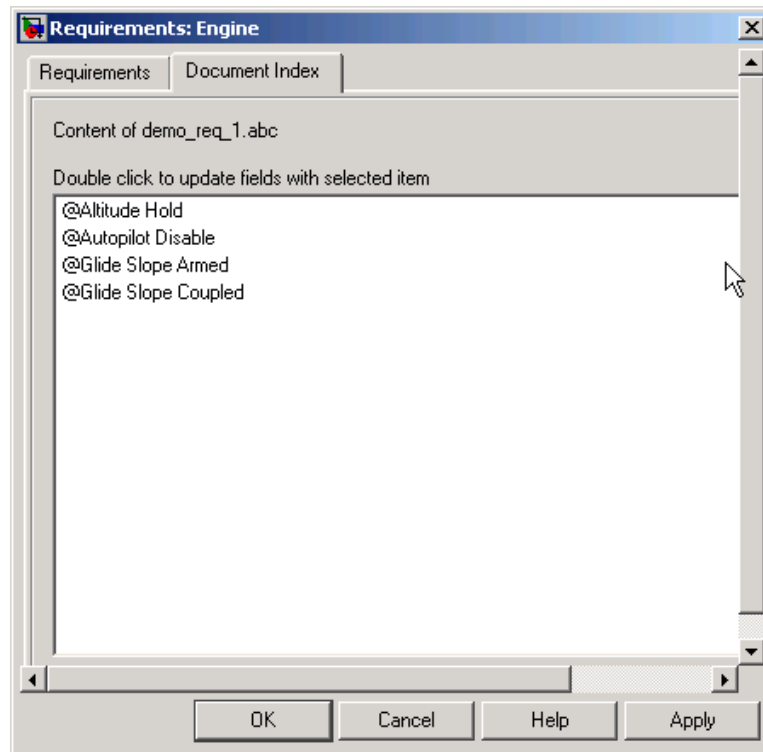
Creating a Document Index

A *document index* is a list of all the requirements in a given document. To create a document index, MATLAB M-code uses file I/O functions to read the contents of a requirements document into a MATLAB variable. Then the RMI extracts the list of requirement items.

The example requirements document, `demo_req_1.abc`, defines four requirements using the string `Requirement::`. To generate the document index for an ABC file, the `ContentsFcn` function, in the `rmicustabcinterface.m` file, extracts the requirements names and inserts @ before each name.

Note To see the code for the `ContentsFcn` file, go to step 1 in “Creating a Custom Link Requirement Type” on page 2-44.

For the `demo_req_1.abc` file, in the **Requirements: Engine** dialog box, click the **Document Index** tab. The `ContentsFcn` function generates the document index automatically.



Navigating to Simulink Objects from External Documents

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

Providing Unique Object Identifiers

Whenever you create a requirement link for a Simulink or Stateflow object, the RMI creates a globally unique identifier for that object. This identifier identifies the object. The identifier does not change if you rename or move the object, or add or delete requirement links. The RMI uses the unique identifier only to resolve an object within a model. The identifier is globally unique and

does not collide with identifiers in other models, unless the two models derive from the same original model. Unique object identifiers have formats such as:GIDa_cd14afcd_7640_4ff8_9ca6_14904bdf2f0f.

Using the `rmiobjnavigate` Function

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

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

Determining the Navigation Command

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

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

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

Using the ActiveX Navigation Control

The RMI uses software that includes a special Microsoft ActiveX control to enable navigation to Simulink objects from Microsoft Word and Excel® documents. You can use this same control in any other application that supports ActiveX within its documents.

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

Typical Code Sequence for Establishing Navigation Controls

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

To automate the procedure for establishing links:

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

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

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

For example, you can use the code for selection-based linking to the Microsoft Word, MicrosoftExcel, and IBM Rational DOORS software. The files are contained in `matlabroot\toolbox\slvnn\reqmgt\private`:

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

Using the System Requirements Block in a Model

In this section...

“About the System Requirements Block” on page 2-56

“Adding the System Requirements Block” on page 2-56

“Renaming the System Requirements Block” on page 2-57


About the System Requirements Block

To list all the requirements for a model or a subsystem in a Simulink model, add the System Requirements block from the Simulink Verification and Validation library. You can place this block anywhere in a model. You cannot connect this block to other Simulink blocks.

After you place the System Requirements block in a Simulink model, the block lists the requirement links for that model or subsystem. If a model includes a System Requirements block, that block automatically updates the listing as you add, modify, or delete requirements for the model or subsystem. Requirements associated with individual blocks in the model are not listed.

Adding the System Requirements Block

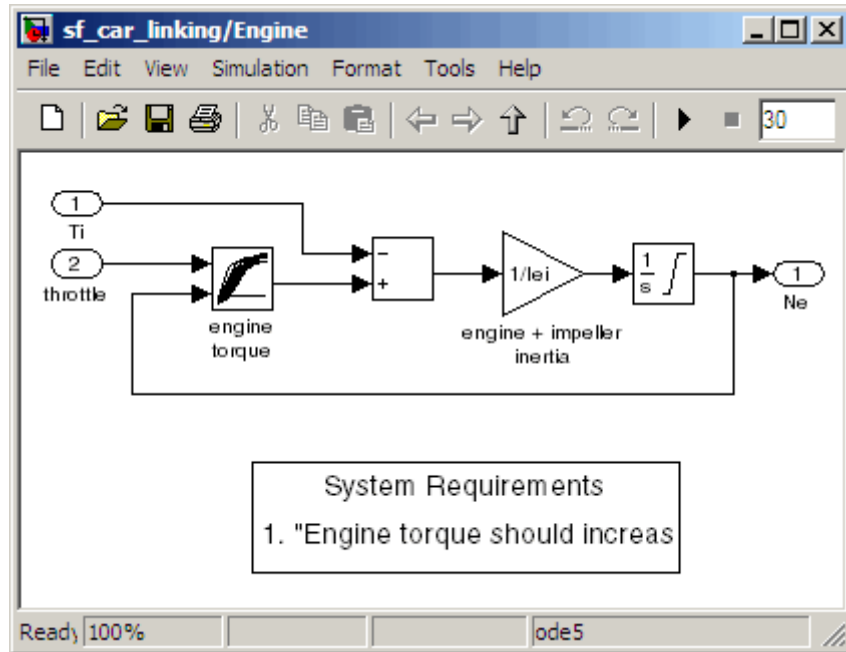
To list the requirements in the Engine subsystem of the `sf_car_linking` model:

- 1 Open the `sf_car_linking` model.
- 2 Double-click the Engine block to open it.
- 3 To open the Simulink Library Browser, click the Library Browser tool .
- 4 In the left pane of the Simulink Library Browser, select **Simulink Verification and Validation**.

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

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

The software automatically populates the block with the system requirement for the Engine subsystem.



- 6 Double-click the link to open the requirements document, requirements.docx, with the linked text highlighted.

Renaming the System Requirements Block

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

- 1 In the sf_car_linking/Engine subsystem, right-click the System Requirements block.
- 2 Select **Mask Parameters**. The Block Parameters dialog box opens.

- 3** In the **Block Title** field, type Engine Requirements and click **OK**.
The block title is updated in the subsystem.

Including Requirements Information with Generated Code

After you simulate your model and verify its performance against the requirements, you can generate code from the model for an embedded real-time application.

The Real-Time Workshop® Embedded Coder™ software generates code for Embedded Real-Time (ERT) targets. If the model has any links to requirements, while generating the code, the software also inserts hyperlinks to any linked requirements documents in the code comments.

For example, if a block has a requirement, the Real-Time Workshop Embedded Coder software generates code for that block. In the code comments for that block, the software inserts a hyperlink to the requirements document that contains the requirement associated with that block.

Note You must have a license for Real-Time Workshop Embedded Coder to generate code for an embedded real-time application.

Generated code includes requirements descriptions and hyperlinks to the requirements documents in the following locations.

Model Object	Requirements 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 does not have a nonvirtual parent, requirement descriptions appear in the main header file for the model, <model>.h.
Nonsubsystem blocks	In the generated code for the block

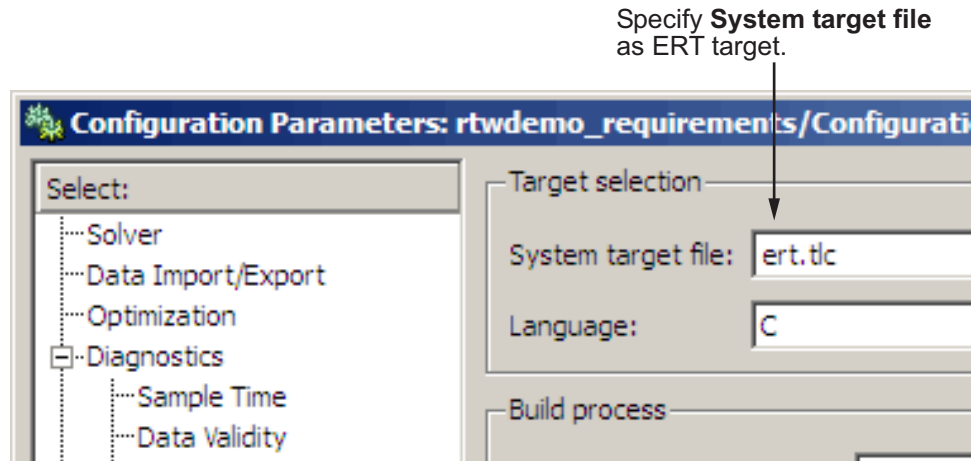
To specify that generated code of an ERT target include requirements:

- 1 Open the `rtwdemo_requirements` demo model.

2 Select **Simulation > Configuration Parameters**.

3 In the Configuration Parameters dialog box, on the **Select** pane, select the **Real-Time Workshop** node.

The currently configured system target must be an ERT target.



4 Under **Real-Time Workshop**, select **Comments**.

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

6 Under **Real-Time Workshop**, select **Report**.

7 On the **Report** pane, select:

- **Create code generation report**
- **Launch report automatically**

8 On the **Real-Time Workshop** pane, click **Build**.

9 In the code-generation report, open `rtwdemo_requirements.c`.

10 Scroll to the code for the Pulse Generator block, `clock`, for the hyperlink to the requirement linked to that block.

```
rtwdemo_requirements.c /* DiscretePulseGenerator: '<Root>/clock' *  
rt_zcfcn.h             * Block requirements for '<Root>/clock':  
rtwdemo_requirements.h * 1. Clock period shall be consistent with chirp tolerance  
                       */
```

- 11 Click the link to open the requirements document.

Note When you click a requirements link in the code, the software opens the application for the requirements document, except in DOORS software. To view a DOORS requirement document, start the DOORS software and log in before clicking the hyperlink in the code.

Managing Requirements with DOORS Software

- “Why Use DOORS Requirements with Simulink Objects?” on page 3-2
- “Configuring the Requirements Management Interface for DOORS Software” on page 3-4
- “Linking Simulink Objects to DOORS Requirements” on page 3-7
- “Synchronizing a Simulink Model to a DOORS Surrogate Module” on page 3-12
- “Viewing Simulink Objects with Requirements” on page 3-28
- “Creating Requirements Reports” on page 3-30
- “Creating Two-Way Links Between Requirements and Simulink Objects” on page 3-35

Why Use DOORS Requirements with Simulink Objects?

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

Simulink models represent functional designs of embedded systems that you build according to specifications and requirements that are managed in DOORS software. Linking Simulink objects to DOORS requirements allows you to:

- Provide traceability from Simulink models to DOORS requirements, which is mandatory for many high-integrity software and hardware projects.
- Access requirements information from each of the linked Simulink objects.
- Use DOORS linking, requirements-analysis, and report-generation capabilities for information captured in both DOORS requirements and Simulink models.

Linking DOORS requirements to Simulink objects allows you to:

- Share design details with engineers who are not directly familiar with the design.
- Improve requirements reviews.

Using RMI and DOORS software, you can:

- Create links between Simulink objects, such as blocks, signals, and subsystems, to DOORS requirements.
- Navigate between Simulink objects and DOORS requirements with links that you created using RMI.
- Create a representation of your Simulink model (called a *surrogate module*) in the DOORS software. Using the surrogate module, you can create and review requirements without modifying the requirements document and without having to use the Simulink software.

- Create a report for your Simulink model that shows which objects have links to requirements.

Configuring the Requirements Management Interface for DOORS Software

In this section...
“Before You Begin” on page 3-4
“Installing DOORS Software” on page 3-4
“Manually Installing Additional Files for DOORS Software” on page 3-4
“Upgrading DOORS Software” on page 3-5

Before You Begin

IBM Rational DOORS software is a requirements-management application for capturing, tracking, and managing requirements. If you plan to use DOORS software with the RMI, you must install additional files to establish communication between the DOORS application and the Simulink software. The sections that follow describe the installation and configuration procedures.

Installing DOORS Software

You can install the DOORS software before or after you install the RMI. In either case, at the MATLAB command line, run the RMI setup script to copy all the necessary files to the correct location:

```
rmi setup
```

Manually Installing Additional Files for DOORS Software

The setup script automatically copies all the files to the correct location. However, in some cases the script fails because of file permissions in your DOORS installation. If the script fails, manually install additional files.

- 1 Close the DOORS software if it is running.
- 2 Copy the following files from `matlabroot\toolbox\slvnx\reqmgt` to the `<doors_install_dir>\lib\dxl\addins` folder.

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

Replace any existing versions of the files if you have not modified them; otherwise, merge their contents.

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

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

Replace any existing versions of these files.

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

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

Upgrading DOORS Software

If you upgrade your DOORS software after installing the RMI, run the setup script (`rmi setup`) again.

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

- 1 In your DOORS installation folder, navigate to the subfolder `...\lib\dxl\startupFiles`.
- 2 In a text editor, open the file `copiedFromDoors7.dxl`.
- 3 Add `//` before this line to comment it out:

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

- 4 Save and close the file.
- 5 Start the DOORS and MATLAB software.

- 6 Run the setup script. At the MATLAB command line, type `rmi setup`.

Linking Simulink Objects to DOORS Requirements

In this section...

“Creating DOORS Requirements” on page 3-7

“Creating One-Way Links from Simulink Objects to DOORS Requirements” on page 3-8

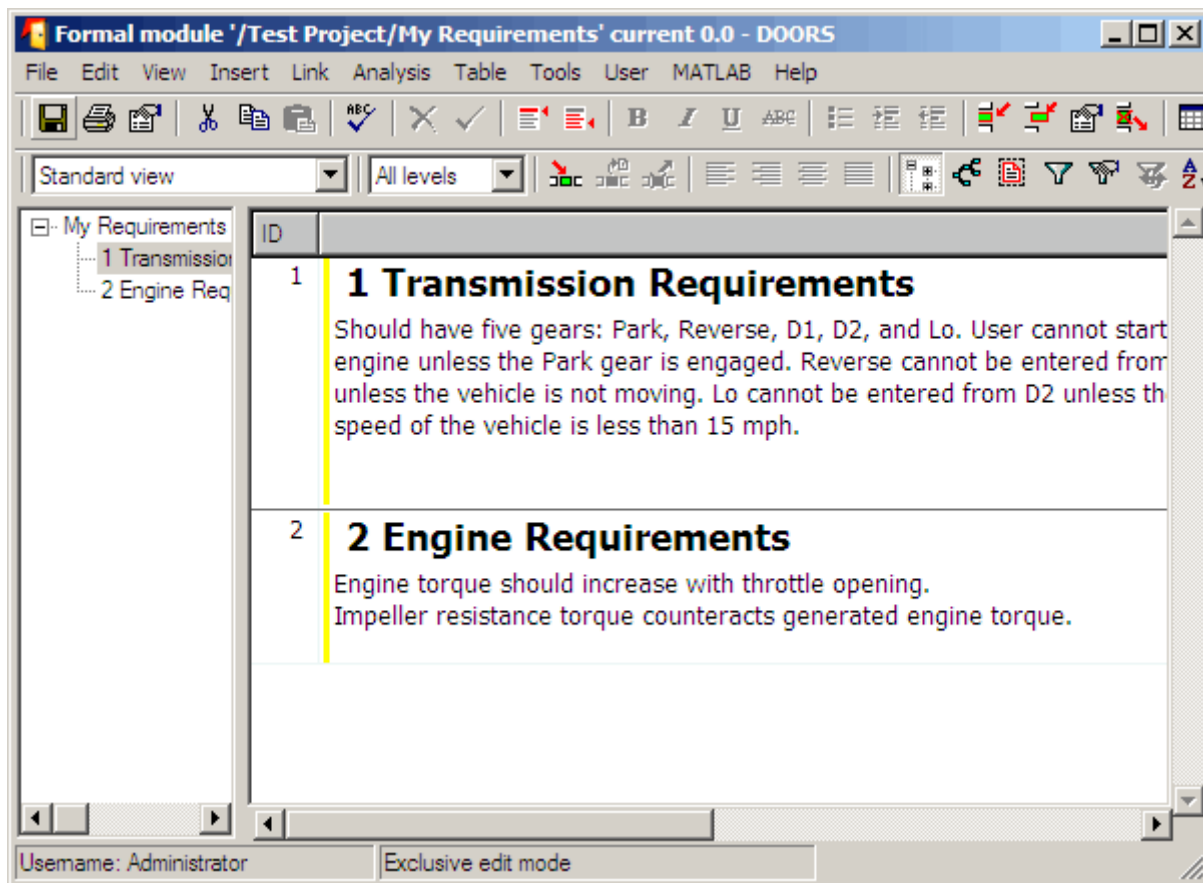
“Navigating from a Simulink Object to a DOORS Requirement” on page 3-10

Creating DOORS Requirements

Use the RMI to create a link from a Simulink object to a DOORS requirement. The following example describes how to create this link.

To begin this example, create a DOORS project and formal module, and insert two requirements:

- 1 Start the DOORS software.
- 2 Create a project named **Test Project** and a new formal module named **Requirements**.
- 3 In the **My Requirements** module, insert a new object.
- 4 In the **Object 1** properties dialog box, enter the **Heading** **Transmission Requirements**, some text for **Object Text**, and click **OK**. (You do not need to enter any **Short Text**.)
- 5 In the **My Requirements** module, insert another new object.
- 6 In the **Object** properties dialog box, enter the **Heading** **Engine Requirements**, some text for **Object Text**, and click **OK**. (You do not need to enter any **Short Text**.)
- 7 Save your changes.



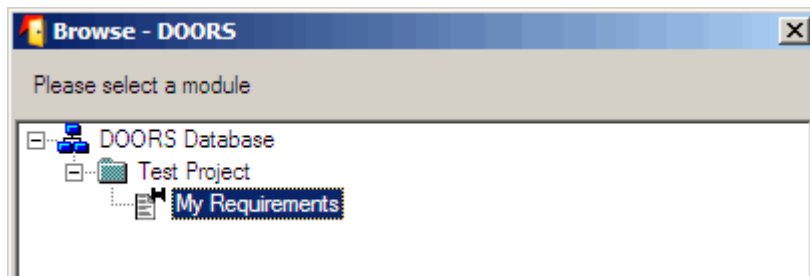
Creating One-Way Links from Simulink Objects to DOORS Requirements

You can create one-way links from Simulink objects to DOORS requirements without having to modify the requirements.

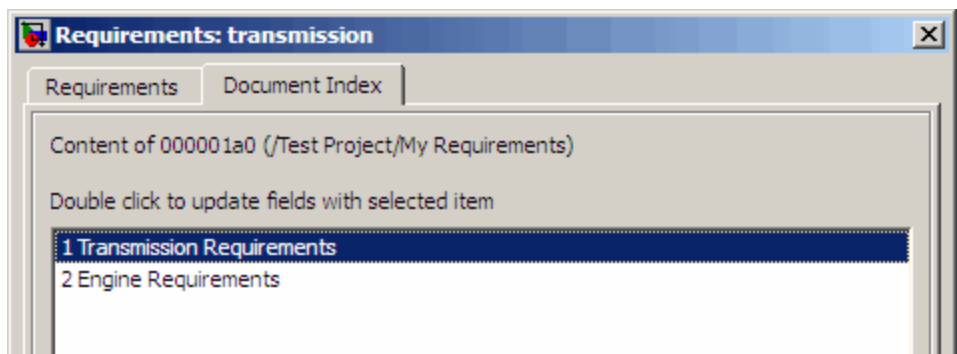
For this example, and for the following examples, you create a writable copy of the `sf_car` model that ships with your Simulink software.

To create a one-way link from a Simulink object to a requirement:

- 1 In the MATLAB Command Window, open the demo model `sf_car.mdl` and save it as `sf_car_doors`.
- 2 In the model, right-click the transmission subsystem and select **Requirements > Edit/Add Links**.
- 3 In the Requirements dialog box, click **New**.
- 4 For **Document type**, select DOORS Item.
- 5 Click **Browse**.
- 6 Browse to and select the My Requirements module. Click **OK**.



- 7 To list the two requirements that you created in the My Requirements module, click the **Document Index** tab.



- 8 Select 1 Transmission Requirements and click **OK**.

9 You have created a one-way link from the transmission subsystem in the model to the Transmission Requirements requirement in the DOORS database. To verify the link, in the `sf_car_doors` model, right-click the transmission subsystem and select Requirements to see the option **Requirements > 1. “1 Transmission Requirements”**.

10 Save the `sf_car_doors` model.

DOORS IDs

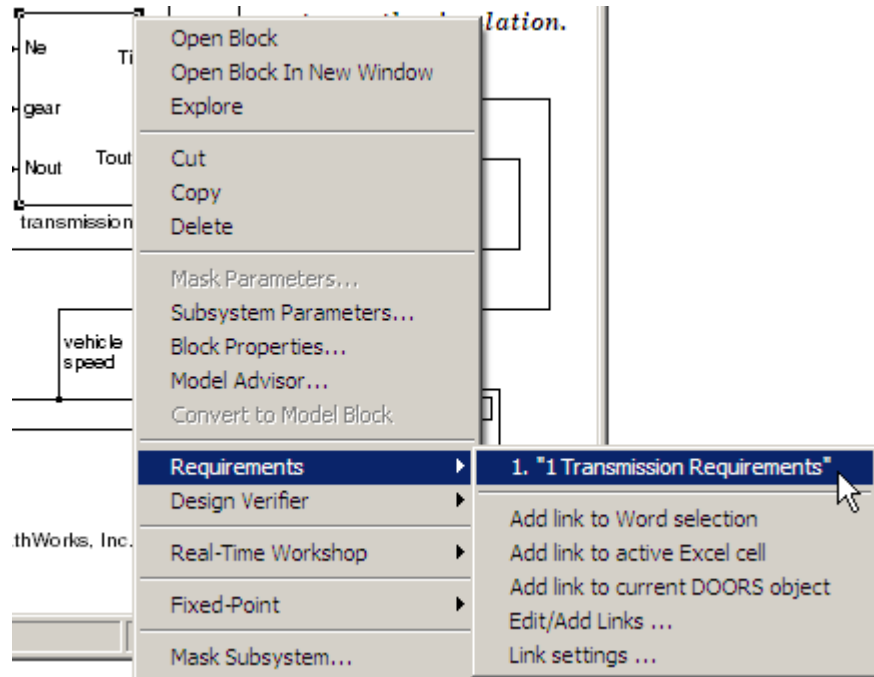
The DOORS software gives each requirement in the formal module a unique ID. In the My Requirements formal module you created, the IDs are 1 and 2. If you rename the module or rearrange the objects in the module, the IDs remain unchanged.

The DOORS software also assigns each module a unique eight-digit hexadecimal number. RMI uses that number, along with the DOORS ID, to locate requirements. In this example, the Requirements dialog box lists 000001a0 as the My Requirements formal module ID. When you execute the preceding steps, the formal module ID may be different.

Navigating from a Simulink Object to a DOORS Requirement

With one-way links, you can navigate from a Simulink object to a requirement in the DOORS database. To navigate from the transmission subsystem in your `sf_car_doors` model using the link that you created:

1 Right-click the transmission subsystem and select **Requirements > 1. “1 Transmission Requirements”**



The My Requirements module opens to the Transmission Requirements object.

Synchronizing a Simulink Model to a DOORS Surrogate Module

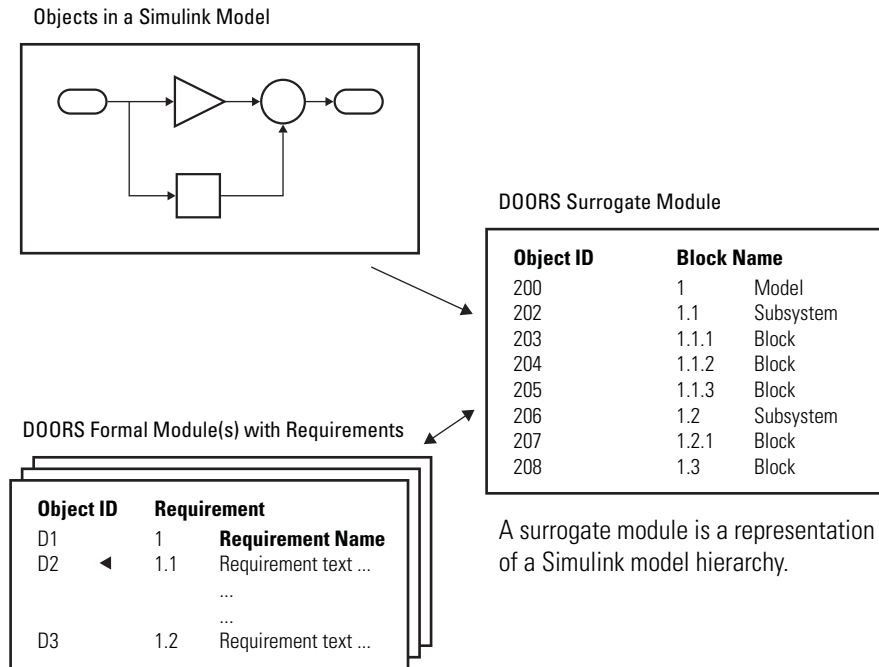
In this section...
“What Is a Surrogate Module?” on page 3-12
“What Is Synchronization?” on page 3-12
“Advantages of Synchronization” on page 3-14
“Synchronizing a Simulink Model to Create a Surrogate Module” on page 3-14
“Customizing the Synchronization” on page 3-16
“Updating the Surrogate Module to Reflect Model Changes” on page 3-22
“Navigating Using the Surrogate Module” on page 3-25

What Is a Surrogate Module?

A *surrogate module* is a DOORS formal module that is a representation of a Simulink hierarchy. You use standard DOORS capabilities to navigate between the Simulink objects in the surrogate module and requirements in other modules.

What Is Synchronization?

Synchronization creates a DOORS surrogate module. The surrogate module facilitates navigation between the Simulink object and the requirements, as the following diagram illustrates.



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

When you synchronize a model for the first time, the DOORS software creates the surrogate module that contains a representation of the model, depending on your synchronization settings. (To customize the synchronization, see “Customizing the Level of Detail in Synchronization” on page 3-19.)

If you create new links or remove existing links, you can resynchronize the model. The new or updated surrogate module reflects any changes in the requirements links since the previous synchronization.

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

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

Advantages of Synchronization


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

- You can navigate from a requirement to a Simulink object without modifying the requirements modules.
- You avoid cluttering your requirements modules with inserted navigation objects.
- The DOORS database contains complete information about requirements links. You can review requirements links and verify traceability, even if the Simulink software is not running.
- You can use DOORS reporting features to analyze requirements coverage.
- You can separate the work of the systems engineers in charge of requirements tracking from the work of the Simulink model developers:
 - Systems engineers can establish requirements links to models without using the Simulink software.
 - Model developers can capture the requirements information and store it with the model.
- You can resynchronize a model with a new surrogate module, updating any model changes, or specifying different synchronization options.

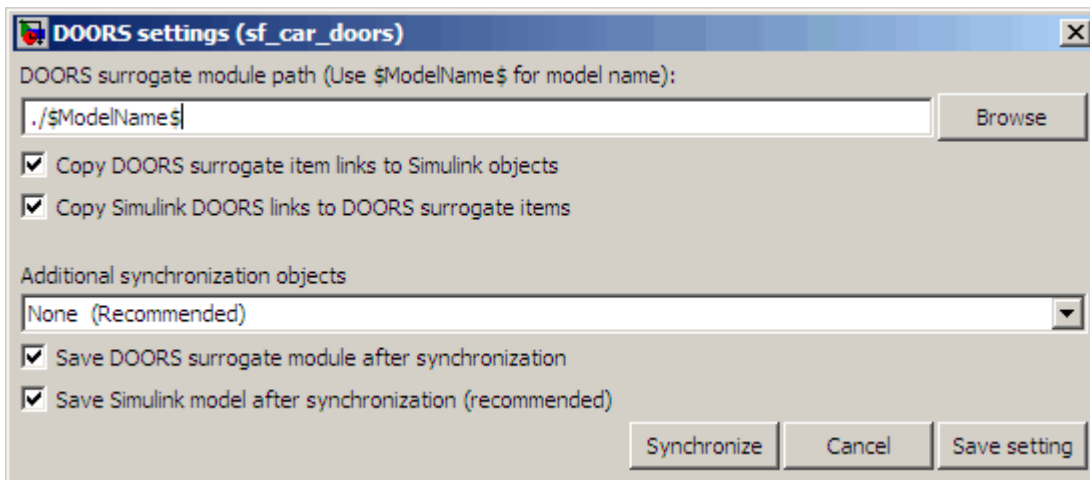
Synchronizing a Simulink Model to Create a Surrogate Module

The first time that you synchronize your model with the DOORS software, the surrogate module is created in the DOORS database.

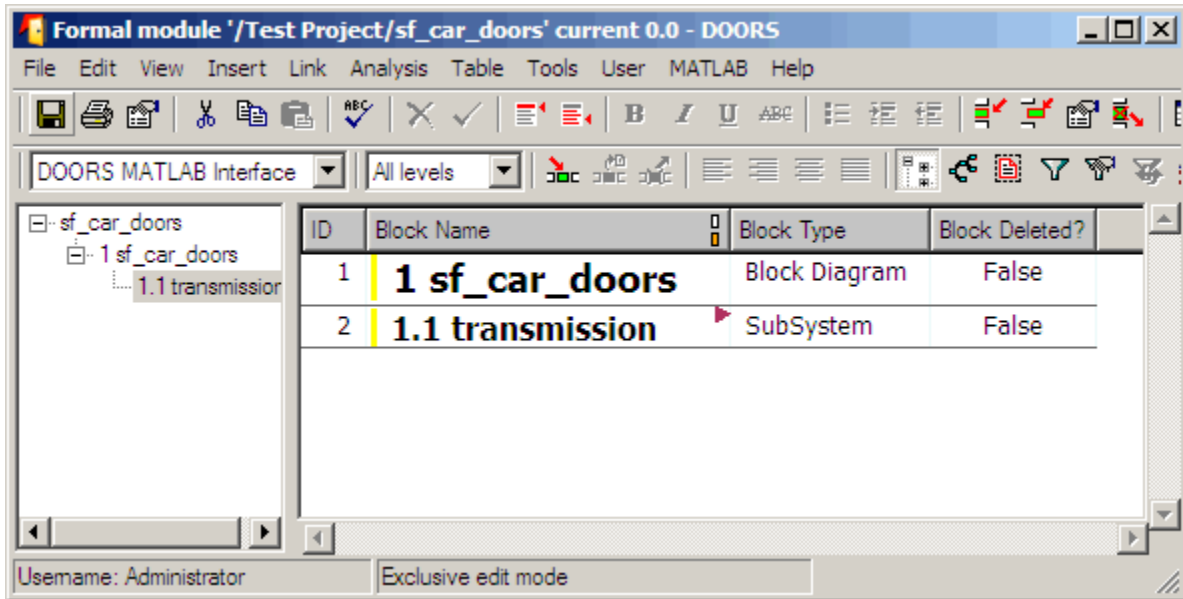
To synchronize the `sf_car_doors` model with the DOORS software:

- 1 Make sure that the DOORS software is running and that the Test Project is open.
- 2 Open the `sf_car_doors` model used in the preceding examples.
- 3 In the Model Editor, select **View > Model Explorer**.
- 4 In the Model Explorer, select the Synchronize Requirements with DOORS tool .

The DOORS settings dialog box opens.



- 5 For this exercise, accept all the default synchronization options. For more information about the options, see "Customizing the Synchronization" on page 3-16 .
- 6 Click **Synchronize** to create and open a surrogate module for all DOORS requirements that have links to objects in the `sf_car_doors` model.



The surrogate module contains a DOORS object that is linked to the transmission subsystem; the only object in the model with a requirement. The red arrow indicates a link from the surrogate module object to the requirement.

7 Save the surrogate module.

Customizing the Synchronization

- “DOORS Synchronization Settings” on page 3-17
- “Resynchronizing a Model with a Different Surrogate Module” on page 3-18
- “Customizing the Level of Detail in Synchronization” on page 3-19
- “Resynchronizing to Include All Simulink Objects” on page 3-20
- “Detailed Information About Surrogate Modules” on page 3-21

DOORS Synchronization Settings

DOORS Settings Option	Description
DOORS surrogate module path	Names the surrogate module using the Simulink model name.
Copy DOORS surrogate item links to Simulink objects	<p data-bbox="872 439 1322 560">Copies all links between the surrogate module and the requirements modules into the Simulink model.</p> <hr/> <p data-bbox="872 630 1322 881">Note If you delete a requirement link from the model only, and then resynchronize the model, the synchronization restores the link. Delete the requirement links from both the model and the surrogate module so that resynchronization does not restore the link.</p>
Copy Simulink DOORS links to DOORS surrogate items	Copies all links between Simulink objects and requirements to the surrogate module. The surrogate module objects link to the same requirements as their Simulink objects.
Additional synchronization objects	The default setting (None) specifies to synchronize only those Simulink objects that have linked requirements. For more information about the other synchronization options, see “Customizing the Level of Detail in Synchronization” on page 3-19.

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

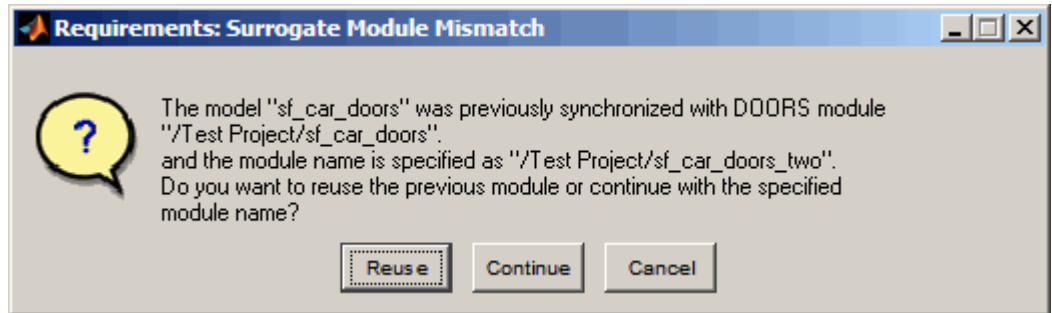
Resynchronizing a Model with a Different Surrogate Module

To create a surrogate module that has more or less detail about the model hierarchy, resynchronize a model with the same or new surrogate module. In the DOORS settings dialog box, select the **DOORS surrogate model module path** option to name the different surrogate module in the DOORS database.

Specify a module with either a relative path (starting with ./) or a full path (starting with /). The software appends relative paths to the current DOORS project. Absolute paths must specify a project and a module name.

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

If you select a new module path or if you have renamed the surrogate module, and you click **Synchronize**, the Requirements: Surrogate Module Mismatch dialog box opens.



Click **Continue** to create a new surrogate module with the new path or name.

Customizing the Level of Detail in Synchronization

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

In “Synchronizing a Simulink Model to Create a Surrogate Module” on page 3-14, you synchronized the model with the **Additional synchronization objects** option set to None. As a result, the surrogate module contains only Simulink objects that have requirement links. Additional synchronization options, described in this section, can increase the level of surrogate detail. However, increasing the level of surrogate detail can slow down synchronization.

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

Drop-Down List Option	Description
None (Recommended)	Maps only Simulink objects that have requirements links and their parent objects to the surrogate module.
Minimal - Nonempty unmasked subsystems and Stateflow charts	Adds all nonempty Stateflow charts and unmasked Simulink subsystems to the surrogate module.

Drop-Down List Option	Description
Moderate - Unmasked subsystems, Stateflow charts, and superstates	Adds Stateflow superstates to the surrogate module.
Average Nontrivial Simulink blocks, Stateflow charts and states	Adds all Stateflow charts and states and Simulink blocks, except for trivial blocks such as ports, bus objects, and data-type converters, to the surrogate module.
Extensive - All unmasked blocks, subsystems, states and transitions	Adds all unmasked blocks, subsystems, states, and transitions to the surrogate module.
Complete - All blocks, subsystems, states and transitions	Copies <i>all</i> blocks, subsystems, states, and transitions to the surrogate module.

Resynchronizing to Include All Simulink Objects

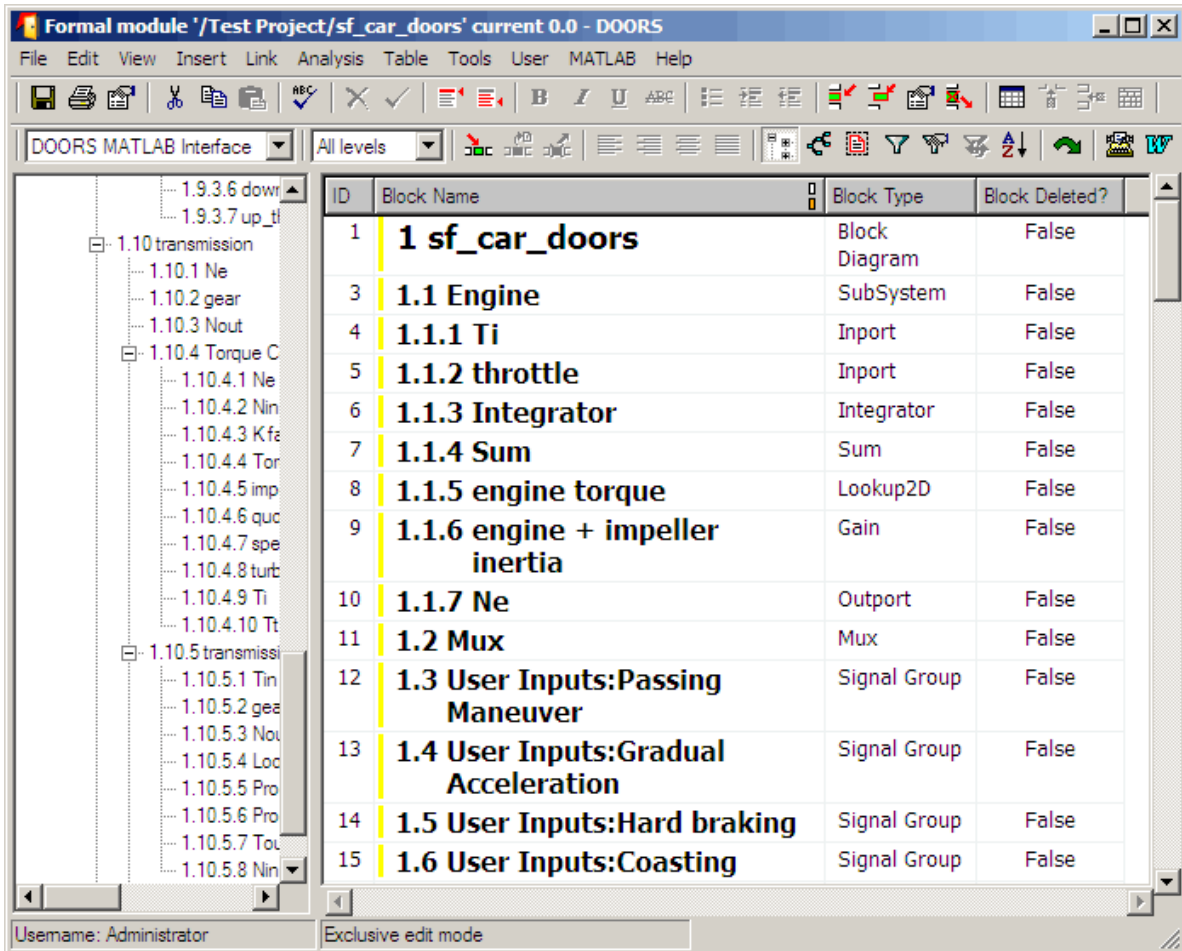
To include *all* Simulink objects in the DOORS surrogate module:

- 1** Open the `sf_car_doors` model.
- 2** In the Model Editor, select **Tools > Requirements > Synchronize with DOORS**.

The DOORS settings dialog box opens.

- 3** To resynchronize with the same surrogate module, make sure that the **DOORS surrogate module path** option reads `/Test Project/sf_car_doors`, which is the surrogate module that you created in “Synchronizing a Simulink Model to Create a Surrogate Module” on page 3-14.
- 4** To update the surrogate module to include *all* objects in your model, from the drop-down list under **Additional synchronization objects**, select **Complete - All blocks, subsystems, states and transitions**.
- 5** Click **Synchronize**.

You see the DOORS surrogate module for the sf_car_doors model. All Simulink objects and all Stateflow objects are now mapped in the surrogate module.



ID	Block Name	Block Type	Block Deleted?
1	1 sf_car_doors	Block Diagram	False
3	1.1 Engine	SubSystem	False
4	1.1.1 Ti	Inport	False
5	1.1.2 throttle	Inport	False
6	1.1.3 Integrator	Integrator	False
7	1.1.4 Sum	Sum	False
8	1.1.5 engine torque	Lookup2D	False
9	1.1.6 engine + impeller inertia	Gain	False
10	1.1.7 Ne	Outport	False
11	1.2 Mux	Mux	False
12	1.3 User Inputs:Passing Maneuver	Signal Group	False
13	1.4 User Inputs:Gradual Acceleration	Signal Group	False
14	1.5 User Inputs:Hard braking	Signal Group	False
15	1.6 User Inputs:Coasting	Signal Group	False

6 Save the surrogate module.

Detailed Information About Surrogate Modules

Notice the following about the surrogate module in the preceding graphic:

- The name of the surrogate module is `sf_car_doors`, as you specified in the Requirements settings dialog box.
- The left pane displays a node for each synchronized object in your model.
- The right pane displays a DOORS object for each Simulink object.
- Each Simulink object has a unique ID in the surrogate module. For example, the ID for the surrogate module object associated with the Mux block in the preceding figure is 11.
- The **Block Type** column identifies each object as a particular block or a subsystem.
- If you delete a previously synchronized object from your model and then resynchronize, the **Block Deleted** column reads **true**. Otherwise, it reads **false**.

If you delete any DOORS object, explicitly purge it to remove the object from the module. For information on how to purge an object, see the DOORS documentation.

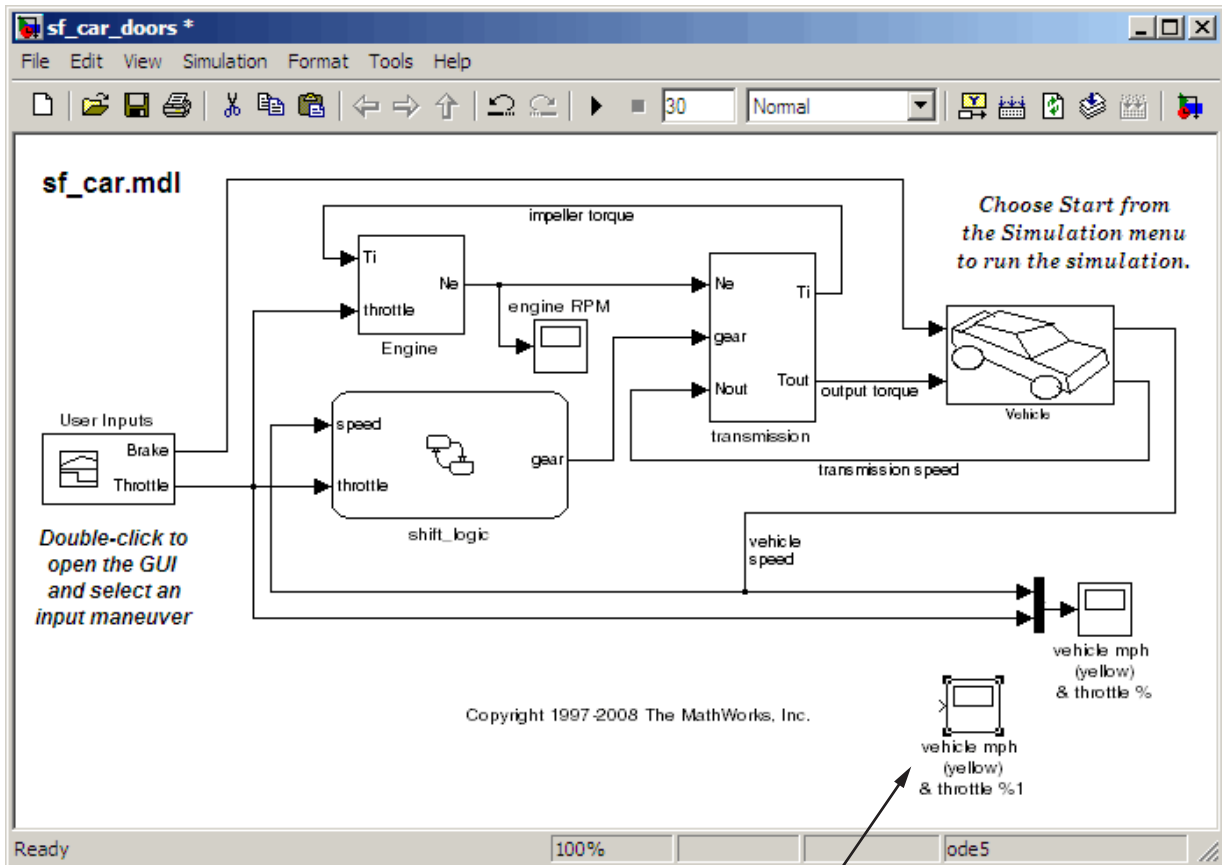
- In the previous exercises, before synchronization, you added the link from the transmission object in `sf_car_doors` to the Transmission Requirements requirement. In the surrogate module, the transmission object retains its ID (2), but the transmission object appears farther down in the module, in hierarchical order. The transmission object in the surrogate module has a red arrow indicating that it links to a Simulink object.

Updating the Surrogate Module to Reflect Model Changes

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

In the next example, you add a new block to the `sf_car_doors` model, and later delete it, resynchronizing after both steps:

- 1 In the `sf_car_doors` model, make a copy of the vehicle mph (yellow) & throttle % Scope block.



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

3 In the DOORS settings dialog box, leave the **Additional synchronization objects** option set to Complete - All blocks, subsystems, states, and transitions, and click **Synchronize**.

The software updates the surrogate module with the new block.

88	1.10.5.8 Nin	Outport	False
89	1.10.6 Ti	Outport	False
90	1.10.7 Tout	Outport	False
91	1.11 vehicle mph (yellow) & throttle %	Scope	False
92	1.12 vehicle mph (yellow) & throttle %1	Scope	False

New block and link

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

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

87	1.10.5.7 Tout	Outport	False
88	1.10.5.8 Nin	Outport	False
89	1.10.6 Ti	Outport	False
90	1.10.7 Tout	Outport	False
91	1.11 vehicle mph (yellow) & throttle %	Scope	False
93	1.12 vehicle mph (yellow) & throttle %1	Scope	True

- 5 Right-click the object and select **Delete** to delete this entry from the surrogate module.

6 Save the surrogate module.

7 Save the sf_car_doors model.

Navigating Using the Surrogate Module

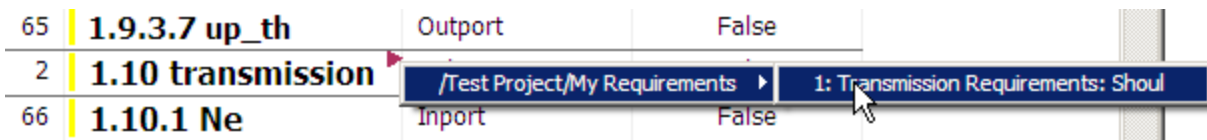
- “Navigating Between Requirements and the Surrogate Module in the DOORS Database” on page 3-25
- “Two-Way Navigation Using the Surrogate Module” on page 3-26

Navigating Between Requirements and the Surrogate Module in the DOORS Database

The requirements in the formal module and the surrogate module are both in the DOORS database. You can review the requirements and the Simulink objects in the surrogate module without starting the Simulink software. When you synchronize your model, the DOORS software creates links between the surrogate module objects and the requirements in the DOORS database.

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

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

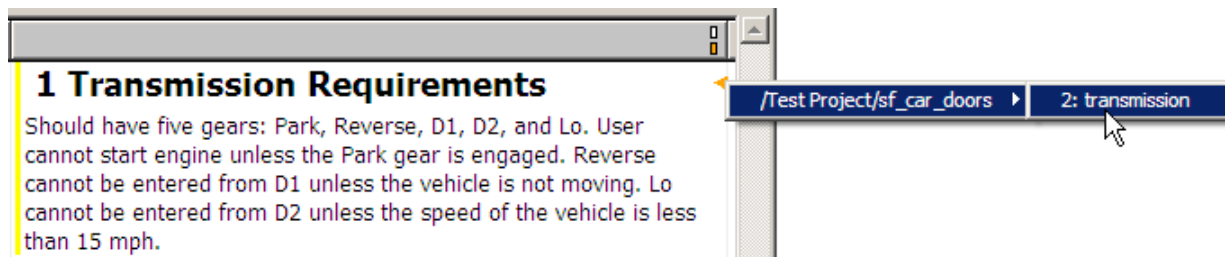


- 2 Select the requirement name.

The My Requirements module opens, scrolled to the Transmission Requirements object.

To navigate from the requirement in the My Requirements module to the surrogate module:

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



- 2 Select the object name.

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

Two-Way Navigation Using the Surrogate Module

If you synchronize your model, you can navigate between Simulink objects and DOORS requirements using the surrogate module as an intermediary.

Navigating from a Simulink Object to a Requirement. To navigate from the transmission subsystem in the sf_car_doors model to the linked requirement in the My Requirements formal module:

- 1 In the sf_car_doors model, right-click the transmission subsystem and select **Requirements > 1. “DOORS Surrogate Item”**.

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

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

The My Requirements module opens, scrolled to the Transmission Requirements requirement.

Navigating from a Requirement to the Model. To navigate from the Transmission Requirements object module to the transmission subsystem in the sf_car_doors model:

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

2 In the surrogate module, select the path to the linked object: **/Test Project/sf_car_doors > 2. transmission.**

The surrogate module opens, scrolled to the Transmission Requirements.

3 In the surrogate module, select the transmission object.

4 Select **MATLAB > Select item.**

The sf_car_doors model opens as follows:

- For a Simulink object, the Model Editor opens with that block or subsystem, and all its parent blocks, highlighted.
- For a Stateflow object, the diagram containing the selected object opens with the object highlighted.

Viewing Simulink Objects with Requirements

In this section...
“Viewing Objects with Requirements in the Model Editor” on page 3-28
“Viewing Objects with Requirements in the Model Explorer” on page 3-28

Viewing Objects with Requirements in the Model Editor

RMI lets you easily distinguish Simulink objects with requirements from objects without requirements.

To highlight Simulink objects with requirements using the Model Editor, select **Tools > Highlight model**. The RMI software highlights all objects in the model hierarchy that have associated requirements.

Viewing Objects with Requirements in the Model Explorer

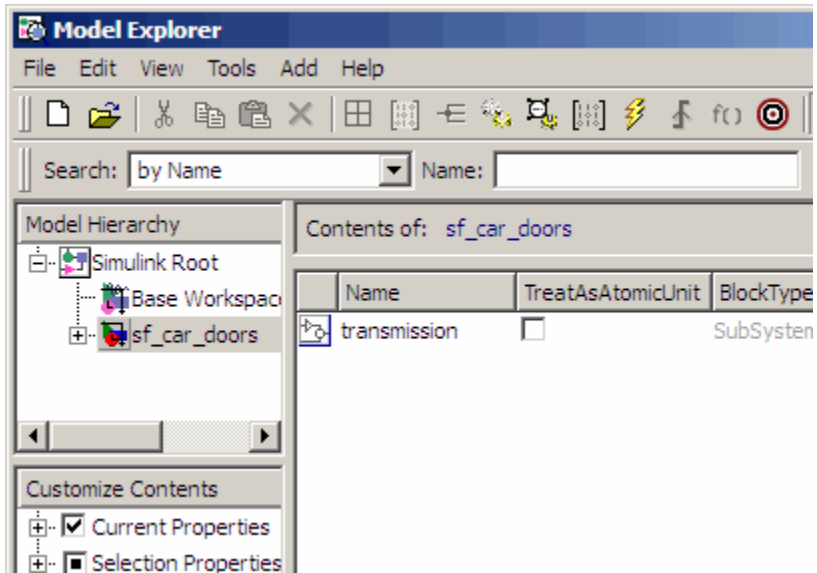
You can also highlight Simulink objects with requirements using the Model Explorer:

- 1 In the Model Explorer, with the `sf_cars_doors` model open, select **View > Model Explorer**.

In the Model Explorer window, in the **Model Hierarchy** pane, you see the highlighted model.

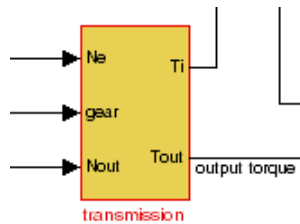
- 2 In the Model Explorer, click the Display Objects with Linked Requirements tool .

The Model Explorer displays only the transmission subsystem, which you added a requirement to in “Linking Simulink Objects to DOORS Requirements” on page 3-7.



- 3 Click the Highlight Items with Requirements on Model tool .

The sf_car_doors model highlights the transmission subsystem.



Creating Requirements Reports

In this section...
“About Requirements Reports” on page 3-30
“Creating a Default Requirements Report for a Model” on page 3-30
“Customizing a Requirements Report with Links to DOORS Requirements” on page 3-31

About Requirements Reports

Requirements reports contain information about DOORS requirements and their implementation in a Simulink model:

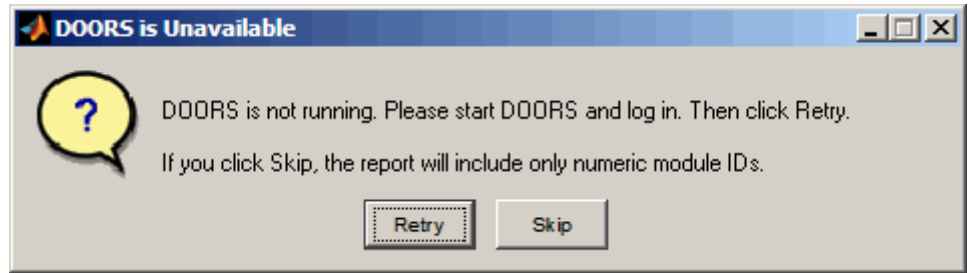
- The date the DOORS requirements module was last modified and by whom
- The number of links from the Simulink model to DOORS requirements modules
- The details about links from Simulink objects to their linked DOORS requirements

This section describes only those areas of the report that are specific to DOORS requirements. For more detailed information about the RMI reports, see “Creating a Requirements Report” on page 2-22.

Creating a Default Requirements Report for a Model

To create a requirements report for the sf_car_doors model:

- 1** Open the sf_car_doors model.
- 2** Select **Tools > Requirements > Generate Report**.
- 3** If the DOORS software is not running, you see the following dialog box.



If you click **Skip**, the report does not include information about when you last modified the DOORS requirement.

- 4 Start the DOORS software.
- 5 Click **Retry** to continue.

By default, the report has the file name `requirements.html`. RMI stores the report under the MATLAB Current Folder.

Customizing a Requirements Report with Links to DOORS Requirements

RMI uses the Simulink Report Generator software to generate the requirements report. You can customize the report using the RMI, or you can use the Simulink Report Generator software for advanced customization.

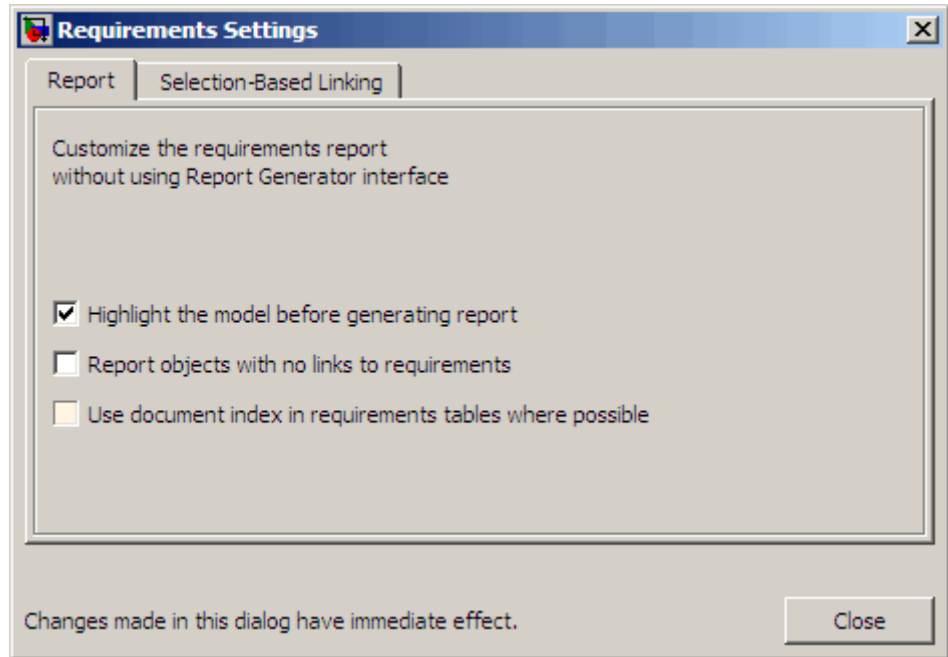
Customizing the Report Using the RMI

To customize the requirements report in the Model Editor:

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

The Requirements Settings dialog box opens.

- 2 Click the **Report** tab.



The options that you select in the Requirements Settings dialog box determine the contents of the report.

Requirements Settings Report Option	Description
Highlight the model before generating report	Highlights the Simulink objects with requirements in the Model Editor before creating the report and highlights them in the report.
Report objects with no links to requirements	Lists Simulink objects that have no requirements.
Use document index in requirements tables where possible	Uses a document ID instead of a path name in the requirements table, if ID is available.

3 Select the options that you want and click **Close**.

Run the report to generate the requirements report for your model.

Customizing the Report Using the Simulink Report Generator Software

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

To customize the requirements report, first start the Simulink Report Generator software. At the MATLAB command prompt, enter the following command:

```
setedit requirements
```

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

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

In addition to the standard report components, Simulink Report Generator provides RMI-specific components. These components insert information about Simulink objects whether or not they have associated requirements:

- **Missing Requirements Block Loop** — Applies all child components to blocks that have no requirements
- **Missing Requirements System Loop** — Applies all child components to systems that have no requirements
- **Requirements Block Loop** — Applies all child components to block that have requirements
- **Requirements Documents Table** — Inserts table that lists requirements documents
- **Requirements Signal Loop** — Applies all child components to signal groups with requirements

- **Requirements Summary Table** — Inserts property table listing blocks that have requirements and requirements details
- **Requirements System Loop** — Applies all child components to systems with requirements
- **Requirements Table** — Inserts table that lists system and subsystem requirements

There are several ways you can customize the requirements report:

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

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

Creating Two-Way Links Between Requirements and Simulink Objects

In this section...

“Creating Two-Way Links” on page 3-35

“Navigating Two-Way Links” on page 3-36

Creating Two-Way Links

Two-way links between Simulink objects and DOORS requirements allow you to navigate from the model to the requirements and from the requirements to the model.

Using this feature, the RMI inserts a link object into your requirements module. To create two-way links without modifying your requirements module, see “Synchronizing a Simulink Model to a DOORS Surrogate Module” on page 3-12.

To create a two-way link between the transmission subsystem in `sf_car_doors` and the `Transmission Requirements` requirement:

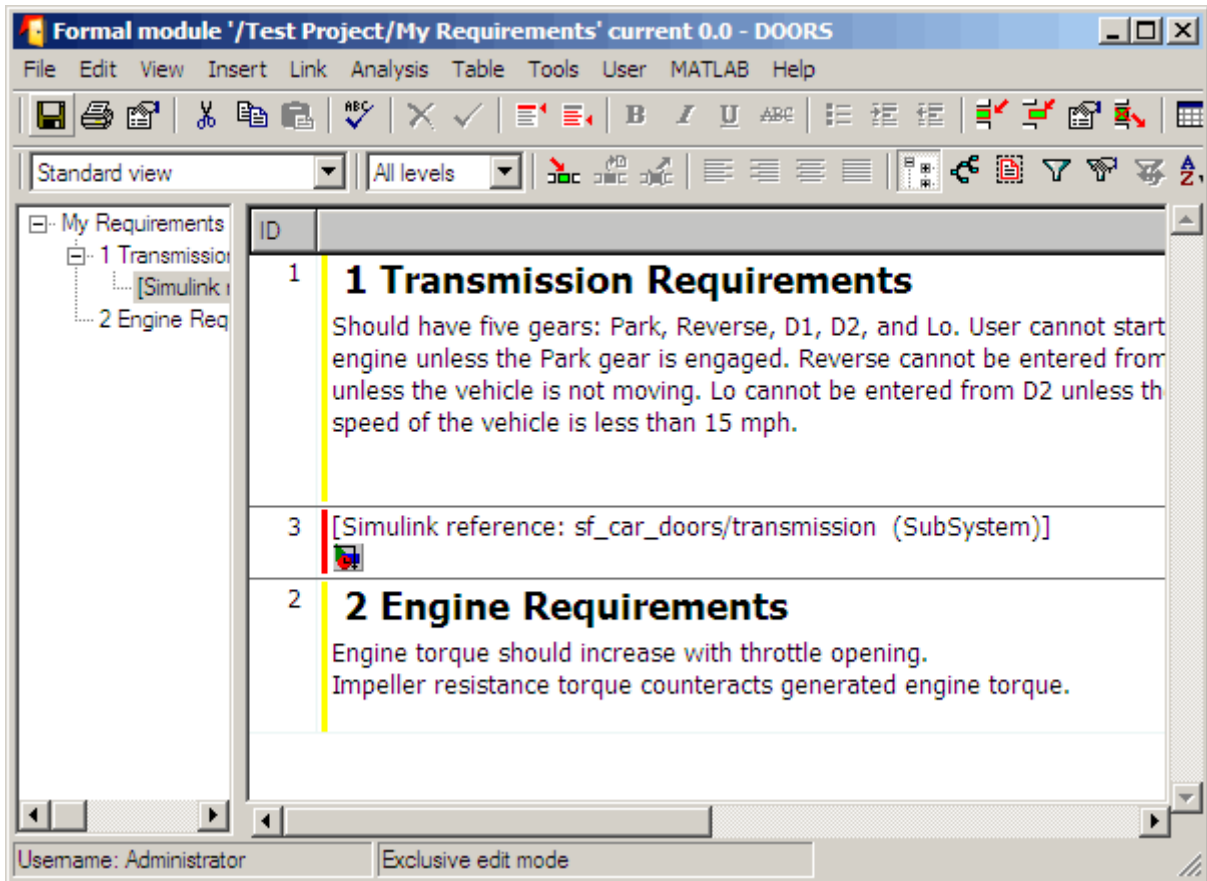
- 1 Open the `sf_car_doors.mdl` model.
- 2 In the DOORS formal module `My Requirements`, in the left pane, click the `Transmission Requirements` node.

Note Always select the requirement node by clicking the requirement name in the left pane. Do not use the up and down arrow keys on the keyboard to select a requirement node.

- 3 In the Model Editor, right-click the transmission subsystem and select **Requirements > Add link to current DOORS object**.

The RMI adds the link to the DOORS requirement. The DOORS software inserts a link object below the `Transmission Requirements` object that

represents the link from the requirement to the transmission subsystem in the sf_car_doors model.



4 Save the DOORS module.

5 Save the sf_car_doors model.

Navigating Two-Way Links

Once you create a two-way link, you can navigate in both directions between a Simulink object and a DOORS requirement. For information about how to navigate from the Simulink object to a requirement, see "Navigating from a

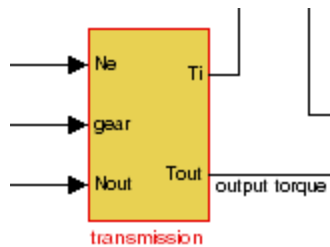
Simulink Object to a DOORS Requirement” on page 3-10 . The following procedure describes how to navigate from a requirement to its linked Simulink object.

To navigate the two-way link from the Transmission Requirements object in the DOORS database to the transmission subsystem in the sf_car_doors model:

1 In the My Requirements module window, in the left pane, select the [Simulink reference: sf_car_doors/tranmission (SubSystem)] subnode under the Transmission Requirements node.

2 Select **MATLAB > Select item.**

The sf_car_doors model highlights the transmission subsystem in the Model Editor.



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” on page 4-2
- “Using the Verification Manager” on page 4-7
- “Managing Verification Requirements” on page 4-24

Using Model Verification Blocks

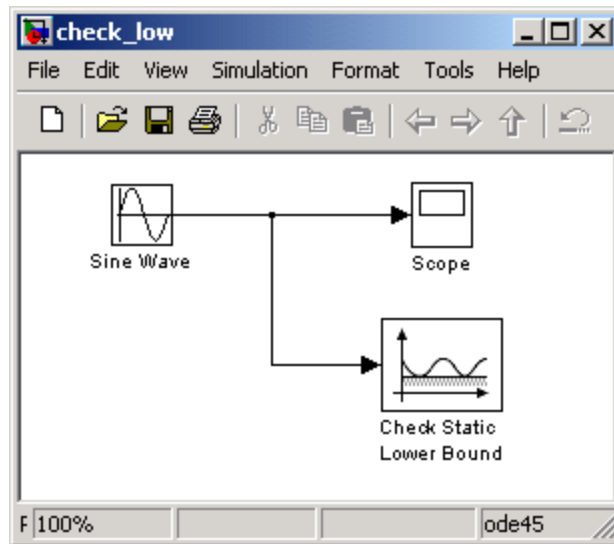
You use Model Verification blocks throughout your model to monitor its signals. You can set a verification block to assert when its signal leaves the specified limit or range. During simulation, when the signal crosses the limit, the verification block can

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

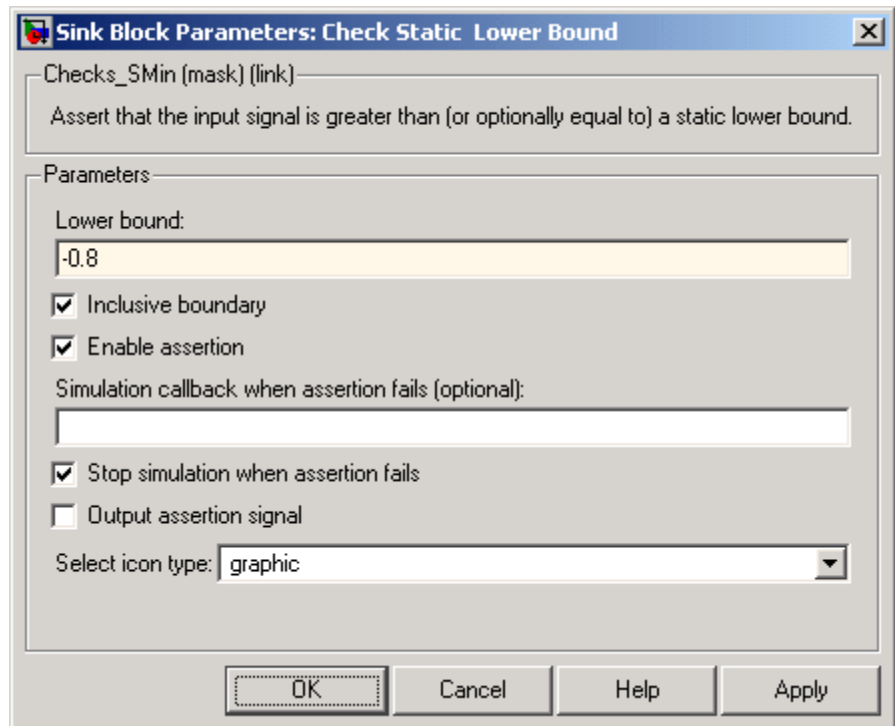
To see a complete list of all Model Verification blocks and references for each, see the “Model Verification” category in the Simulink Block Reference documentation.

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

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



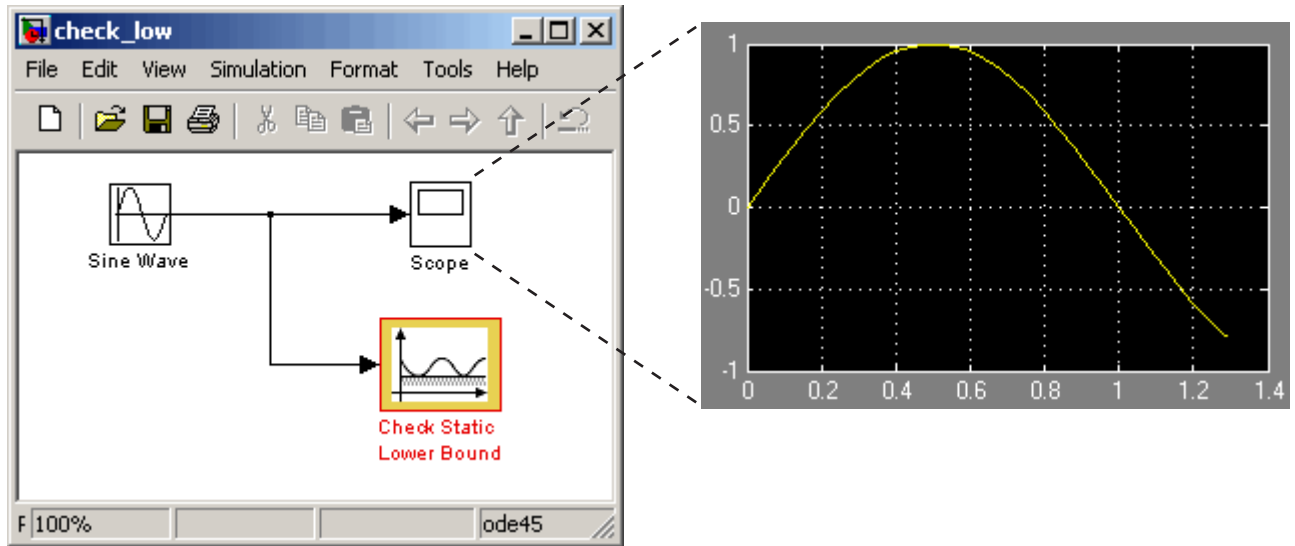
- 2 Set the model to run for 2 seconds while the Sine Wave block outputs a signal with an amplitude of 1 and a frequency of π radians per second.
- 3 Open the Check Static Lower Bound block and set the parameters as follows:



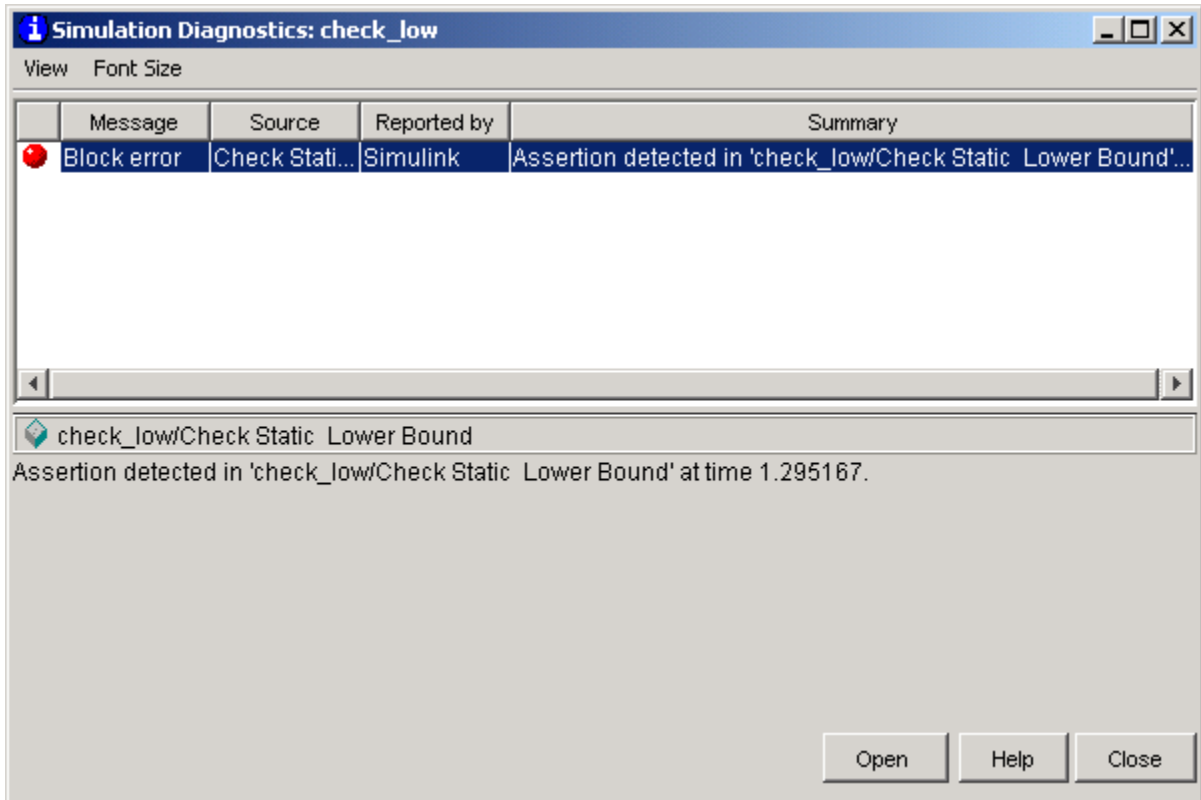
A verification block is enabled for an assertion when the **Enable assertion** check box is selected (this is the default setting). According to the preceding property settings, the Check Static Lower Bound block is set to detect a signal value of -0.8 or lower. If this signal is detected, simulation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

- 4 Run the simulation.

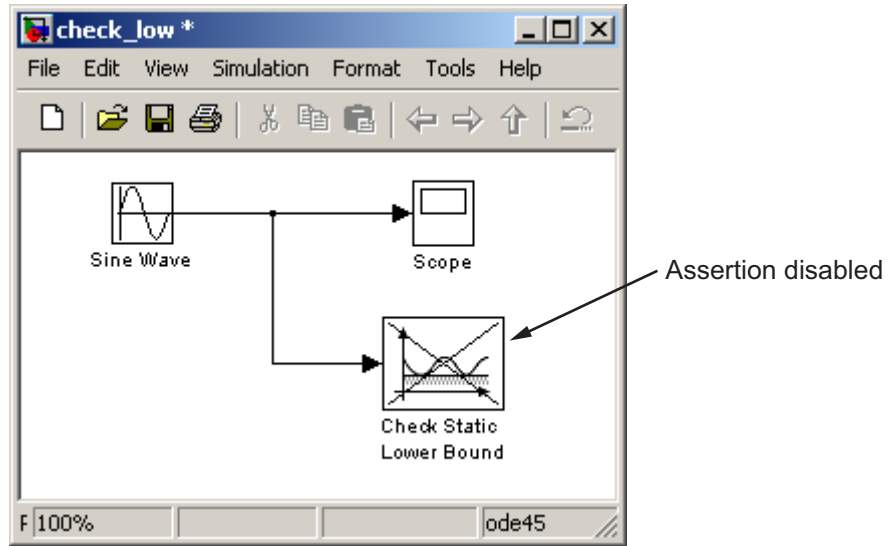
The model stops simulating after 1.295 seconds, when the output is -0.8, as shown. This brings focus to the asserting verification block, which is highlighted.



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

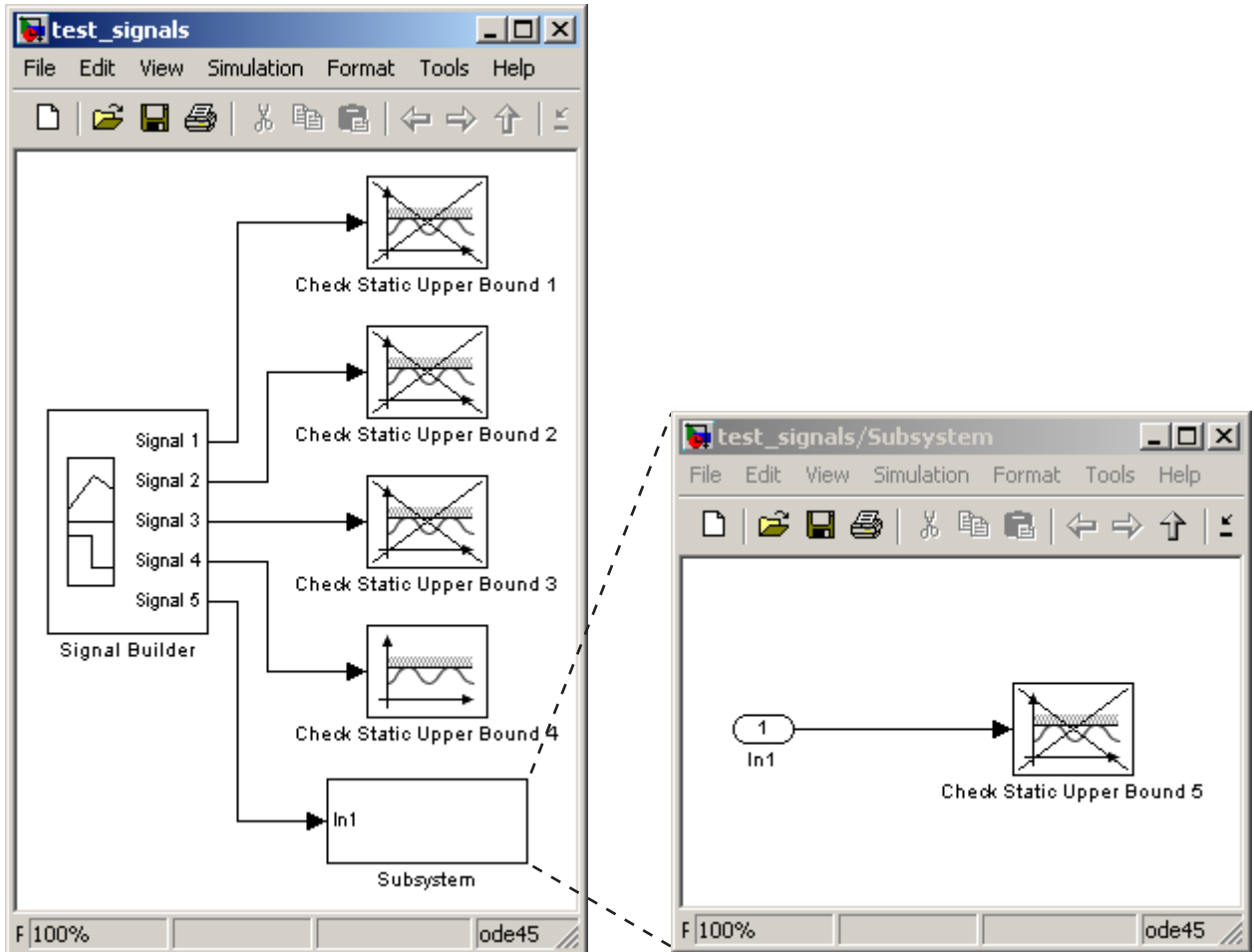
What Is the Verification Manager?

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 Simulink model that you use to examine the Verification Manager in the following steps:

- 1 Create the following example model in the Simulink software.

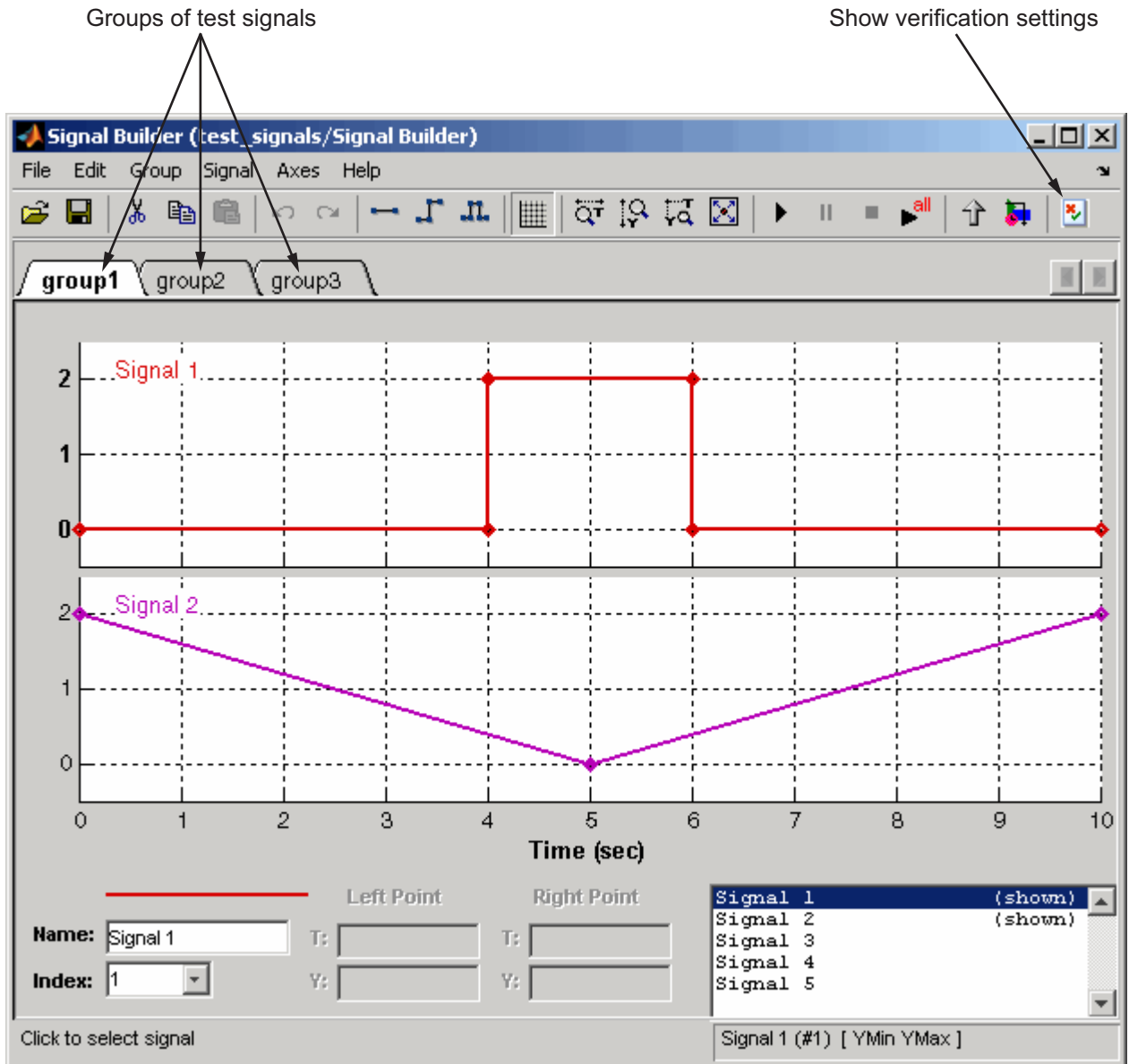


Typically, a Signal Builder block provides test signals for an entire model from one location. The example model contains a Signal Builder block feeding five test signals to Model Verification blocks. Signals 1 through 4 are sent directly to Check Static Upper Bound Model Verification blocks. The fifth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

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


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

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



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

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

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

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

The screenshot displays 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 editing tools, and a workspace divided into three groups: group1, group2, and group3.

The main workspace contains two signal plots over a 10-second time interval:

- Signal 1 (Red):** A step function that is 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 (Purple):** A triangular wave that starts at 2 at 0 seconds, decreases linearly to 0 at 5 seconds, and then increases linearly back to 2 at 10 seconds.


At the bottom of the workspace, there are input fields for "Name:", "Index:", "Left Point", "Right Point", "T:", "Y:", and "Click to select signal".

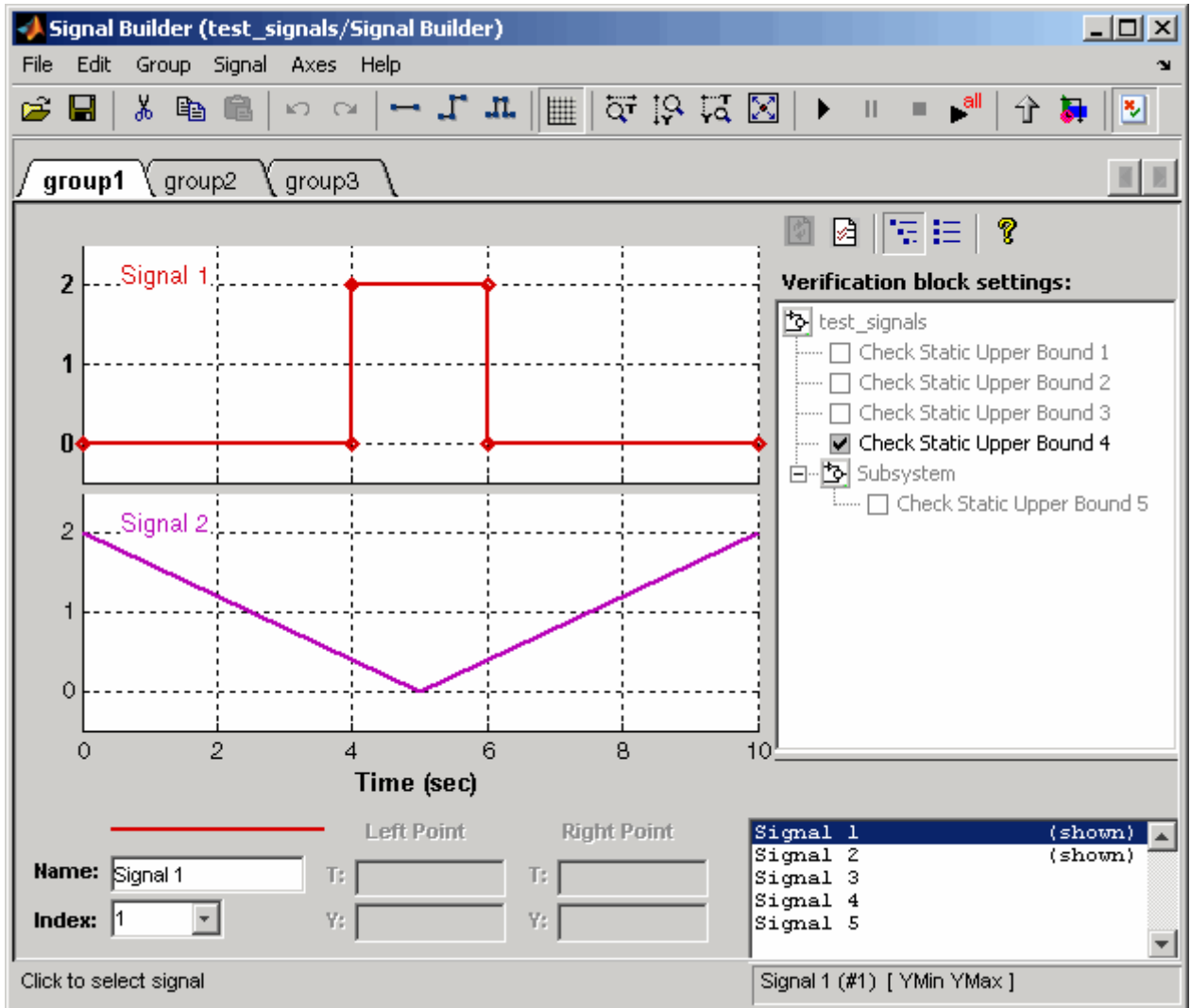
On the right side, there are two panels:

- Verification pane:** Labeled "Verification pane" with an arrow pointing to the "Verification block settings:" section. This section shows a tree view for "test_signals" with the following options:
 - 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 pane:** Labeled "Requirements pane" with an arrow pointing to the "Requirements:" section. This section displays the text: "No requirements in this group".


At the bottom right, there is a list of signals: Signal 1 (shown), Signal 2 (shown), Signal 3, Signal 4, and Signal 5.

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.

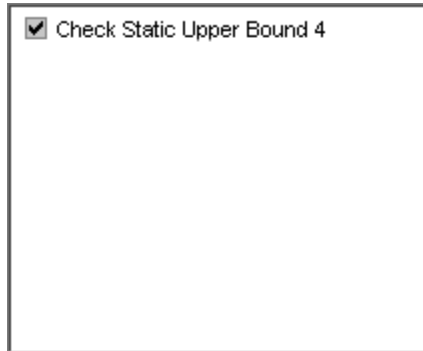
- 4** Just above the **Verification block settings** pane, select  to close the **Requirements** pane.




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.

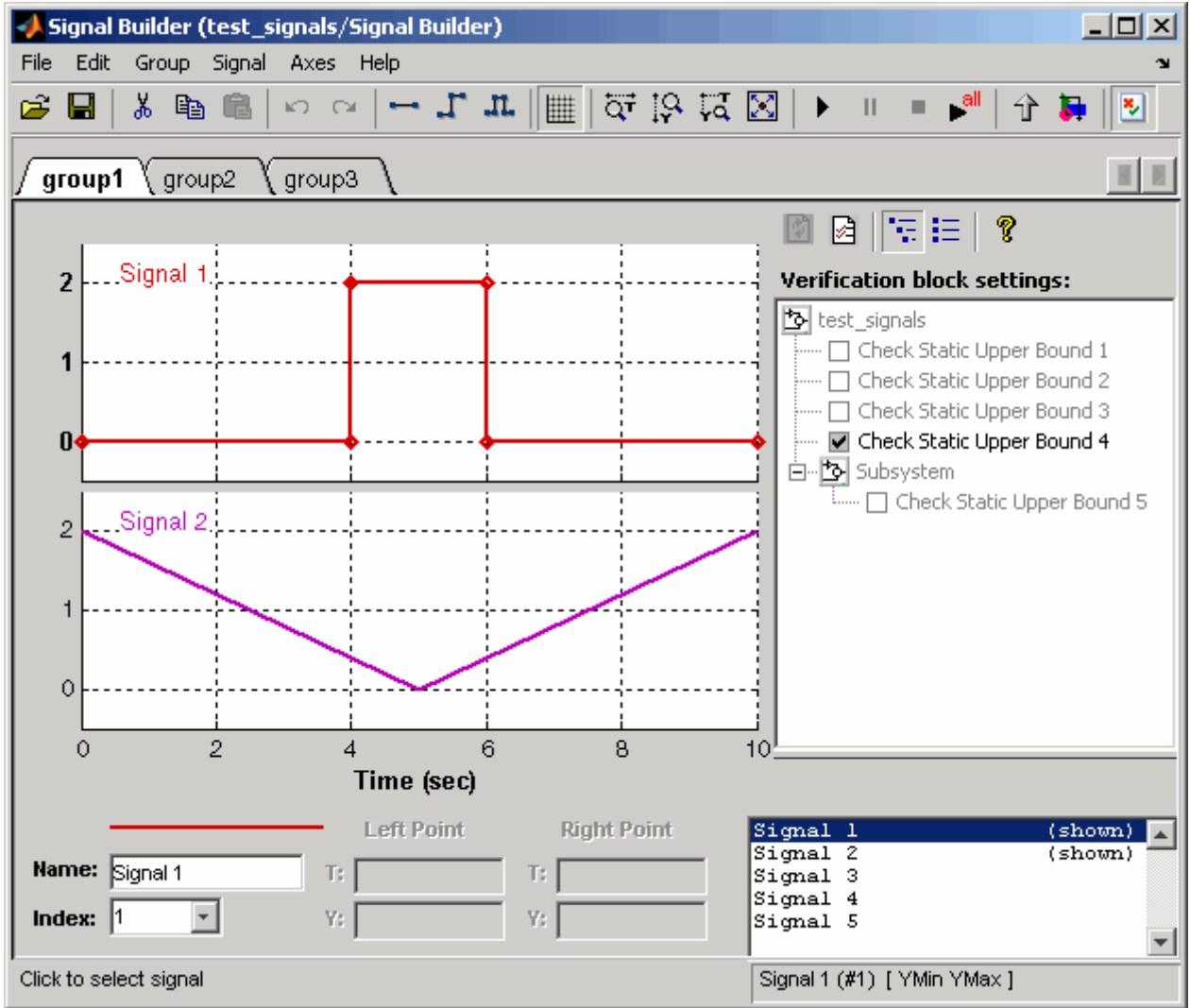
The **Verification block settings** pane now shows only the enabled Model Verification blocks for the current group, as shown.



- 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.



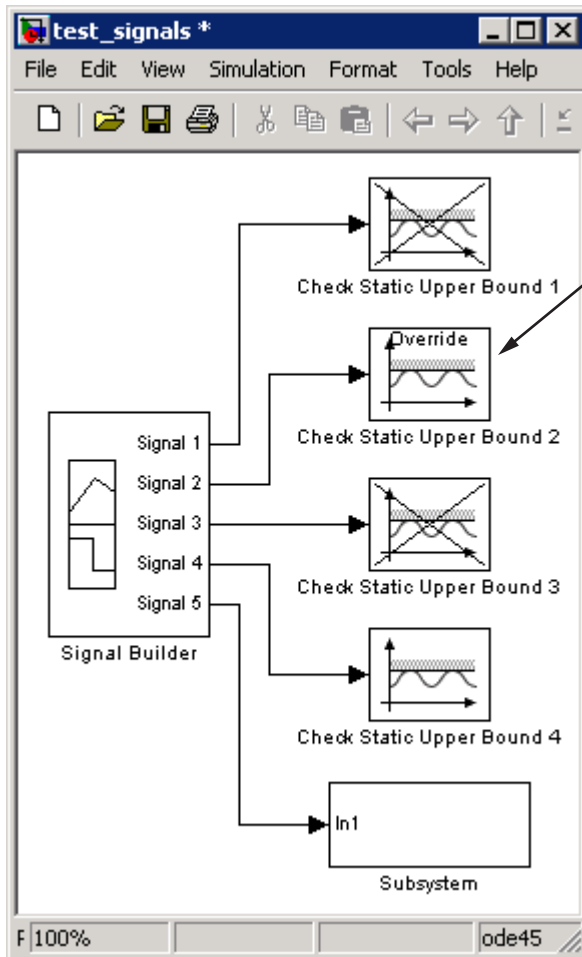
The **Verification block settings** pane in the preceding example lists the Model Verification blocks in the model. Each verification block has a preceding status node that indicates whether its assertion is enabled or disabled and whether that setting applies universally or to the active group. The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

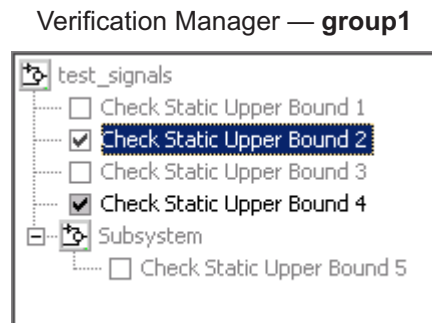
Use the Verification Manager to enable or disable model verification blocks in the `test_signals` model you created in “Opening the Verification Manager” on page 4-7, as follows:

- 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.

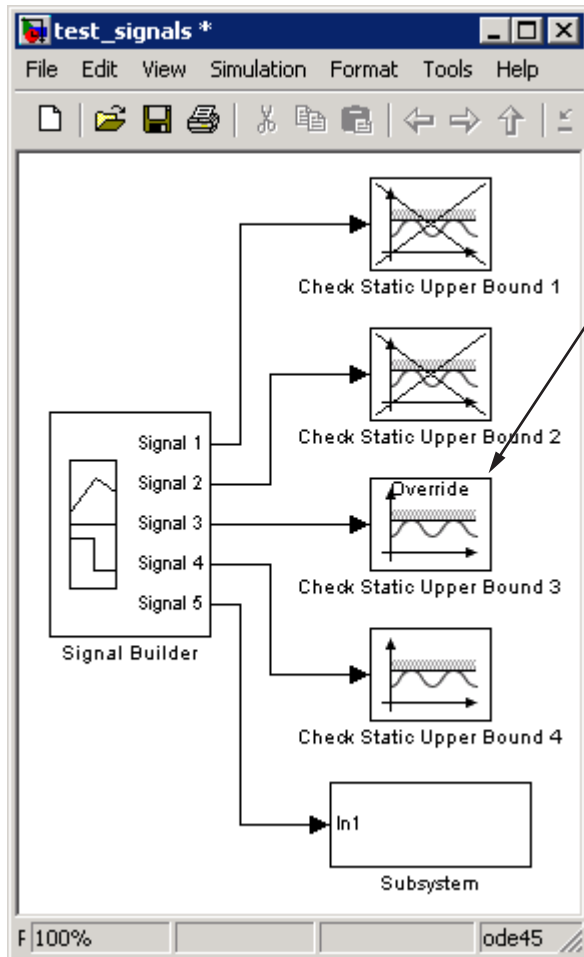


Disabled but enabled in current group (**group1**)

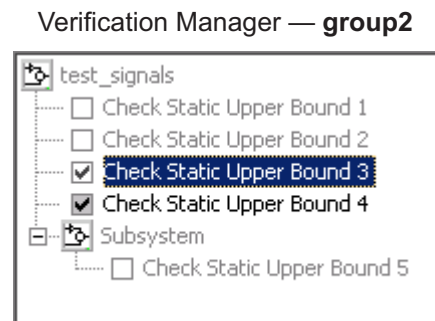


Because it is enabled in the current group, the Check Static Upper Bound 2 block gains an **Override** label and loses its cross-out. The meaning behind the change in appearance becomes clearer when another group is selected.

- 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**)




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

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

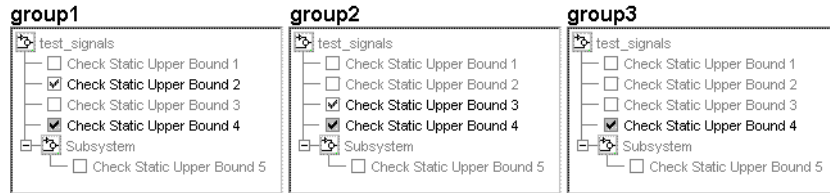
Assertion Block	Description
	Enabled for all groups
	Disabled in current group
	Enabled in current group

Using Enabling and Disabling Tools in the Verification Manager

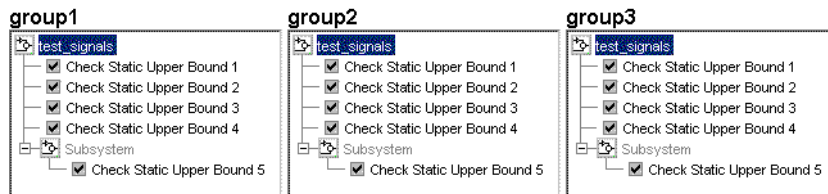
If you have many verification blocks, it is tedious to enable and disable blocks individually. For this reason, the Verification Manager lets you enable and disable blocks through selections from a context menu. These selections vary with the node as follows:

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

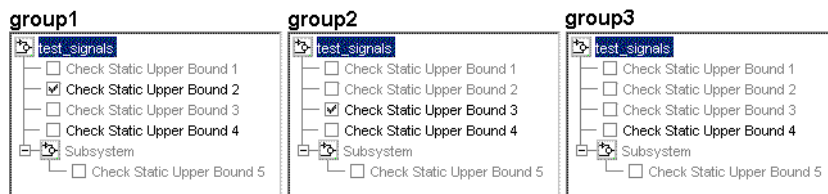


- 1 Right-click the `test_signals` node and select **Contents enable for all groups**.



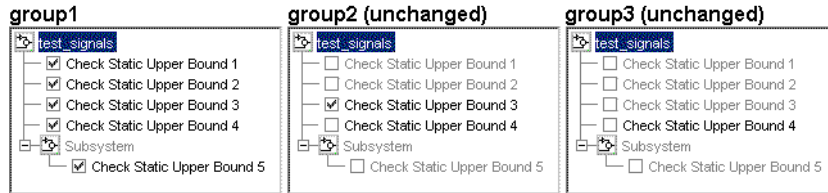
Applying the **Contents enable for all groups** selection to the model node enables all contained Model Verification blocks, for all test groups, in all contained subsystems.

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



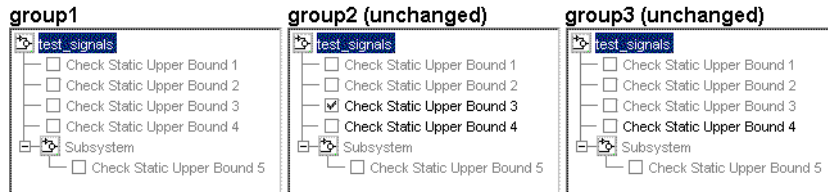
Applying the **Contents enable by group** selection to the model node restores the previous individually enabled/disabled settings for each block in each group.

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



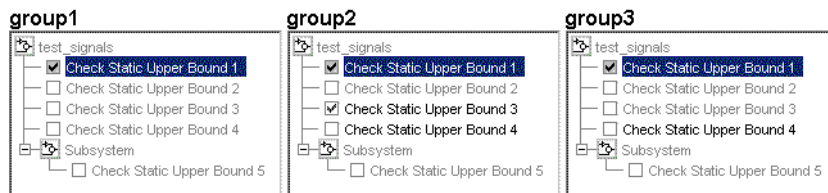
Applying **Contents group enable** to the `test_signals` model node in **group1** individually enables all contained blocks for **group1**, but leaves the other groups untouched.

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



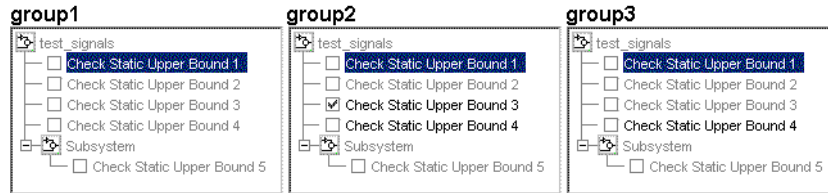
Applying **Contents group disable** to the `test_signals` model node in **group1** individually disables all contained blocks for **group1**, but leaves the other groups untouched.

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



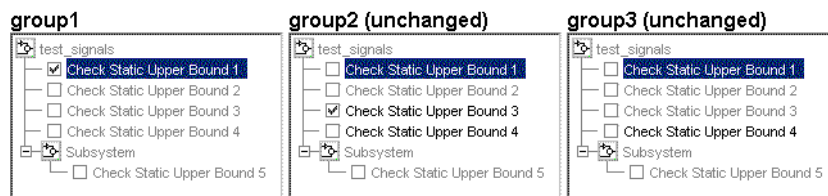
Applying **Block enable for all groups** to the individual **group1** block node for `Check Static Upper Bound 1` in **group1** enables this block for all groups.

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



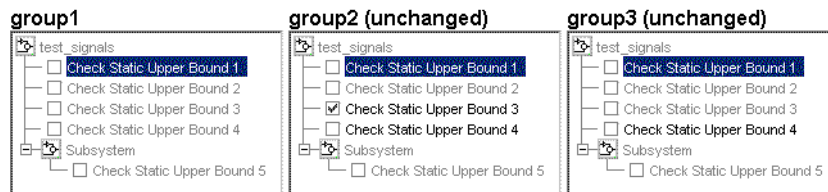
Applying **Block enable by group** to the individual **group1** block node for Check Static Upper Bound 1 in **group1** restores the previous individually enabled/disabled state to this block for all groups. This lets you enable or disable this node individually for each group.

- 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**.



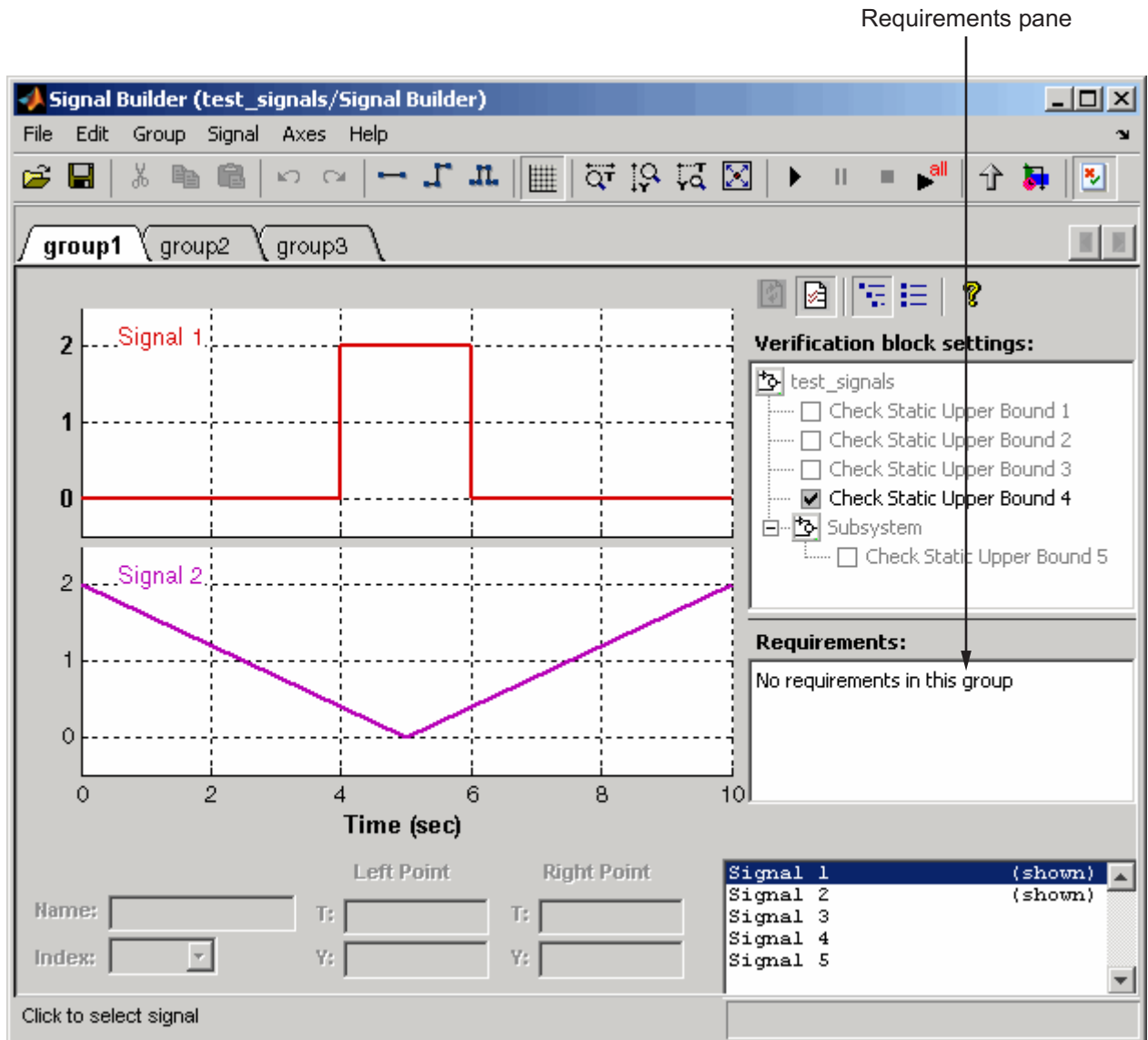
Applying **Block group disable** to the individual block node for Check Static Upper Bound 1 in **group1** disables this block for this group only. This is equivalent to clearing the check box for this node.

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 for details on linking requirements documents to individual Simulink blocks.

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

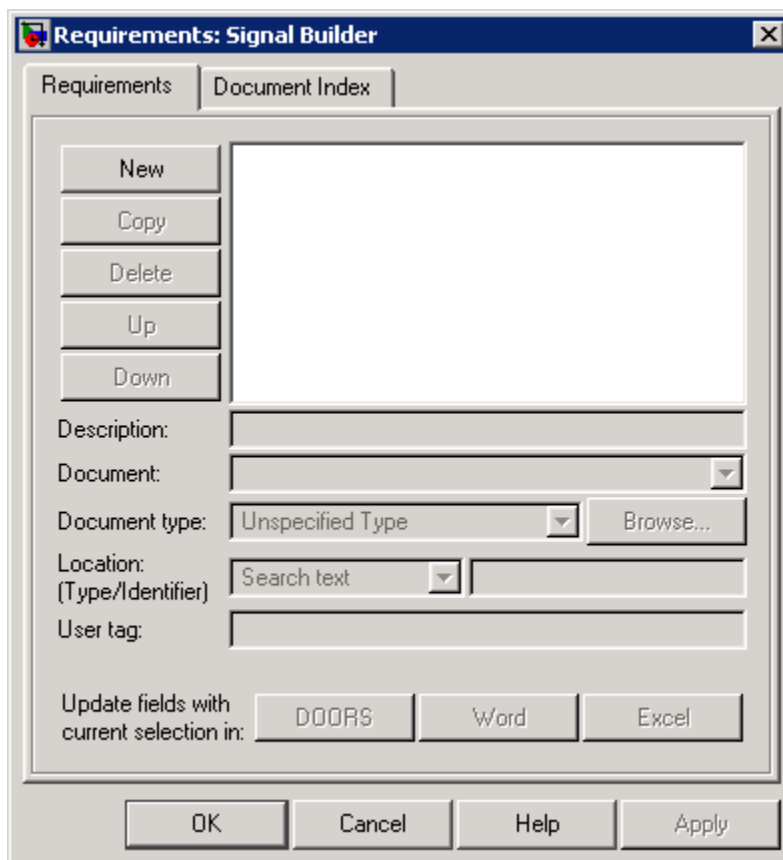


1 Right-click anywhere in the **Requirements** pane.

A pop-up menu appears.

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

The Requirements dialog box appears, as shown.



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

- Add links to requirements documents as described in steps 4 through 9 of .

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

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, decreases linearly to 0 at 5 seconds, and then increases linearly back to 2 at 10 seconds.
- Verification block settings:** A tree view on the right 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 this list to the "Check Static Upper Bound 4" checkbox in the settings panel.
- 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:", "Left Point Y:", "Right Point T:", and "Right Point Y:" are visible at the bottom.

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

- “Introduction to Model Coverage” on page 5-2
- “Analyzing Model Coverage” on page 5-11
- “Model Coverage Reporting Options” on page 5-16
- “Understanding Model Coverage Reports” on page 5-30
- “Colored Simulink Diagram Coverage Display” on page 5-69
- “Using Model Coverage Commands” on page 5-74
- “Using Model Coverage Commands for Referenced Models” on page 5-81
- “Model Coverage for Embedded MATLAB Function Blocks” on page 5-87

Introduction to Model Coverage

In this section...
“What Is Model Coverage?” on page 5-2
“How Model Coverage Works” on page 5-2
“Simulink Optimizations and Model Coverage” on page 5-2
“Types of Model Coverage” on page 5-3
“Blocks That Receive Model Coverage” on page 5-8

What Is Model Coverage?

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

How Model Coverage Works

Model coverage analyzes the execution of blocks that directly or indirectly determine simulation pathways through your model. It can also analyze the states and transitions of Stateflow charts in a model. 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.

Review the types of coverages that model coverage performs in “Types of Model Coverage” on page 5-3. Then, for an example of a model coverage report, see “Understanding Model Coverage Reports” on page 5-30.

Simulink Optimizations and Model Coverage

Two Simulink optimization parameters, in the Configuration Parameters dialog box **Optimization** pane, can affect your model coverage data:

- “Block reduction” on page 5-3

- “Conditional input branch execution” on page 5-3

Block reduction

To achieve faster execution during model simulation and in generated code, enable the **Block reduction** parameter on the Configuration Parameters dialog box **Optimization** pane. The Simulink software collapses certain groups of blocks into a single, more efficient block, or removes them entirely.

One of the model coverage options, **Force block reduction off**, allows you to ignore the **Block reduction** parameter when collecting model coverage.

If you do not enable the **Block reduction** parameter, or if you select **Force block reduction off**, the Simulink Verification and Validation software provides coverage data for every block in the model that collects coverage.

If you enable the **Block reduction** parameter and do not set **Force block reduction off**, the coverage report lists the reduced blocks that would have collected coverage.

Conditional input branch execution

To improve model execution when the model contains Switch and Multiport Switch blocks, enable **Conditional input branch execution**. If you select this parameter, the simulation executes only blocks that are required to compute the control input and the data input selected by the control input.

This optimization is limited to certain kinds of Switch blocks, as described in “Optimizing Code for Switch Blocks” in the *Real-Time Workshop User’s Guide*.

For example, if your model has a Switch block with output flagged as a test point, such as when a Scope block is attached, that Switch block is not executed, and the model coverage data is incomplete. If you have a model with Switch blocks and you want to ensure that the model coverage data is complete, disable **Conditional input branch execution**.

Types of Model Coverage

Simulink Verification and Validation software can perform several types of coverage analysis:

- “Cyclomatic Complexity” on page 5-4
- “Decision Coverage (DC)” on page 5-4
- “Condition Coverage (CC)” on page 5-5
- “Modified Condition/Decision Coverage (MCDC)” on page 5-5
- “Lookup Table Coverage” on page 5-6
- “Signal Range Coverage” on page 5-7
- “Signal Size Coverage” on page 5-7
- “Simulink Design Verifier Coverage” on page 5-7

Cyclomatic Complexity

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

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

$$c = \sum_1^N (o_n - 1)$$

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

For an example of cyclomatic complexity data in a model coverage report, see “Cyclomatic Complexity” on page 5-38.

Decision Coverage (DC)

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

For an example of decision coverage data in a model coverage report, see “Decisions Analyzed” on page 5-39.

Condition Coverage (CC)

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

When you collect coverage for a model, you may not be able to achieve 100% condition coverage. For example, if you specify to short-circuit Logic blocks, you might not be able to achieve 100% condition coverage for that block. See “Treat Simulink Logic blocks as short-circuited” on page 5-28 for more information.

For an example of condition coverage data in a model coverage report, see “Conditions Analyzed” on page 5-41.

Modified Condition/Decision Coverage (MCDC)

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

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

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

Some Simulink blocks support MCDC coverage, some blocks support only condition coverage, and some blocks support only decision coverage. The table “Blocks That Receive Model Coverage” on page 5-8 lists which blocks receive which types of model coverage. For example, the Combinatorial Logic block can receive decision coverage and condition coverage, but not MCDC coverage.

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

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

When you collect coverage for a model, you may not be able to achieve 100% MCDC coverage. For example, if you specify to short-circuit Logic blocks, you may not be able to achieve 100% MCDC coverage for that block. See “Treat Simulink Logic blocks as short-circuited” on page 5-28 for more information.

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

For an example of MCDC coverage data in a model coverage report, see “MCDC Analysis” on page 5-41. For an example of accumulated coverage results, see “Cumulative Coverage ” on page 5-43.

For more information about the DO-178B standard, see “DO-178B Checks” on page 14-4.

Lookup Table Coverage

Lookup table coverage (LUT) examines blocks, such as the Lookup Table block, that output information from inputs in a table of inputs and outputs, interpolating between or extrapolating from table entries. Lookup table coverage records the frequency that table lookups use each interpolation interval. A test case achieves full coverage when it executes each interpolation and extrapolation interval at least once. For each lookup table block in

the model, the coverage report displays a colored map of the lookup table, indicating where each interpolation was performed.

For an example of lookup table coverage data in a model coverage report, see “N-Dimensional Lookup Table” on page 5-45.

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

Signal Range Coverage

Signal range coverage records the minimum and maximum signal values at each block in the model, as measured during simulation. Only blocks with output signals receive signal range coverage.

For an example of signal range coverage data in a model coverage report, see “Signal Range Analysis” on page 5-54.

Signal Size Coverage

Signal size coverage records the minimum, maximum, and allocated size for all variable-size signals in a model. Only blocks with variable-size output signals are included in the report.

For an example of signal size coverage data in a model coverage report, see “Signal Size Coverage for Variable-Dimension Signals” on page 5-56.

For more information about variable-size signals, see “Working with Variable-Size Signals”.

Simulink Design Verifier Coverage

The Simulink Verification and Validation software collects model coverage data for the following Simulink® Design Verifier™ blocks:

- Test Condition
- Test Objective

- Proof Assumption
- Proof Objective

If you do not have a Simulink Design Verifier license, you can collect model coverage for a model that contains these blocks, but you cannot analyze the model using the software.

By adding one or more of those Simulink Design Verifier blocks into your model, you can:

- Check the results of a Simulink Design Verifier analysis by running generated test cases, and use the blocks to observe the results.
- Define model requirements using the Test Objective block and verify the results with model coverage data that the software collected during simulation.
- Analyze the model, create a test harness, and then simulate the harness with the Test Objective block to collect model coverage data.
- Analyze the model and use the Proof Assumption block to verify any counterexamples that the Simulink Design Verifier identifies.

For an example of coverage data for Simulink Design Verifier blocks in a model coverage report, see “Simulink® Design Verifier Block Coverage” on page 5-58.

Blocks That Receive Model Coverage

Certain Simulink blocks can receive any type of model coverage. Other blocks can only receive certain types of coverage, as the following table shows.

For Stateflow states, events, and state temporal logic decisions, model coverage provides only decision coverage. For Stateflow transitions, model coverage provides decision, condition, and MCDC coverage. For more information, see “Understanding Model Coverage for Stateflow Charts” in the Stateflow documentation.

Block	Decision	Condition	MCDC	LUT	Simulink Design Verifier
Abs	•				
Combinatorial Logic	•	•			
Direct Lookup Table (n-D)				•	
Discrete-Time Integrator (when saturation limits are enabled)	•				
Embedded MATLAB Function	•	•	•		
Enabled and Triggered Subsystem	•	•	•		
Enabled Subsystem	•				
Fcn (Boolean operators only)		•			
For Iterator, For Iterator Subsystem	•				
If, If Action Subsystem	•				
Interpolation Using Prelookup				•	
Logical Operator		•	•		
Lookup Table				•	
Lookup Table (2-D)				•	
Lookup Table (n-D)				•	
MinMax	•				
Model	•	•	•	•	
Multiport Switch	•				
Proof Assumption					•
Proof Objective					•

Block	Decision	Condition	MCDC	LUT	Simulink Design Verifier
Rate Limiter	• (Relative to slew rates)				
Relay	•				
Saturation	•				
Stateflow charts	•	•	•		
Switch	•				
Switch Case, Switch Case Action Subsystem	•				
Test Condition					•
Test Objective					•
Triggered Subsystem	•	•	•		
While Iterator, While Iterator Subsystem	•				

Analyzing Model Coverage

In this section...
“Model Coverage Analysis Workflow” on page 5-11
“Creating and Running Test Cases” on page 5-11

Model Coverage Analysis Workflow

To develop effective tests with model coverage:

- 1 Develop one or more test cases for your model. (See “Creating and Running Test Cases” on page 5-11.)
- 2 Run the test cases to verify that the model behavior is correct.
- 3 Analyze the coverage reports produced by the Simulink Verification and Validation software.
- 4 Using the information in the coverage reports, modify the test cases to increase their coverage or add new test cases to cover areas not currently covered.
- 5 Repeat the preceding steps until you are satisfied with the coverage of your test set.

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

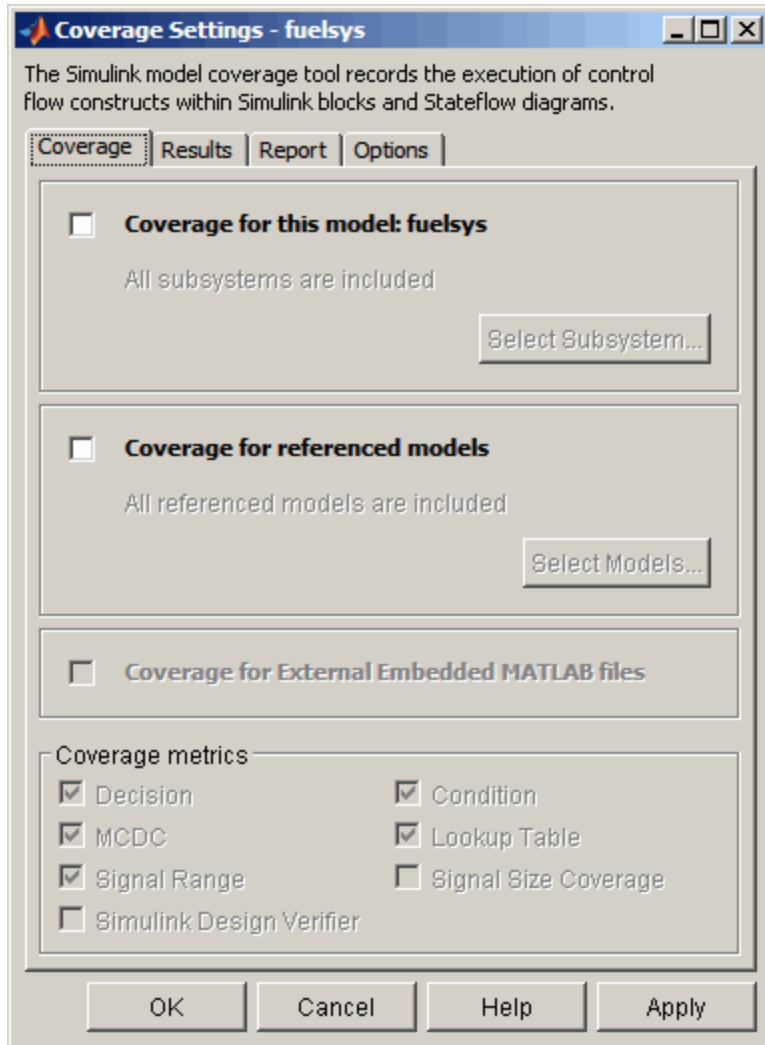
Creating and Running Test Cases

To create and run test cases, model coverage provides two MATLAB commands, `cvtest` and `cvsim`. The `cvtest` command creates test cases that the `cvsim` command runs. (See “Creating Tests with `cvtest`” on page 5-74 and “Running Tests with `cvsim`” on page 5-76.)

You can also run the coverage tool interactively:

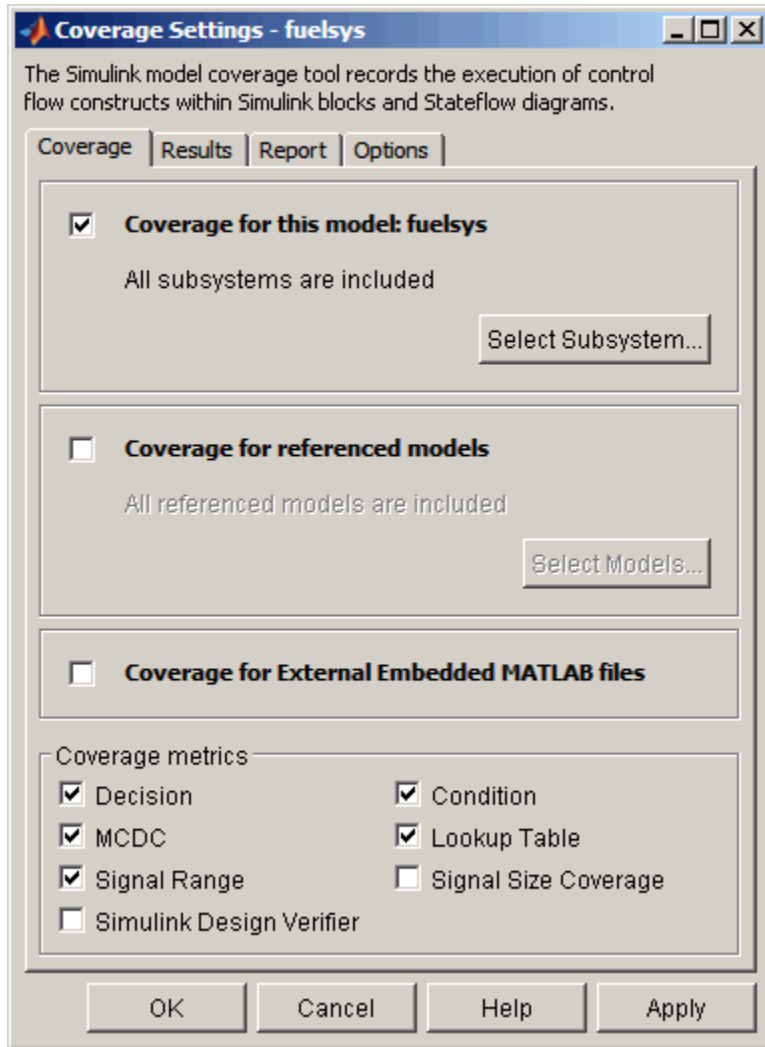
- 1 Open the fuelsys model.
- 2 In the Simulink model window, select **Tools > Coverage Settings**.

The Coverage Settings dialog box **Coverage** tab appears.



3 Select **Coverage for this model**, which enables:

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



- 4 Under **Coverage metrics**, select the types of coverage that you want to record in the coverage report.

For a complete description of all coverage options in the Coverage Settings dialog box, see “Model Coverage Reporting Options” on page 5-16.

5 Click **OK**.

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

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

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

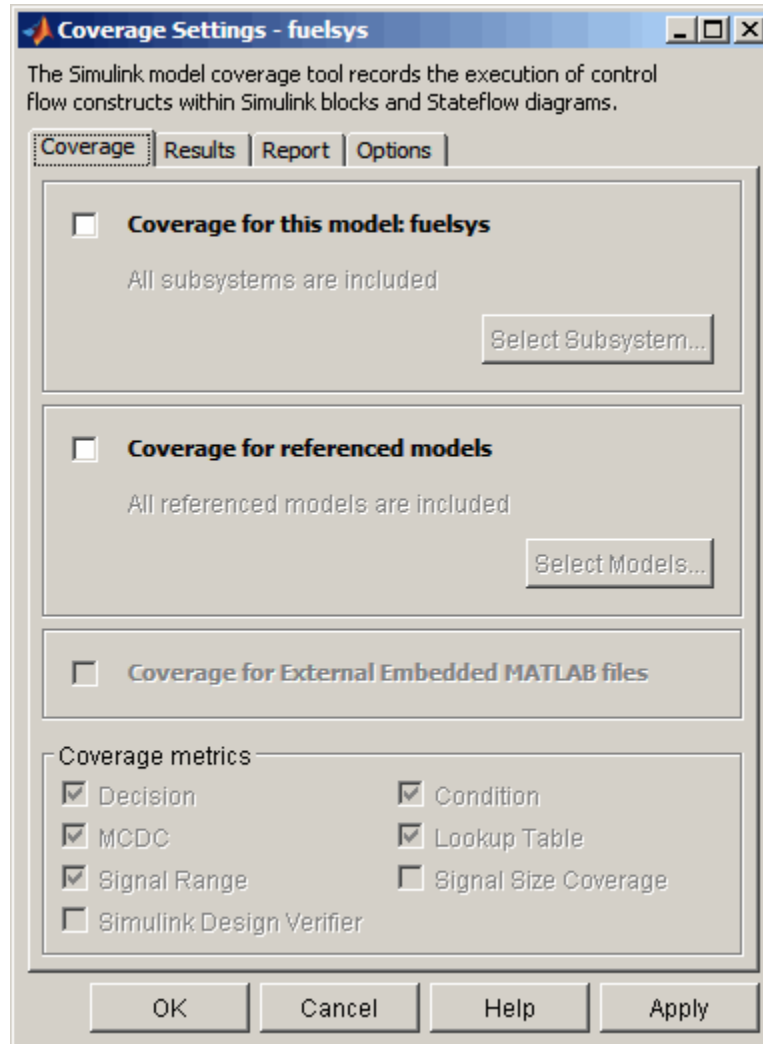
You cannot select both block reduction and conditional branch input optimization when you perform coverage analysis because they interfere with coverage recording.

Model Coverage Reporting Options

In this section...
“Coverage Settings Dialog Box” on page 5-16
“Coverage Tab” on page 5-18
“Results Tab” on page 5-21
“Report Tab” on page 5-23
“Options Tab” on page 5-27

Coverage Settings Dialog Box

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



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

Coverage Tab

On the **Coverage** tab, select the model coverages calculated during simulation.

Coverage for this model

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

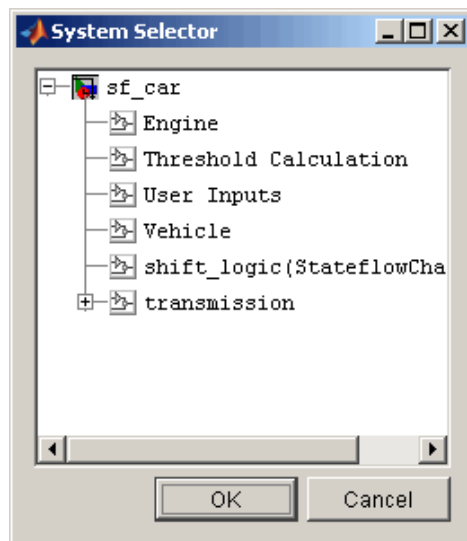
Select Subsystem

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

To restrict coverage reporting to a particular subsystem:

- 1 On the **Coverage** tab, click **Select Subsystem**.

The System Selector dialog box appears.



- 2 In the System Selector dialog box, select the subsystem for which you want to enable coverage reporting and click **OK**.

Coverage for referenced models

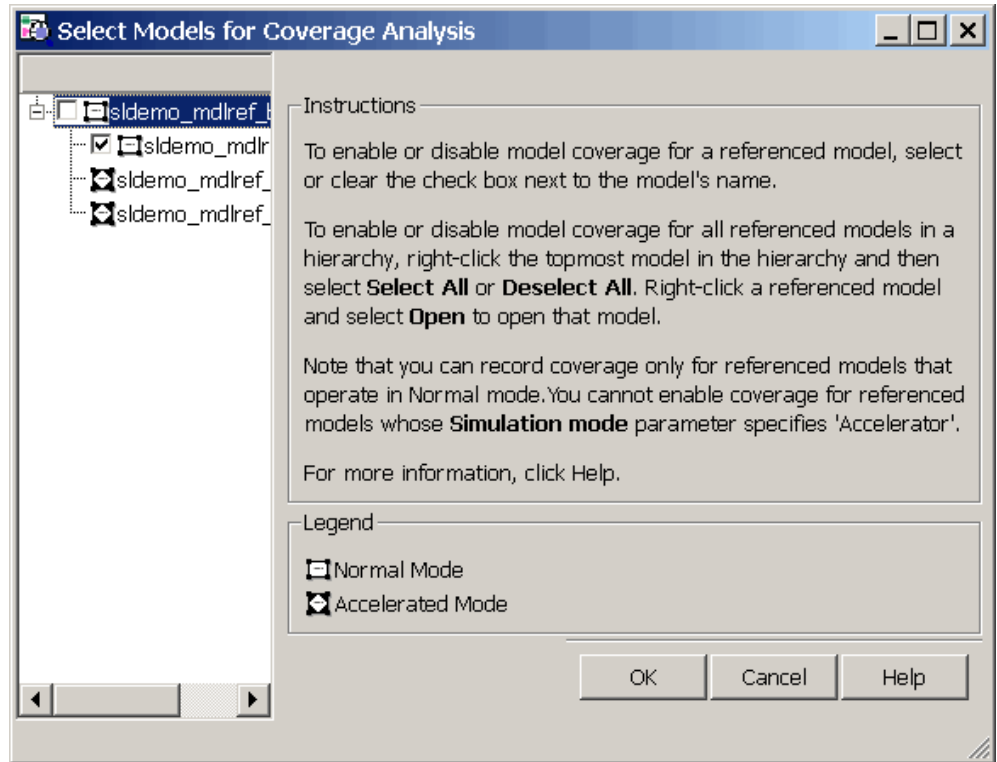
Causes the software to gather and report the model coverages that you specify for referenced models during simulation. When you select the **Coverage for referenced models** option, the **Select Models** button and the **Coverage metrics** section of the **Coverage** tab become available.

Select Models

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

To enable coverage reporting for particular referenced models:

- 1 On the **Coverage** pane, click **Select Models**.



- 2 In the Select Models for Coverage Analysis dialog box, select the referenced models for which you want coverage reporting and then click **OK**.

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

Coverage for External Embedded MATLAB Files

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

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

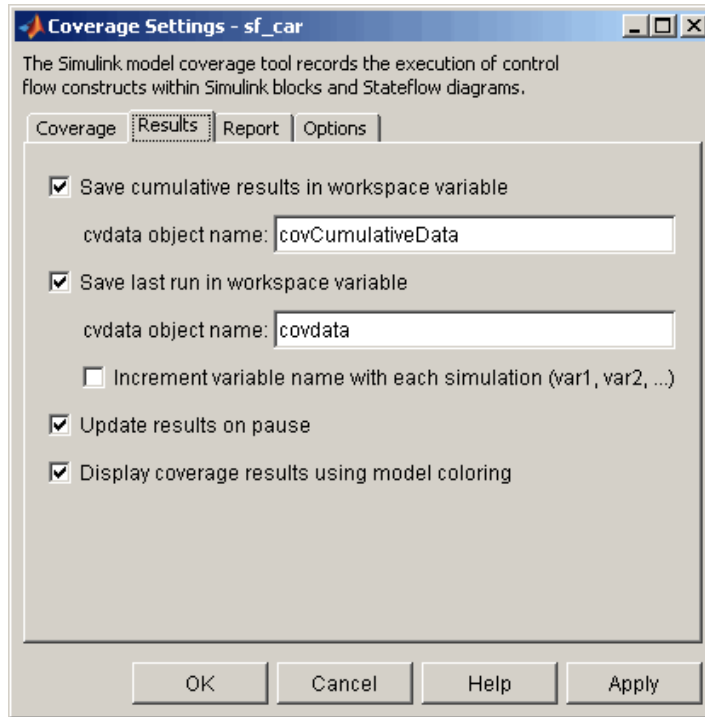
Coverage metrics

Select the types of test case coverage analysis that you want the tool to perform (see “Types of Model Coverage” on page 5-3). The Simulink Verification and Validation software gathers and reports those types of coverage for the subsystem, model, and referenced models .

Note To specify different types of coverage analysis for each of the referenced models in a hierarchy, use the `cv.cvtestgroup` and `cvsimref` functions. For more information, see “Using Model Coverage Commands for Referenced Models” on page 5-81.

Results Tab

On the **Results** pane, select the destination for model coverage results.



Save Cumulative Results in Workspace Variable

Causes model coverage to accumulate and save the results of successive simulations in the workspace variable that you specify in the **cvdata object name** field.

Save Last Run in Workspace Variable

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

Increment Variable Name with Each Simulation

Causes the Simulink Verification and Validation software to increment the name of the coverage data object variable used to save the last run with each simulation, so that the current simulation run does not overwrite the results of the previous run.

Update Results on Pause

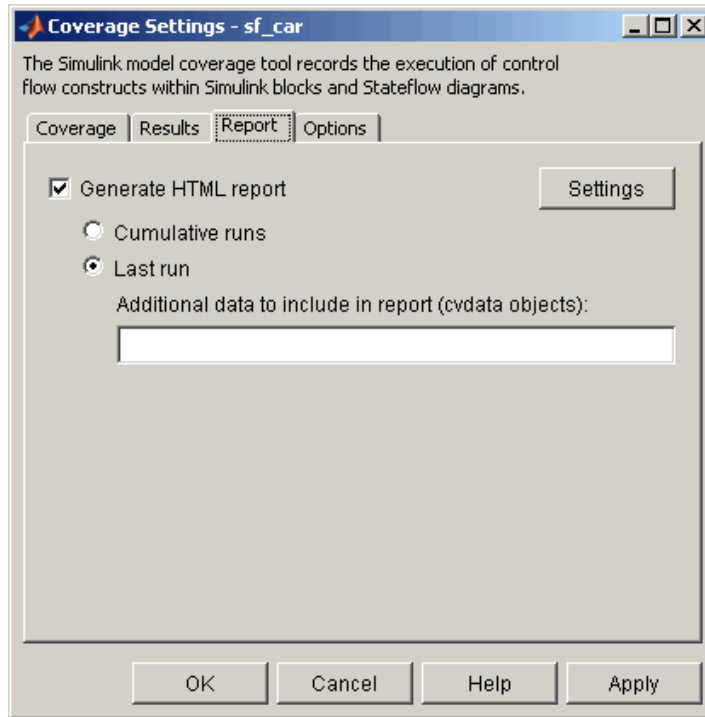
Causes the model coverage results to be recorded up to the point at which you pause the simulation for the first time. When you resume simulation and later pause or stop, the model coverage report reappears, with coverage results up to the current pause or stop time.

Display Coverage Results Using Model Coloring

Causes coloring of Simulink blocks according to their level of model coverage, after simulation. Blocks highlighted in light green received full coverage during testing. Blocks highlighted in light red received incomplete coverage. See “Colored Simulink Diagram Coverage Display” on page 5-69.

Report Tab

On the **Report** tab, specify whether the model coverage tool should generate an HTML report and what data the report should include.

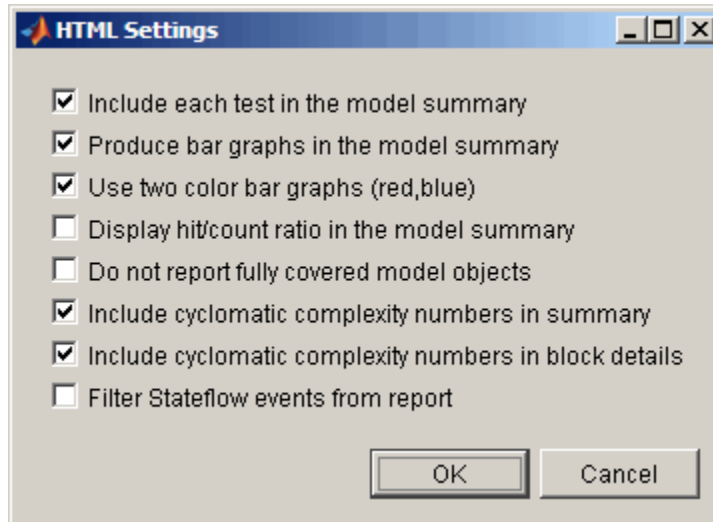


Generate HTML Report

Causes the Simulink Verification and Validation software to create an HTML report containing the coverage data. At the end of the simulation, the report appears in the MATLAB Web browser. Click the **Settings** button to select various reporting options.

Settings

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



Option	Description
Include each test in the model summary	The model hierarchy table at the top of the HTML report includes columns listing the coverage metrics for each test. If you do not select this option, the model summary reports only the total coverage.
Produce bar graphs in the model summary	Causes the model summary to include a bar graph for each coverage result for a visual representation of the coverage.
Use two color bar graphs (red, blue)	Red and blue bar graphs appear in the report instead of black and white.
Display hit/count ratio in the model summary	Reports coverage numbers as both a percentage and a ratio, for example, 67% (8/12).

Option	Description
Do not report fully covered model objects	The coverage report includes only model objects that the simulation does not cover fully, useful when developing tests, because it reduces the size of the generated reports.
Include cyclomatic complexity numbers in summary	Includes the cyclomatic complexity (see “Types of Model Coverage” on page 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 for Stateflow Chart blocks.
Include cyclomatic complexity numbers in block details	Includes the cyclomatic complexity metric in the block details section of the report.
Filter Stateflow events from report	Excludes coverage data on Stateflow events.

Cumulative Runs

Display the coverage results from successive simulations in the report. For more information, see “Save Cumulative Results in Workspace Variable” on page 5-22.

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

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

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

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

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

You can also calculate cumulative coverage results at the command line through the `+` operator:

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

Last run

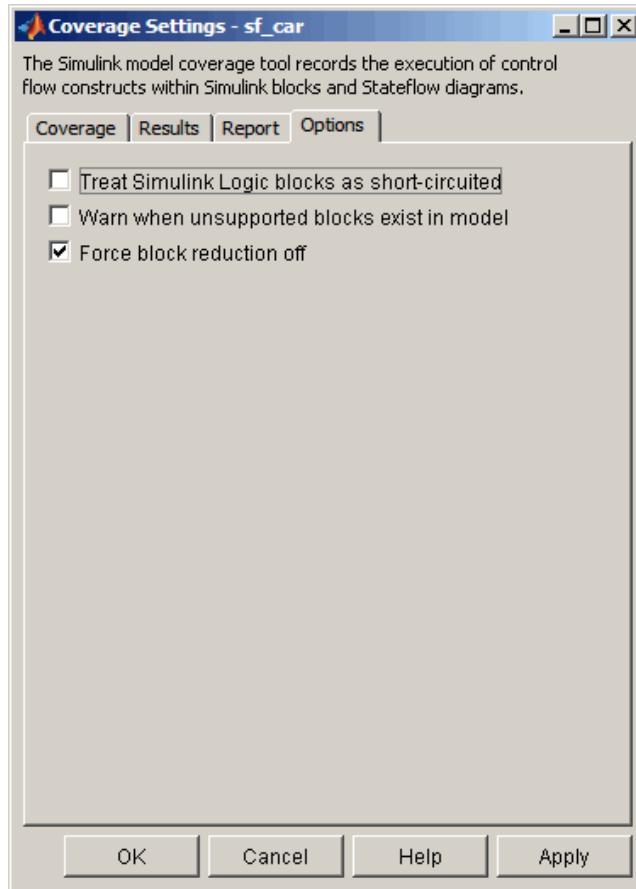
Include only the results of the most recent simulation run in the report.

Additional data to include in report

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

Options Tab

On the **Options** tab, select options for model coverage reports.



Treat Simulink Logic blocks as short-circuited

The **Treat Simulink Logic blocks as short-circuited** option applies only to condition and MCDC coverage. If you select this option, coverage analysis treats Simulink Logic blocks as if the block ignores remaining inputs when the previous inputs alone determine the block's output. For example, if the first input to a Logical Operator block whose **Operator** parameter specifies AND is false, MCDC coverage analysis ignores the values of the other inputs when determining MCDC coverage for a test case.

If you enable this feature and Logic blocks are short-circuited while collecting model coverage, you may not be able to achieve 100% coverage for that block.

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

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

Warn when unsupported blocks exist in model

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

Force block reduction off

To achieve faster execution during model simulation and in generated code, enable the **Block reduction** parameter on the Configuration Parameters dialog box **Optimization** pane. The Simulink software collapses certain groups of blocks into a single, more efficient block, or removes them entirely.

One of the model coverage options, **Force block reduction off**, allows you to ignore the **Block reduction** parameter when collecting model coverage.

If you do not enable the **Block reduction** parameter, or if you select **Force block reduction off**, the Simulink Verification and Validation software provides coverage data for every block in the model that collects coverage.

If you enable the **Block reduction** parameter and do not set **Force block reduction off**, the coverage report lists the reduced blocks that would have collected coverage.

The model coverage report identifies any reduced blocks. For an example of a reduced blocks report, see “Block Reduction” on page 5-52.

Understanding Model Coverage Reports

In this section...
“Types of Coverage Reports” on page 5-30
“Model Coverage Reports” on page 5-31
“Model Summary Reports” on page 5-61
“Model Reference Coverage Reports” on page 5-62
“External M-File Coverage Reports” on page 5-62
“Subsystem Coverage Reports” on page 5-66

Types of Coverage Reports

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

Report Type	Description	HTML Report File Name
“Model Coverage Reports” on page 5-31	Provides coverage information for all model elements, including the model itself.	<i>model_name_cov.html</i>
“Model Summary Reports” on page 5-61	Provides links to coverage results for all referenced models and external M-files in the model hierarchy. Created when the top-level model includes Model blocks or calls one or more external M-files.	<i>model_name_summary_cov.html</i>
“Model Reference Coverage Reports” on page 5-62	Created for each referenced model in the model hierarchy; has the same format as the model coverage report.	<i>reference_model_name_cov.html</i>

Report Type	Description	HTML Report File Name
“External M-File Coverage Reports” on page 5-62	Provides detailed coverage information about any external M-file that the model calls. There is one report for each M-file called.	<i>M-file_name_cov.html</i>
“Subsystem Coverage Reports” on page 5-66	Model coverage report includes only coverage results for the subsystem, if you select one.	<i>model_name_cov.html</i> ; <i>model_name</i> is the name of the top-level model

Model Coverage Reports

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

- “Coverage Summary” on page 5-32
- “Details” on page 5-33
- “Cyclomatic Complexity” on page 5-38
- “Decisions Analyzed” on page 5-39
- “Conditions Analyzed” on page 5-41
- “MCDC Analysis” on page 5-41
- “Cumulative Coverage ” on page 5-43
- “N-Dimensional Lookup Table” on page 5-45
- “Block Reduction” on page 5-52
- “Signal Range Analysis” on page 5-54
- “Signal Size Coverage for Variable-Dimension Signals” on page 5-56
- “Simulink® Design Verifier Block Coverage” on page 5-58

Coverage Summary

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

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

The coverage summary has two subsections:

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

[Summary](#) | [Details](#) | [Signal Ranges](#) | [Help](#)

Coverage Report for fuelsys

Model Information

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

Simulation Optimization Options

Inline Parameters: off
 Block Reduction: forced off
 Conditional Branch Optimization: on

Coverage Options

Logic block short circuiting: off

Tests

Test 1

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

Summary

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

Details

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

- “Model Details” on page 5-34
- “Subsystem Details” on page 5-35
- “Block Details” on page 5-36
- “Chart Details” on page 5-36
- “Embedded MATLAB Function Details” on page 5-37

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

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

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

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

Details:**1. Model "fuelsys"**

Child Systems: [EGO sensor](#), [MAP sensor](#), [engine speed](#), [engine gas dynamics](#), [fuel rate controller](#), [speed sensor](#), [throttle command](#), [throttle sensor](#)

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

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

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

2. Subsystem "EGO sensor"


Parent: [/fuelsys](#)

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

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

Switch block "SwitchControl"

Parent: [fuelsys/EGO sensor](#)

Uncovered Links: 

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

Decisions analyzed:

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

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

2. Subsystem "[Chart2](#)"

Parent: [/mExternalMfile](#)
Child Systems: [Chart2](#)

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

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

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

eM Function "[lib_em2](#)"

Parent: [mExternalMlib/Chart2](#)
Uncovered Links:

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

```

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

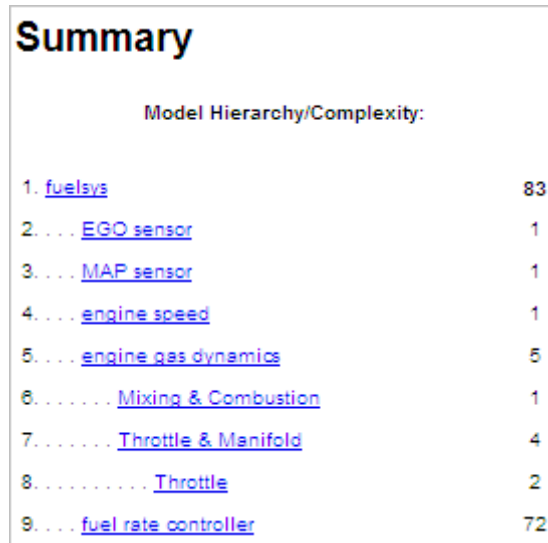
```

For more information about coverage for Embedded MATLAB functions, see “Model Coverage for Embedded MATLAB Function Blocks” on page 5-87.

Cyclomatic Complexity

You can specify that the model coverage report include cyclomatic complexity numbers in two locations in the report:

- The Summary section contains the cyclomatic complexity numbers for each object in the model hierarchy. For a subsystem or Stateflow chart, that number includes the cyclomatic complexity numbers for all their descendants.



Model Hierarchy/Complexity:	
1. fuelsys	83
2. . . . EGO sensor	1
3. . . . MAP sensor	1
4. . . . engine speed	1
5. . . . engine gas dynamics	5
6. Mixing & Combustion	1
7. Throttle & Manifold	4
8. Throttle	2
9. . . . fuel rate controller	72

- The Details sections for each object list the cyclomatic complexity numbers for all individual objects.

7. Subsystem "Throttle & Manifold"

Parent: [fuelsys/engine gas dynamics](#)

Child Systems: [Throttle](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	4
Decision (D1)	NA	63% (5/8) decision outcomes

Saturate block "Limit to Positive"

Parent: [fuelsys/engine gas dynamics/Throttle & Manifold](#)

Metric	Coverage
Cyclomatic Complexity	2
Decision (D1)	50% (2/4) decision outcomes

Decisions Analyzed

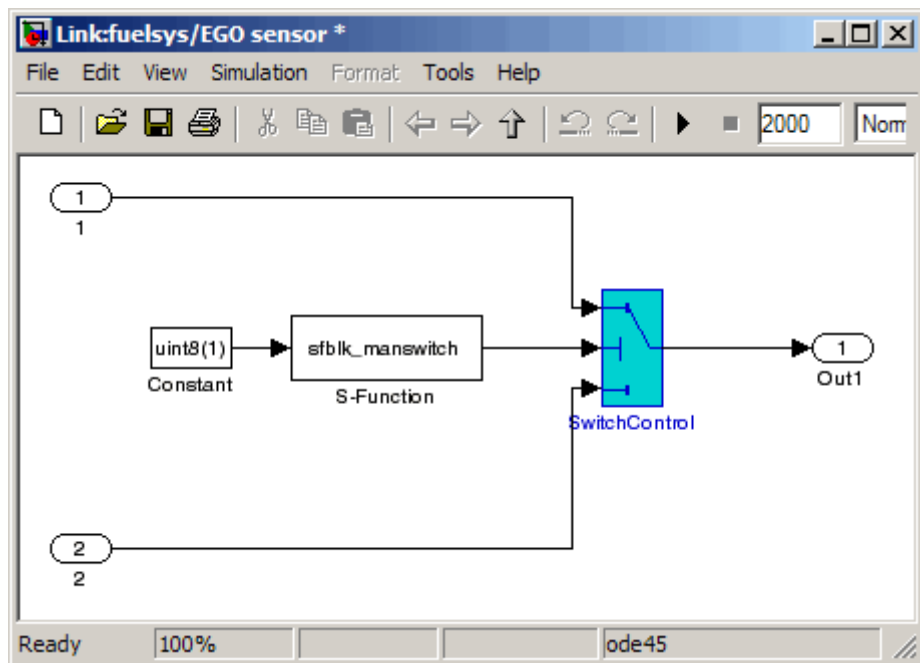
The Decisions analyzed table lists possible outcomes for a decision and the number of times that an outcome occurred in each test simulation. Outcomes that did not occur are in red highlighted table rows.

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

Decisions analyzed:

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

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



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

#1: function em2 out = lib em2(em2 in)**Decisions analyzed:**

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

Conditions Analyzed

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

Conditions analyzed:

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

MCDC Analysis

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

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

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

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

The MC/DC analysis table shows a condition case expression to represent a condition case. A condition case expression is a character string where:

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

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

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

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

If you select **Treat Simulink Logic blocks as short-circuited** in the Coverage Settings dialog box (see “Model Coverage Reporting Options” on page 5-16), MC/DC coverage analysis does not verify whether short-circuited inputs actually occur. The MC/DC analysis table uses an x in a condition expression (for example, TFxxx) to indicate short-circuited inputs that were not analyzed by the tool.

If you enable this feature and Logic blocks are short-circuited while collecting model coverage, you may not be able to achieve 100% coverage for that block.

Navigation Arrows. The section for each block contains a backward and a forward arrow. Click the forward arrow to go to the next section in the

report that lists an uncovered outcome. Click the back arrow to return to the previous uncovered outcome in the report.

Cumulative Coverage

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

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

A cumulative coverage report contains information about:

- **Current Run** — The coverage results of the simulation just completed.
- **Delta** — Percentage of coverage added to the cumulative coverage achieved with the simulation just completed. If the previous simulation's cumulative coverage and the current coverage are nonzero, the delta may be 0 if the new coverage does not add to the cumulative coverage.
- **Cumulative** — The total coverage collected for the model up to, but not including, the simulation just completed.

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

Summary

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

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

Decisions analyzed:

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

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

Conditions analyzed:

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

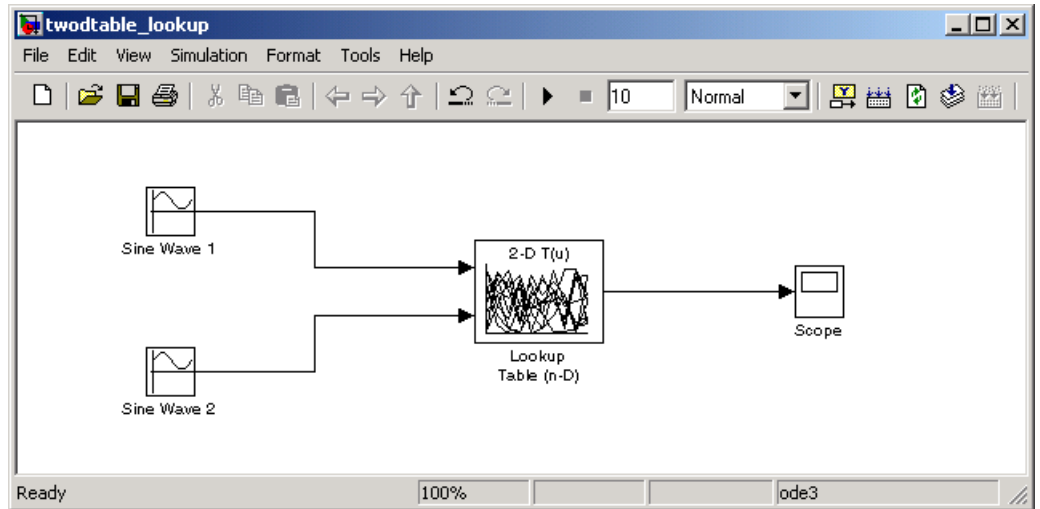
The MC/DC analysis **#n True Out** and **#n False Out** columns show the condition cases for each test case. The **Total Out T** and **Total Out F** column show the cumulative results.

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

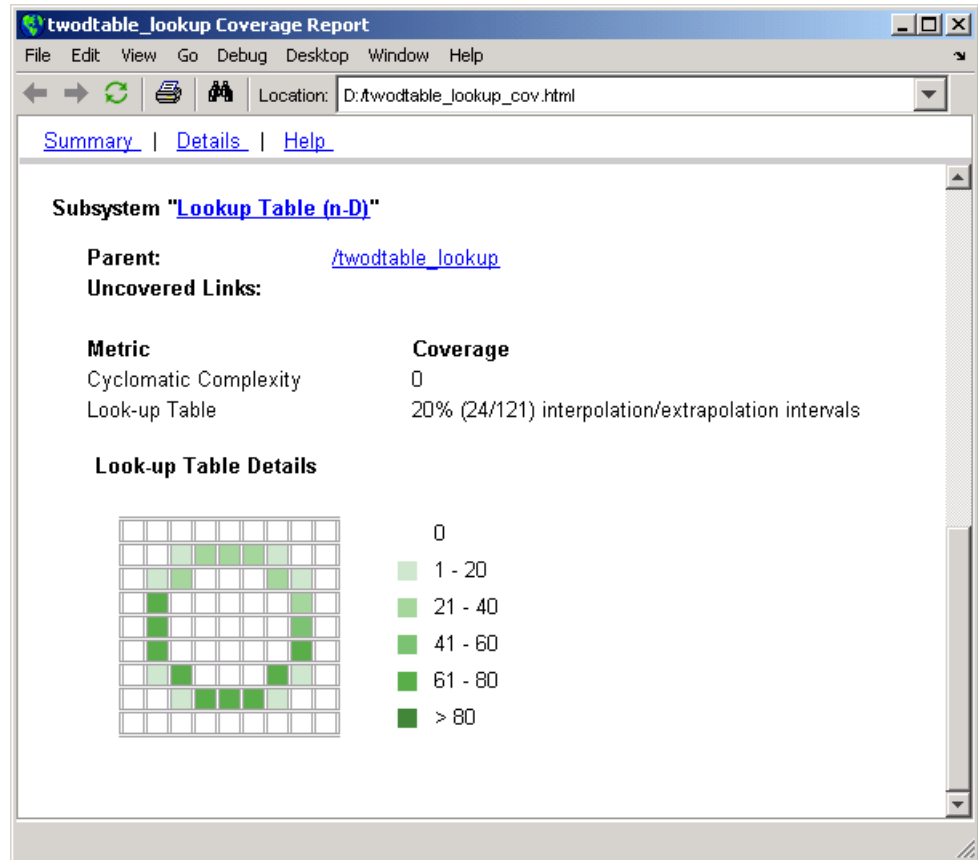
Decision/Condition:	#1 True Out	#1 False Out	#2 True Out	#2 False Out	Total Out T	Total Out F
Transition trigger expression						
Condition 1, "in(GS.Active.Coupled)"	(Txx)	FTF	(Txx)	(FTF)	(Txx)	FTF
Condition 2, "alt_ctrl"	FFx	FTF	FFx	(FTF)	FFx	FTF
Condition 3, "wow"	(FTT)	FTF	(FTT)	(FTF)	(FTT)	FTF

N-Dimensional Lookup Table

The following interactive chart summarizes the extent to which elements of a lookup table are accessed. In this example, two Sine Wave blocks generate *x* and *y* indices that access a Lookup Table (n-D) block of 10-by-10 elements filled with random values.



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

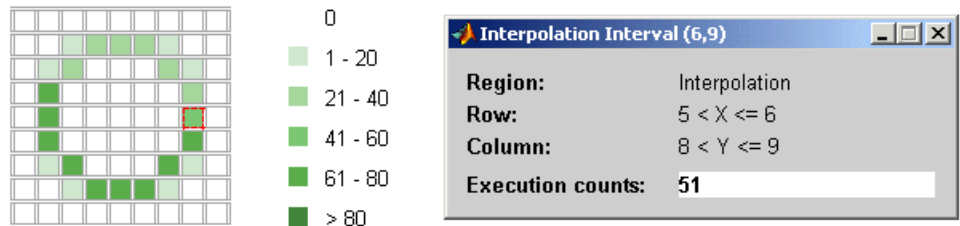


The report contains a two-dimensional table representing the elements of the lookup table. The element indices are represented by the cell border grid lines, which number 10 in each dimension. Areas where the lookup table interpolates between table values are represented by the cell areas. Areas of extrapolation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

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

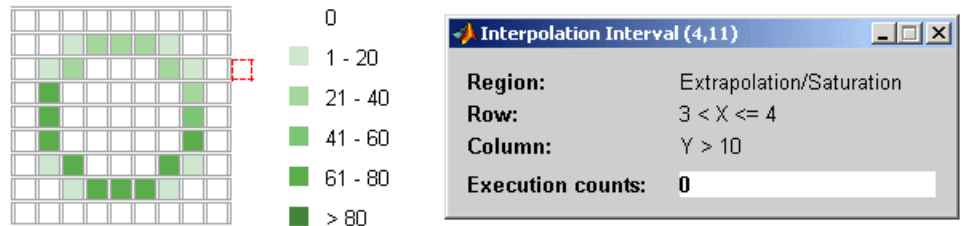
If you click an individual table cell, you see a dialog box that displays the index location of the cell and the exact number of execution counts generated for it during testing. The following example shows the contents of a color-shaded cell on the right edge of the circle.

Lookup Table Details



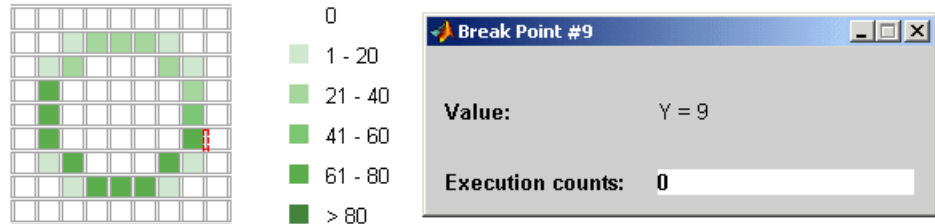
The selected cell is outlined in red. You can also click the extrapolation cells on the edge of the table.

Lookup Table Details

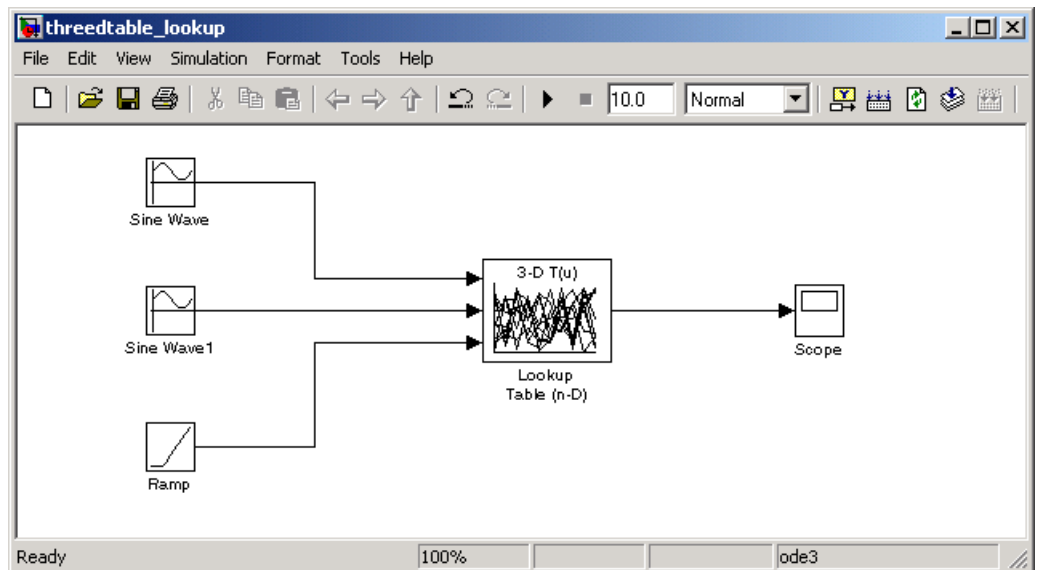


A bold grid line indicates that at least one block input equal to its exact index value occurred during the simulation. Click the border to display the exact number of hits for that index value.

Lookup Table Details

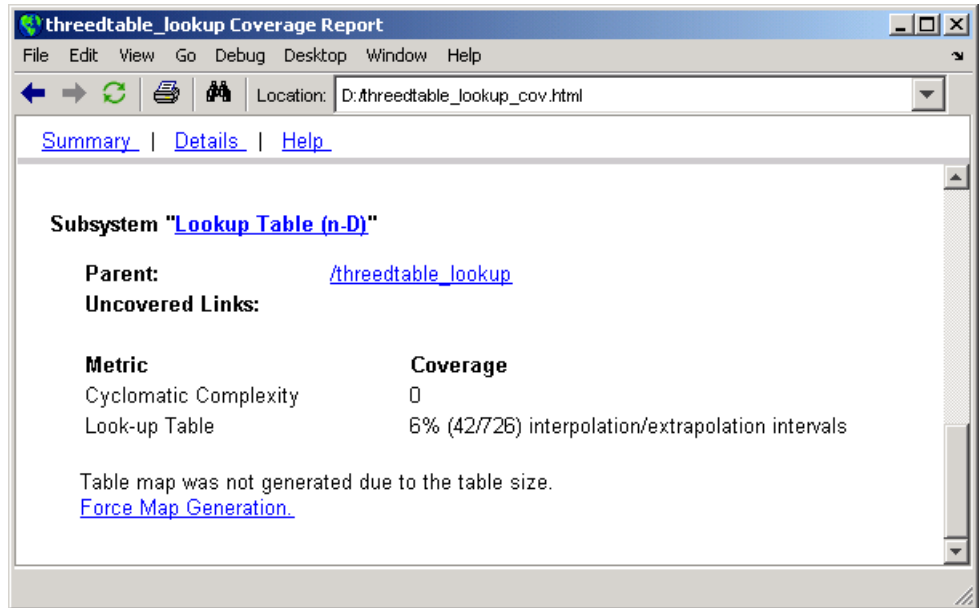


The following example model uses a Lookup Table (n-D) block of 10-by-10-by-5 elements filled with random values.

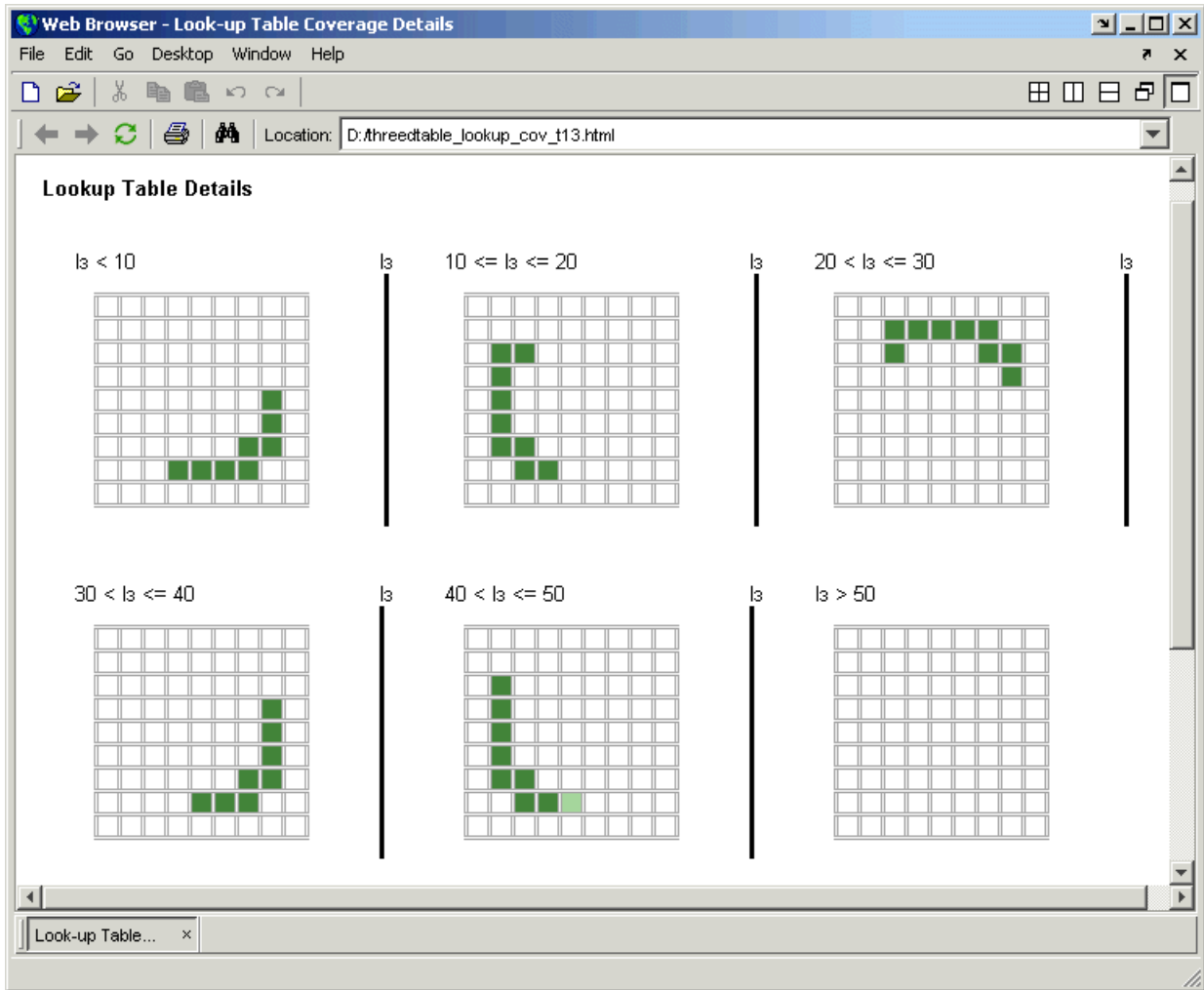


Both the x and y table axes have the indices 1, 2, ..., 10. The z axis has the indices 10, 20, ..., 50. Lookup table values are accessed with x and y indices that the two Sine Wave blocks generated, in the preceding example, and a z index that a Ramp block generates.

After simulation, you see the following lookup table report.



Instead of a two-dimensional table, you see the link Force Map Generation , which displays the following tables



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

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

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

Block Reduction

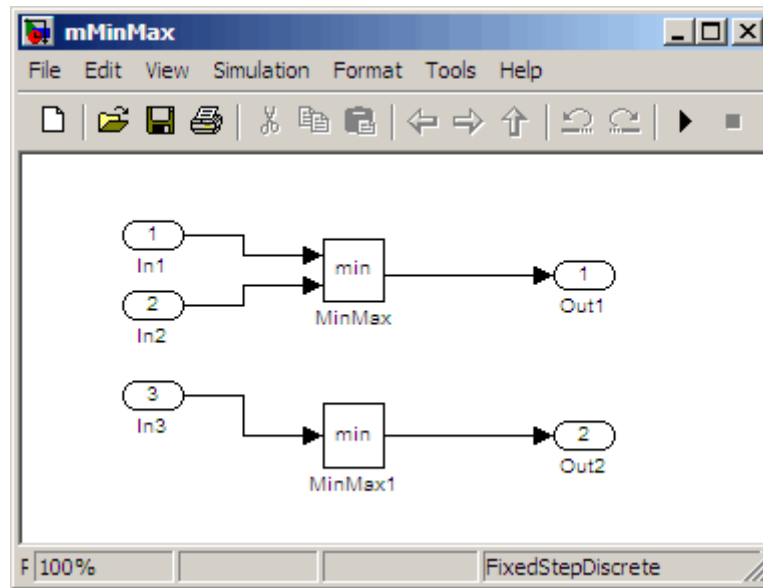
All model coverage reports indicate the status of the Simulink **Block reduction** parameter at the beginning of the report. In the following example, you set **Force block reduction off**.

Simulation Optimization Options	
Inline Parameters	off
Block Reduction	forced off
Conditional Branch Optimization	on

In the next example, you enabled the Simulink **Block reduction** parameter, and you did not set **Force block reduction off**.

Simulation Optimization Options	
Inline Parameters	off
Block Reduction	on
Conditional Branch Optimization	on


Consider the following model where the simulation does not execute the MinMax1 block because there is only one input—the constant 3.



If you set **Force block reduction off**, the report contains no coverage data for this block because the minimum input to the MinMax1 block is always 1

MinMax block "[MinMax1](#)"

Parent: [/mMinMax](#)

Uncovered Links: 

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

Decisions analyzed:

Logic to determine output	0%
input 1 is the minimum	0/0

If you do not set **Force block reduction off**, the report contains no coverage data for reduced blocks.

Reduced Blocks

Blocks eliminated from coverage analysis by block reduction model simulation setting:

... [mMinMax/MinMax1](#)

Signal Range Analysis

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

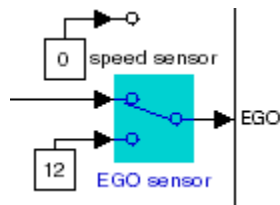
Access the Signal Range Analysis report quickly with the **Signal Ranges** link in the nonscrolling region at the top of the model coverage report, as shown for the `fuelSys` model.

[Summary](#) | [Details](#) | [Signal Ranges](#) | [Help](#)

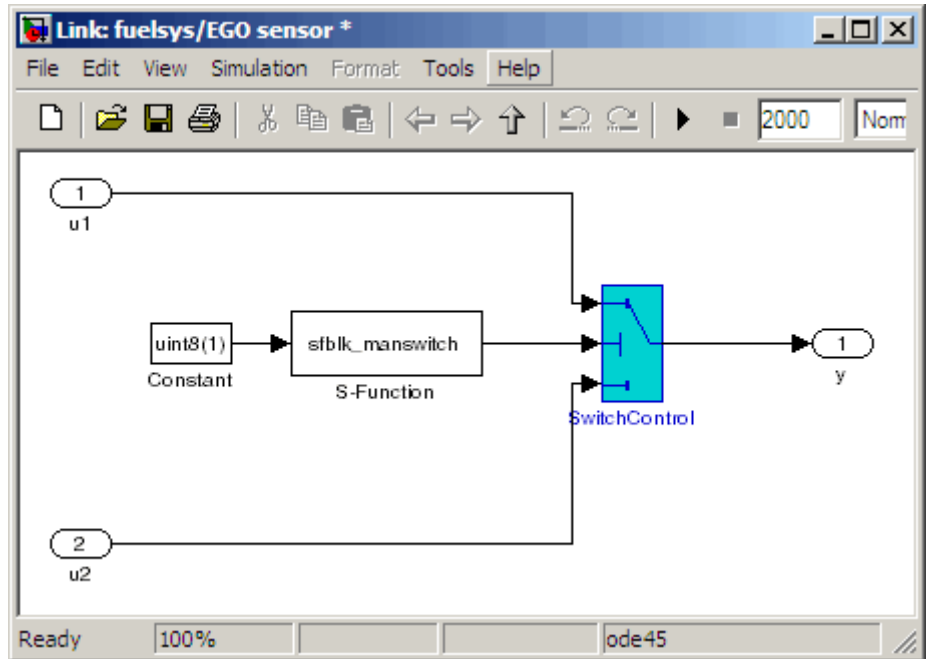
Signal Ranges:

Hierarchy	Min	Max
fuelsys		
... Constant2	-	-
... Constant3	12	12
... Constant4	0	0
... Constant5	0	0
... High Speed (rad./Sec.)	-	-
... Nominal Speed	300	300
... EGO sensor		
..... SwitchControl	0.229199	1
... MAP sensor		
..... SwitchControl	0.405559	0.889674
... engine speed		

Each block is reported in hierarchical fashion; child blocks appear directly under parent blocks. Each block name in the **Signal Ranges** report is a link. For example, select the EGO sensor link to display this block highlighted in its native diagram.



Select the SwitchControl link to display this block in its own subsystem by looking under the mask for EGO sensor.



Signal Size Coverage for Variable-Dimension Signals

If you select **Signal Size Coverage**, the software creates a Variable Signal Widths section after the Signal Ranges data in the model coverage report. This section lists the maximum and minimum signal sizes for all output ports in the model that have variable-size signals. It also lists the memory that Simulink allocated for that signal, as measured during simulation. This list does *not* include signals whose size does not vary during simulation.

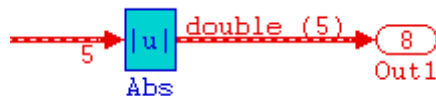
The following example shows the Variable Signal Widths section in a coverage report.

Variable Signal Widths:

Hierarchy	Min	Max	Allocated
... Abs	2	5	5
... Abs1	4	4	5
... MinMax1	2	5	5
... Switch	2	5	5
... Switch1	2	5	5
... Selector	4	4	5
... c2ri			
..... out1	4	4	5
..... out2	4	4	5
... Subsystem			
..... LogicalOperator	1	2	2
..... Switch1	1	2	2
..... Switch2	1	2	2

Each block is reported in hierarchical fashion; child blocks appear directly under parent blocks. Each block name in the Variable Signal Widths list is a link.

In this example, the Abs block signal size varied from 2 to 5, with an allocation of 5. Click the Abs link in the report. The Model Editor becomes current, with the Abs block highlighted.

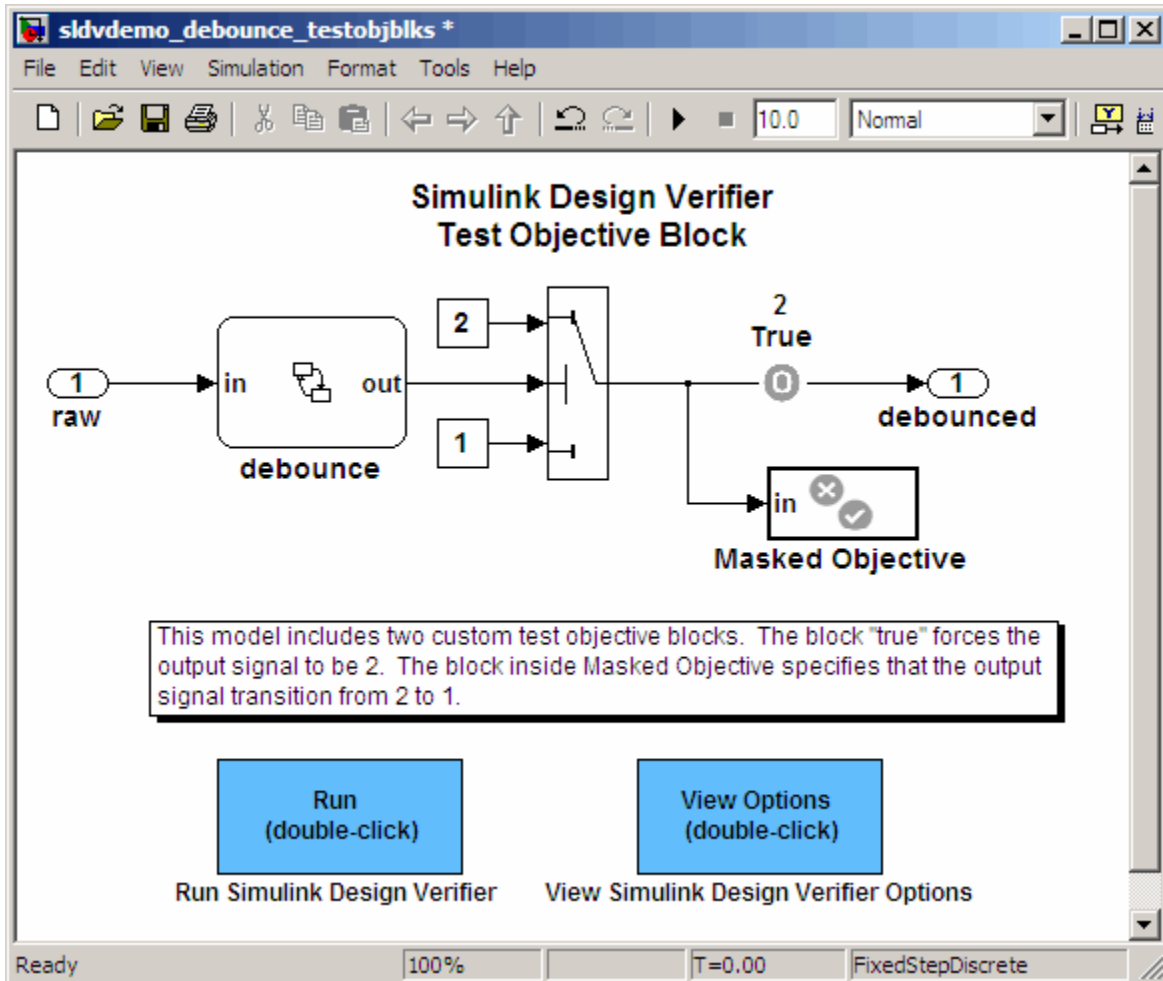


After the analysis, the variable-size signals have a wider line design. `double (5)` in this example indicates the data type and allocation for that signal.

Simulink Design Verifier Block Coverage

If you select **Simulink Design Verifier**, the analysis collects coverage data for all Simulink Design Verifier blocks in your model.

For an example of how this works, consider the `sldvdemo_debounce_testobjblks` model.



This model contains two Test Objective blocks:

- The True block defines a property that the signal have a value of 2.
- The Edge block, in the Masked Objective subsystem, describes the property where the output of the AND block in the Masked Objective subsystem changes from 2 to 1.

The Simulink Design Verifier software analyzes this model and produces a harness model that contains test cases that achieve certain test objectives. To see if the original model achieves those objectives, simulate the harness model and collect model coverage data. The model coverage tool analyzes any decision points or values within an interval that you specify in the Test Objective block.

In this example, the coverage report shows that you achieved 100% coverage of the True block because the signal value was 2 at least once. The signal value was 2 in 6 out of 14 time steps.

Design Verifier Test Objective block "True"

Parent: [sldvdemo debounce testobjblks harness/Test Unit \(copied from sldvdemo debounce testobjblks\)](#)

Metric	Coverage
Test Objective	100% (1/1) objective outcomes

Points/Intervals analyzed:

Point : 2	6/14
-----------	------

The input signal to the Edge block achieved a value of True once out of 14 time steps.

Design Verifier Test Objective block "[Edge](#)"

Parent: [sldvdemo debounce testobjblks harness/Test Unit \(copied from sldvdemo debounce testobjblks\)/Masked Objective](#)

Metric	Coverage
Test Objective	100% (1/1) objective outcomes

Points/Intervals analyzed:

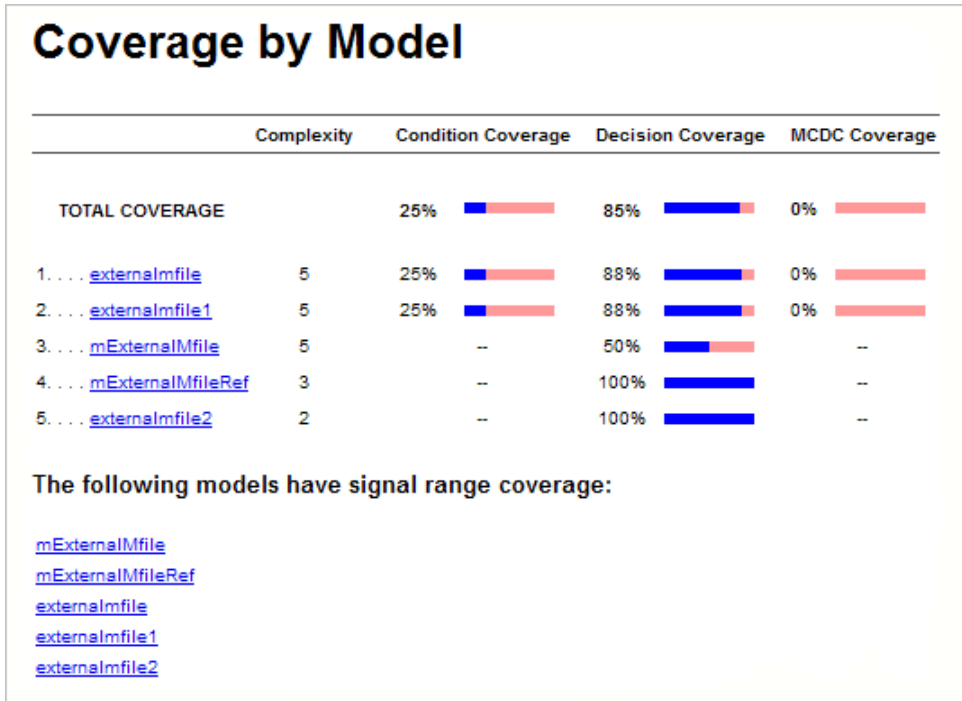
Point : T	1/14
-----------	------

Model Summary Reports

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

The summary report lists and provides links to coverage reports for all Model block referenced models and external M-files called by Embedded MATLAB code in the model. For more information, see “Model Reference Coverage Reports” on page 5-62 and “External M-File Coverage Reports” on page 5-62.

The following graphic shows an example of a model summary report. It contains links to the model coverage report (`mExternalMfile`), a report for the Model block (`mExternalMfileRef`), and three external M-files called from the model (`externalmfile`, `Iexternalmfile1`, and `externalmfile2`).



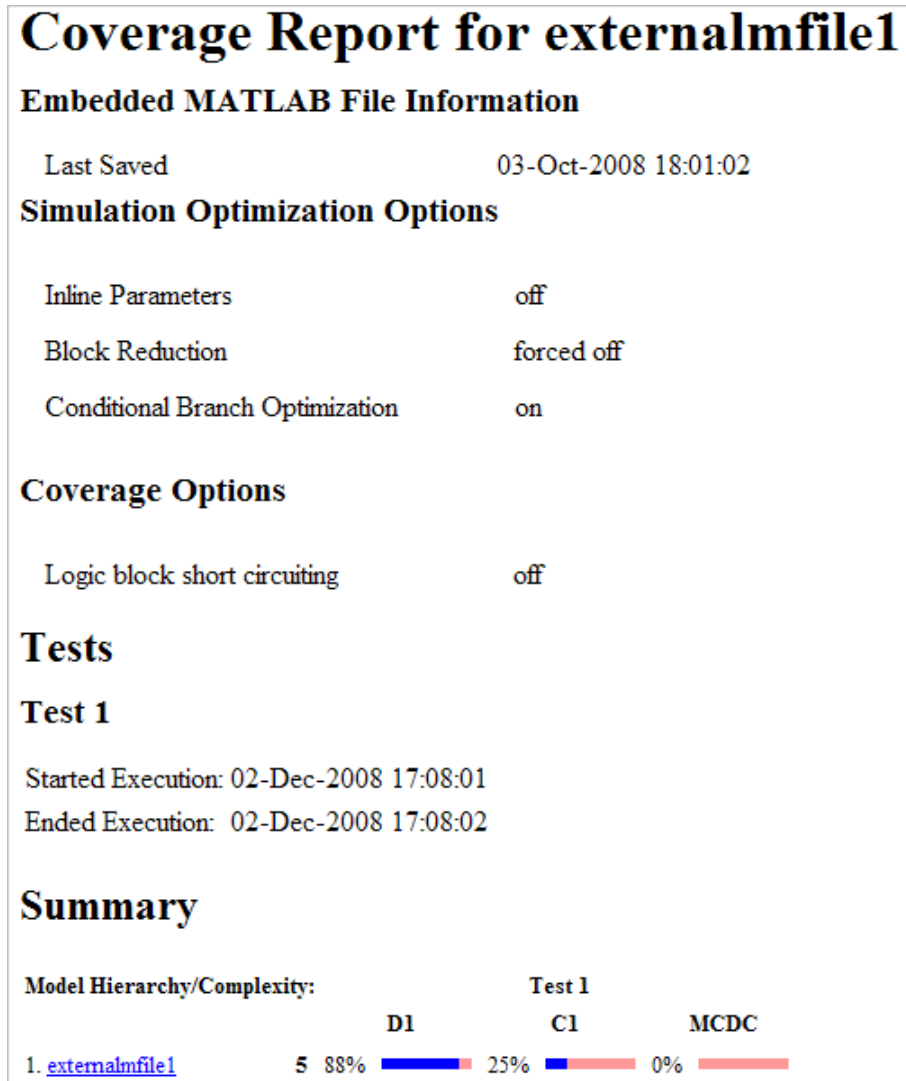
Model Reference Coverage Reports

If your top-level model references a model in a Model block, the software creates a separate report, named *reference_model_name_cov.html*, that includes coverage for the referenced model. This report has the same format as the “Model Coverage Reports” on page 5-31. Coverage results are recorded as if the referenced model was a standalone model; the report gives no indication that the model is referenced in a Model block.

External M-File Coverage Reports

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

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



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

Details:

1. Embedded MATLAB file "[externalmfile](#)"

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

Embedded MATLAB function "[externalmfile](#)"

Parent: [externalmfile](#)

Uncovered Links:

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

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


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

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

#2: function y = externalmfile1(u)

Decisions analyzed:

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

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

Decisions analyzed:

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

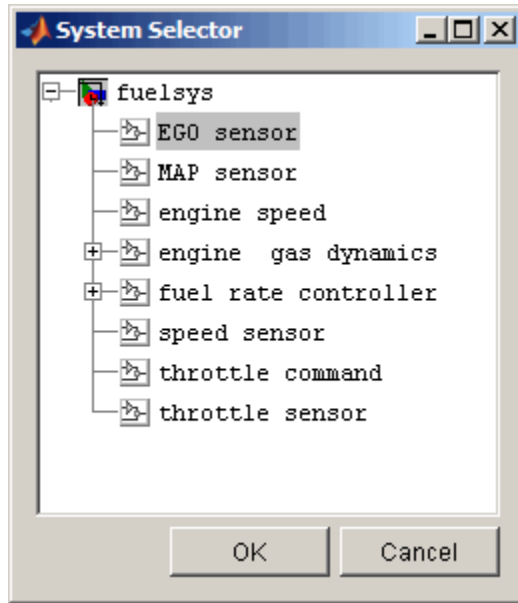
Subsystem Coverage Reports

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

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

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

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



The report is similar to the model coverage report, except that it includes only results for the EGO sensor subsystem and its contents.

Coverage Report for fuelsys



Summary

Model Hierarchy/Complexity: Test 1
D1
1. [EGO sensor](#) 2 50%

Details:

1. Subsystem "[EGO sensor](#)"

Parent: [/fuelsys](#)

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

Test 1

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

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

Colored Simulink Diagram Coverage Display

In this section...

- “How Model Coverage Highlighting Works” on page 5-69
- “Enabling the Colored Diagram Display” on page 5-69
- “Displaying Model Coverage with Model Coloring” on page 5-70
- “Accessing Coverage Information for Colored Blocks” on page 5-72

How Model Coverage Highlighting Works

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

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

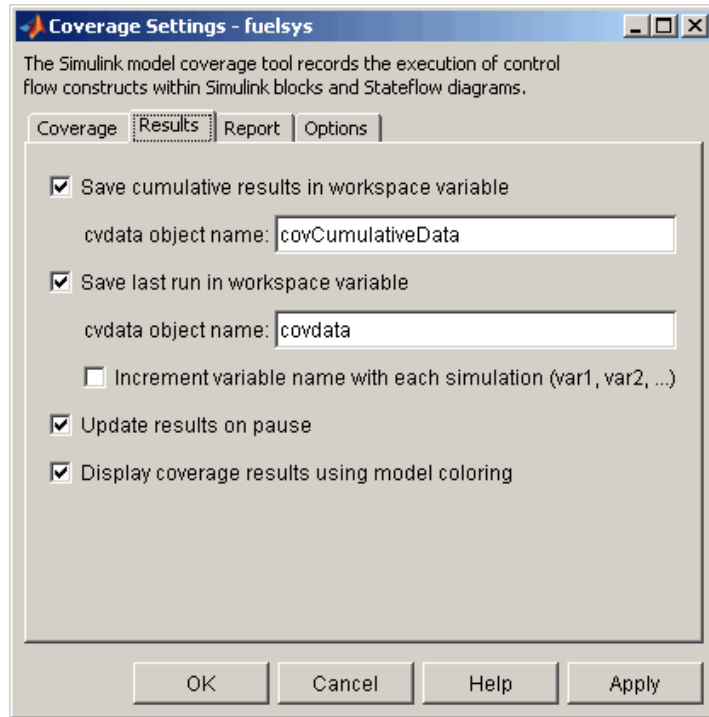
Coloring highlights structural coverage in Simulink models. When you enable coloring for model coverage results (see “Enabling the Colored Diagram Display” on page 5-69), the tool highlights blocks that received the following types of model coverage:

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

Enabling the Colored Diagram Display

To enable the model coverage colored diagram display:

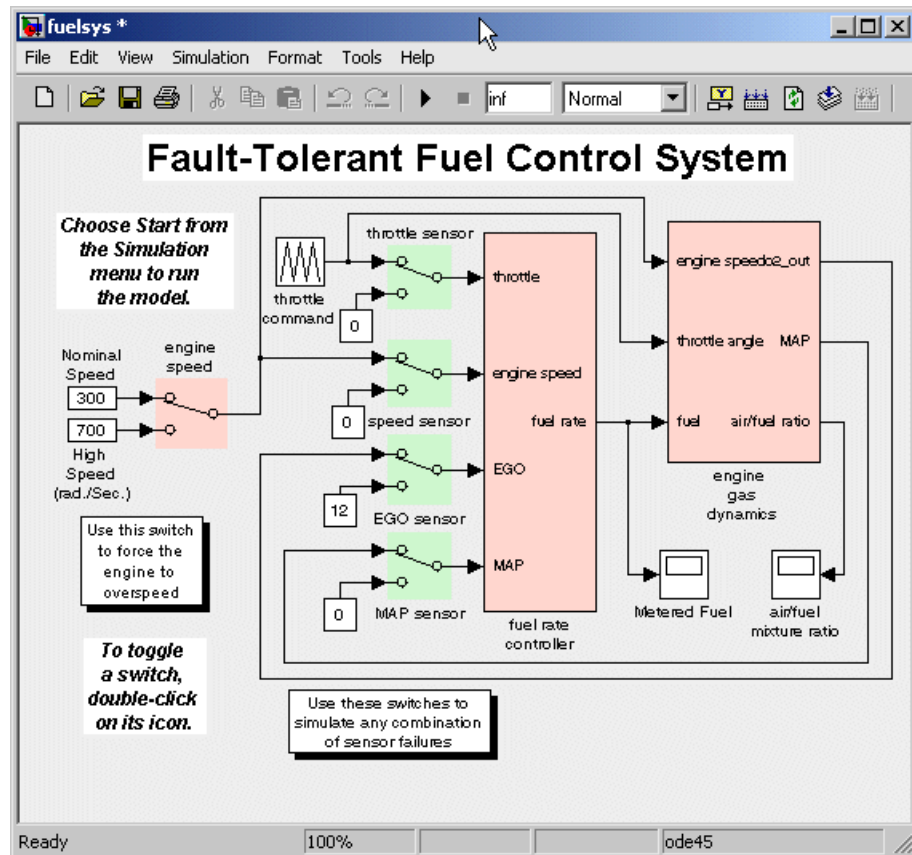
- 1** In the Simulink window, from the **Tools** menu, select **Coverage Settings**.
- 2** In the **Coverage** tab, select **Coverage for this model**.
- 3** Select the **Results** tab.



The **Display coverage results using model coloring** option is selected by default for all models. This check box becomes visible only after **Coverage for this model** is enabled. You can clear 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 5-69. 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.

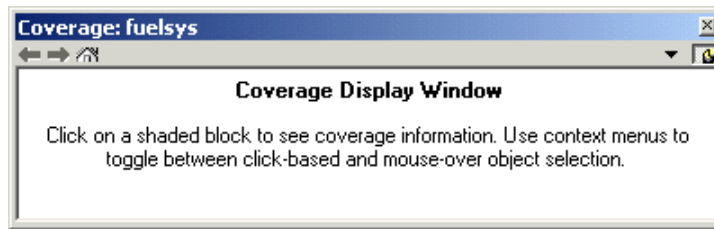


The light green Manual Switch blocks received full coverage during testing. The light red blocks (the engine speed Manual Switch block, and the fuel rate controller and engine gas dynamics subsystems) received incomplete coverage during testing. Blocks with no color highlighting (the Constant blocks,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

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

Accessing Coverage Information for Colored Blocks

“Displaying Model Coverage with Model Coloring” on page 5-70 describes the highlighted Simulink diagram that appears after simulation when you enable display coverage with model coloring. Along with the highlighted Simulink diagram, a Coverage Display window appears.



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



Summary coverage information appears in the Coverage Display window for the block, whose hyperlinked name is 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**.

To set the Coverage Display window to display coverage for a block in response to a hovering mouse cursor (instead of a mouse click):

- On the right side of the Coverage Display window, select the down arrow. From the resulting menu, select **Focus**. Or,
- Right-click a colored block. From the context menus, select **Coverage** followed by **Display details on mouse-over** .

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

Using Model Coverage Commands

In this section...

“About Model Coverage Commands” on page 5-74

“Creating Tests with `cvtest`” on page 5-74

“Running Tests with `cvsim`” on page 5-76

“Producing HTML Reports with `cvhtml`” on page 5-77

“Saving Test Runs to a File with `cvsave`” on page 5-78

“Loading Stored Coverage Test Results with `cvload`” on page 5-79

“Coverage Script Example” on page 5-79

About Model Coverage Commands

Using model coverage commands lets you automate the entire model coverage process with MATLAB scripts. You can use model coverage commands to set up model coverage tests, execute them in simulation, and store and report the results. For a list of the model coverage commands that the Simulink Verification and Validation software provides, see Chapter 10, “Function Reference”.

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

Creating Tests with `cvtest`

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

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

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

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

To create a test object with a specified label (used for reporting results):

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

To create a test with a setup command:

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

You execute the setup command in the base MATLAB workspace, just prior to running the instrumented simulation. Use this command for loading data prior to a test.

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

Field	Description
<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 for analysis
<code>label</code>	String for reporting results
<code>setupCmd</code>	Command executed prior to simulation
<code>settings.condition</code>	Set to 1 for condition coverage
<code>settings.decision</code>	Set to 1 for decision coverage
<code>settings.designverifier</code>	Set to 1 for coverage for Simulink Design Verifier blocks.
<code>settings.mcdc</code>	Set to 1 for MCDC coverage
<code>settings.sigrange</code>	Set to 1 for signal range coverage
<code>settings.sigsize</code>	Set to 1 for signal size coverage.
<code>settings.tableExec</code>	Set to 1 for lookup table coverage

Field	Description
<code>modelRefSettings.enable</code>	String specifying one of the following values: <ul style="list-style-type: none"> • <code>Off</code> — Disables coverage for all referenced models • <code>all</code> — Enables coverage for all referenced models • <code>filtered</code> — Enables coverage for only referenced models not listed in the <code>excludedModels</code> subfield
<code>modelRefSettings.excludeTopModel</code>	Set to 1 for excluding coverage for the top model
<code>modelRefSettings.excludedModels</code>	String specifying a comma-separated list of referenced models for which coverage is disabled when <code>modelRefSettings.enable</code> specifies <code>filtered</code>
<code>emlSettings.enableExternal</code>	Set to 1 to enable coverage for external M-files called by Embedded MATLAB functions in your model
<code>options.forceBlockReduction</code>	Set to 1 to override the Simulink Block reduction parameter if it is enabled.

Running Tests with `cvsim`

Use the `cvsim` command to simulate a test specification object.

Note You do not have to enable model coverage reporting (see “Creating and Running Test Cases” on page 5-11) 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`. When recording coverage for multiple models in a hierarchy, `cvsim` returns its results in a `cv.cvdatalogroup` object.

You can also control the simulation in a `cvsim` command by using parameters for the Simulink `sim` command:

- 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)
```

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

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

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

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

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

Producing HTML Reports with `cvhtml`

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

The following command creates an HTML report of the coverage results in the `cvdata` object `cvdo`. The results are written to the file `file` in the current MATLAB folder.

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

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

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

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

You can specify the detail level of the report with the value of `detail`, an integer between 0 and 3:

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

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

Saving Test Runs to a File with `cvsave`

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

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

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

```
cvsave(filename, cvt01, cvt02, ...)
```

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

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

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

Loading Stored Coverage Test Results with `cvload`

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

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

The `cvtest` objects that are 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 `restorettotal` is 1, the cumulative results from prior runs are restored:

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

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

cvload Special Considerations

When using the `cvload` command, be aware of the following considerations:

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

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. MCDC 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, at the MATLAB command prompt, enter `simcovdemo`.

Using Model Coverage Commands for Referenced Models

In this section...

“Introduction” on page 5-81

“Creating a Test Group with `cv.cvtestgroup`” on page 5-84

“Running Tests with `cvsimref`” on page 5-84

“Extracting Results from `cv.cvdatagroup`” on page 5-85

Introduction

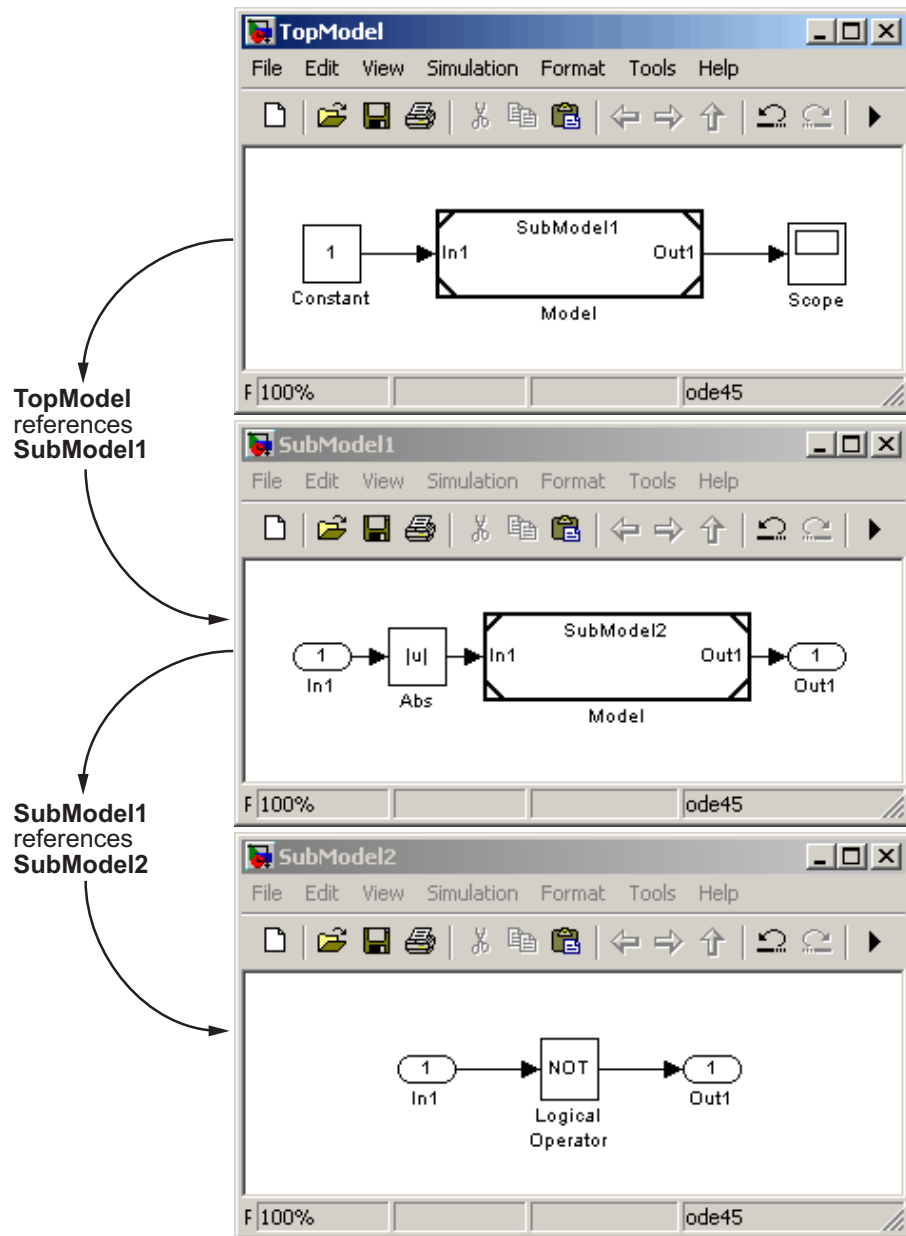
Using Simulink software, you can include one model in another with Model blocks. Each Model block represents a reference to another model, called a *referenced model* or *submodel*. A referenced model can contain Model blocks that reference other models. You can construct a hierarchy of referenced models, in which the topmost model is called the *top model*. For more information, see “Referencing a Model” in *Simulink User’s Guide* .

Model coverage supports referenced models that operate in Normal mode. You can record coverage only for those Model blocks whose **Simulation mode** parameter specifies Normal. You can use model coverage commands to record coverage for referenced models (see “Using Model Coverage Commands” on page 5-74). However, if you want to record different types of coverage for models in a hierarchy, you must use the `cvsimref` function. The following steps describe a basic workflow for using this function to obtain model coverage results for Model blocks.

Step	Description	See...
1	Use <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 5-84

Step	Description	See...
2	Use <code>cv.simref</code> to simulate the top model in a hierarchy and record coverage results for its referenced models.	“Running Tests with <code>cv.simref</code> ” on page 5-84
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 5-85

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 5-74). 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`, so that you can customize the contents of the `cv.cvtestgroup` object to meet your needs. For more information, see the documentation for the `cv.cvtestgroup` function.

Running Tests with `cvsimref`

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

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

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

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

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

You can use parameters from the Simulink `sim` function in a `cvsimref` command to control the simulation:

- To return the simulation time vector `t`, matrix of state values `x`, and matrix of output values `y`:

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

- To override default simulation values with new values:

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

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

Extracting Results from `cv.cvdatabroup`

Once you simulate a test group with `cvsimref`, the function returns results that reside in a `cv.cvdatabroup` 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.cvdatabroup` object provides methods, such as `allNames` and `get`, so that you can extract individual `cvdata` objects. For example, enter the following command to obtain a cell array that lists all model names associated with the data group `cvdg`:

```
modelName = cvdg.allNames
```

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

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

After you extract the individual `cvdata` objects, you can use other model coverage commands to use the coverage data of a particular model. For example, you can use the `cvhtml` function to create and display an HTML

report of the coverage results (see “Producing HTML Reports with cvhtml” on page 5-77).

Model Coverage for Embedded MATLAB Function Blocks

In this section...

“Types of Model Coverage in Embedded MATLAB Function Blocks” on page 5-87

“Creating a Model with Embedded MATLAB Function Block Decisions” on page 5-88

“Understanding Embedded MATLAB Function Block Model Coverage” on page 5-92

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 only if you have a Simulink Verification and Validation software license.

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

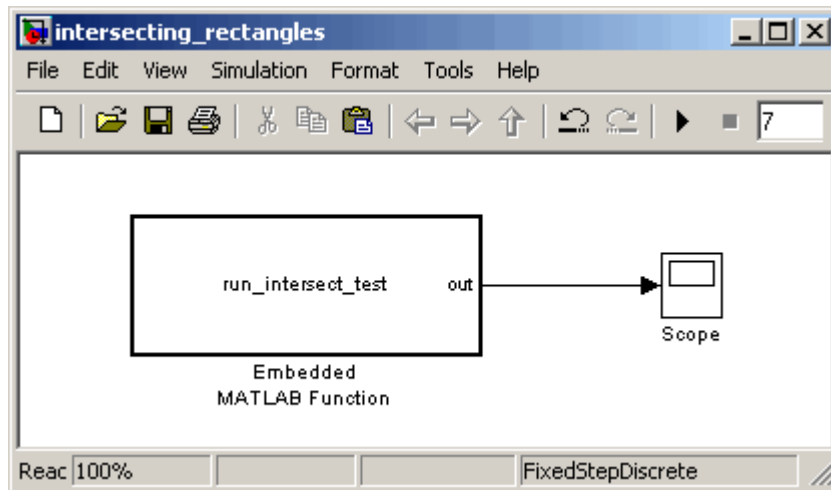
- **Function header** — Decision coverage is 100% if the function or subfunction is executed.
- **if** — Decision coverage is 100% if the **if** expression evaluates to true at least once, and false at least once.
- **switch** — Decision coverage is 100% if every **switch** case is taken, including the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

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

- if statement conditions
- while statement conditions, if present

Creating a Model with Embedded MATLAB Function Block Decisions

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



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

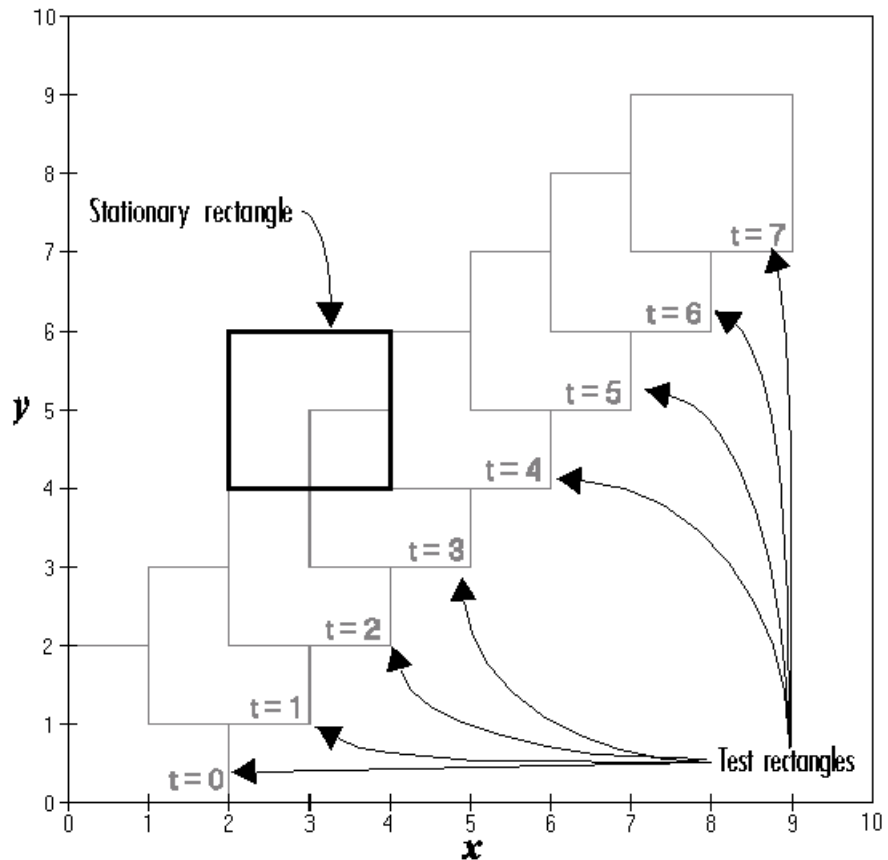

```
Embedded MATLAB Editor - Block: intersecting_rectangles/Embedded ...
File Edit Text Debug Tools Window Help
[Icons]

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

Ready Ln 1 Col 1
```

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

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

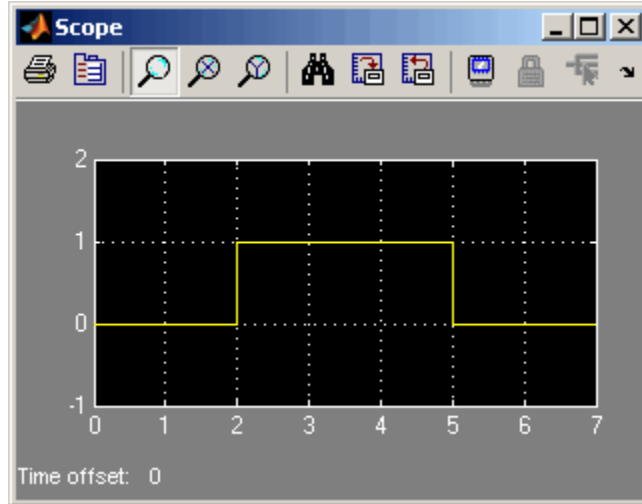


The fixed rectangle is shown in bold with a lower-left origin of (2, 4) and a width and height of 2. At time $t = 0$, the first test rectangle has an origin of (0, 0) and a width and height of 2. For each succeeding sample, the origin of the test rectangle increments by (1, 1). The rectangles at sample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

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

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

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



Understanding Embedded MATLAB Function Block Model Coverage

You can specify that model coverage reports generate automatically after a simulation. For instructions on how to specify a model coverage report, see “Creating and Running Test Cases” on page 5-11 .

After the simulation, 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 5-88, in model-block-function order, is in the following table.

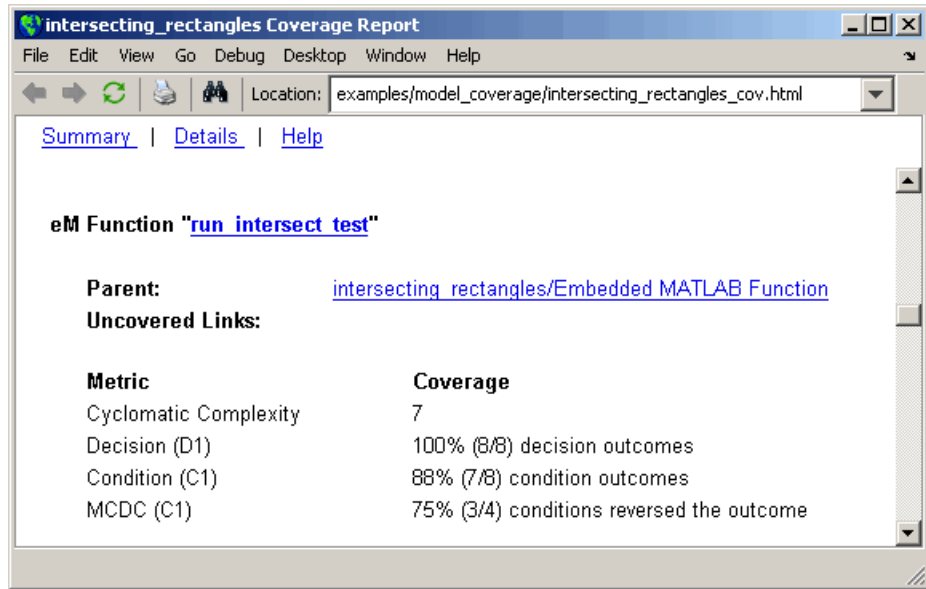
Model:	intersecting_rectangles
Block:	Embedded MATLAB Function

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

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

Model Coverage for the Embedded MATLAB Function Block Function run_intersect_test

You see model coverage for the Embedded MATLAB Function block function `run_intersect_test` 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. The coverage metrics for `run_intersect_test` include decision, condition, and MCDC coverage. You can best understand these metrics by examining the code listing for `run_intersect_test`.

```

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

```

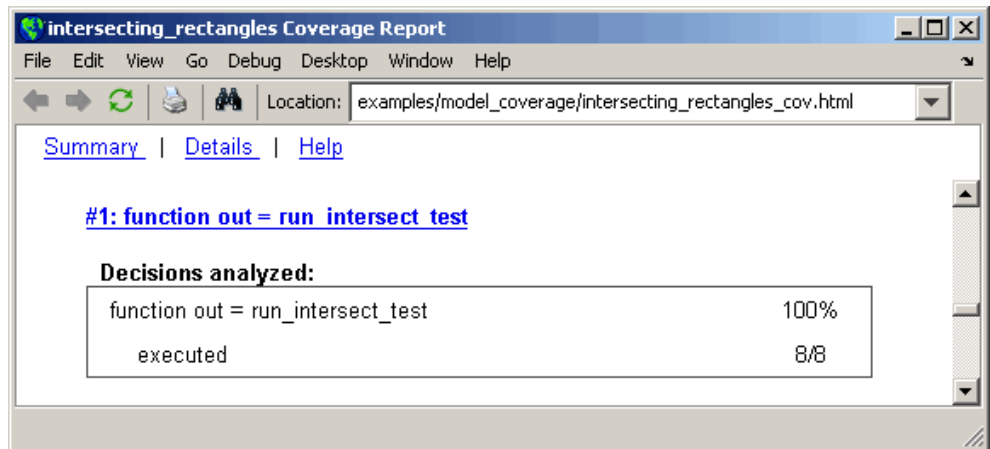
Lines with coverage elements are marked by a highlighted line number in the listing. Line 1 receives decision coverage on whether the top-level function `run_intersect_test` is executed. Line 6 receives decision coverage for its `if` statement. Line 14 receives decision coverage on whether the subfunction

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

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

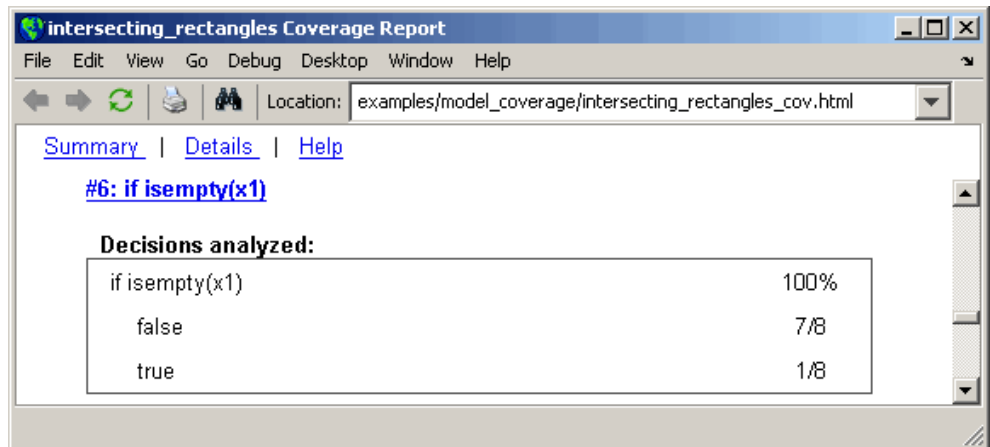
The following sections display the coverage for each `run_intersect_test` decision line. The coverage for each line is titled with the line itself, which is linked to display the function with the line highlighted.

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



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

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



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

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



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

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

#27: if (top1 < bottom2 || top2 < bottom1)

Decisions analyzed:

if (top1 < bottom2 top2 < bottom1)	100%
false	5/8
true	3/8

Conditions analyzed:

Description:	True	False
top1 < bottom2	2	6
top2 < bottom1	1	5

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

Decision/Condition:	True Out	False Out
top1 < bottom2 top2 < bottom1		
top1 < bottom2	Tx	FF
top2 < bottom1	FT	FF

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

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

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

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

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

The screenshot shows a web browser window titled "intersecting_rectangles Coverage Report". The address bar shows the location: "examples/model_coverage/intersecting_rectangles_cov.html". The page has navigation links for "Summary", "Details", and "Help".

The main content area displays the following information for line #30:

#30: if (right1 < left2 || right2 < left1)

Decisions analyzed:

if (right1 < left2 right2 < left1)	100%
false	3/5
true	2/5

Conditions analyzed:

Description:	True	False
right1 < left2	0	5
right2 < left1	2	3

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

Decision/Condition:	True Out	False Out
right1 < left2 right2 < left1		
right1 < left2	(Tx)	FF
right2 < left1	FT	FF

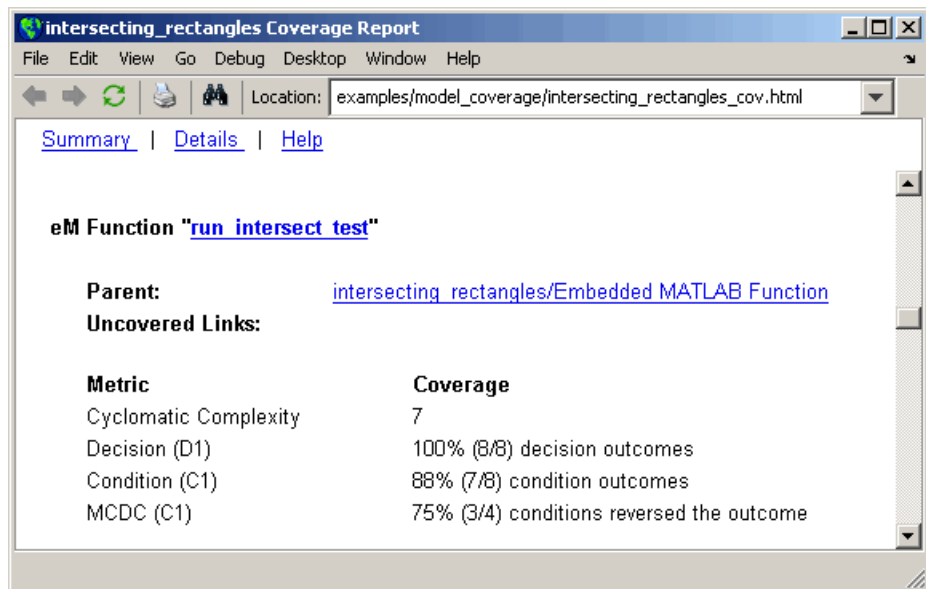
The line 30 decision, `if (right1 < left2 || right2 < left1)`, is nested in the `if` statement of the line 27 decision and is evaluated only if the line 27 decision is false. Because the line 27 decision evaluated false five times, line 30 is evaluated five times, three of which are false. Because both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

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

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

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

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



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

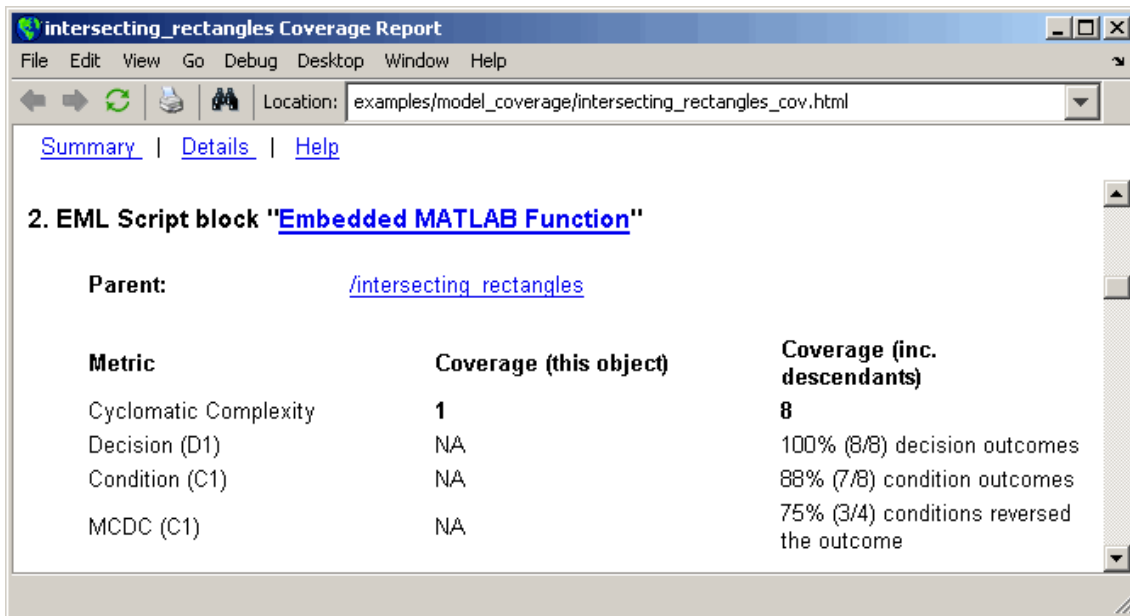
- There are eight decision outcomes reported for run_intersect_test in the line reports: one for line 1 (executed), two for line 6 (true and false), one for line 14 (executed), two for line 27 (true and false), and two for line 30 (true and false). The decision coverage for each line shows 100% decision

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

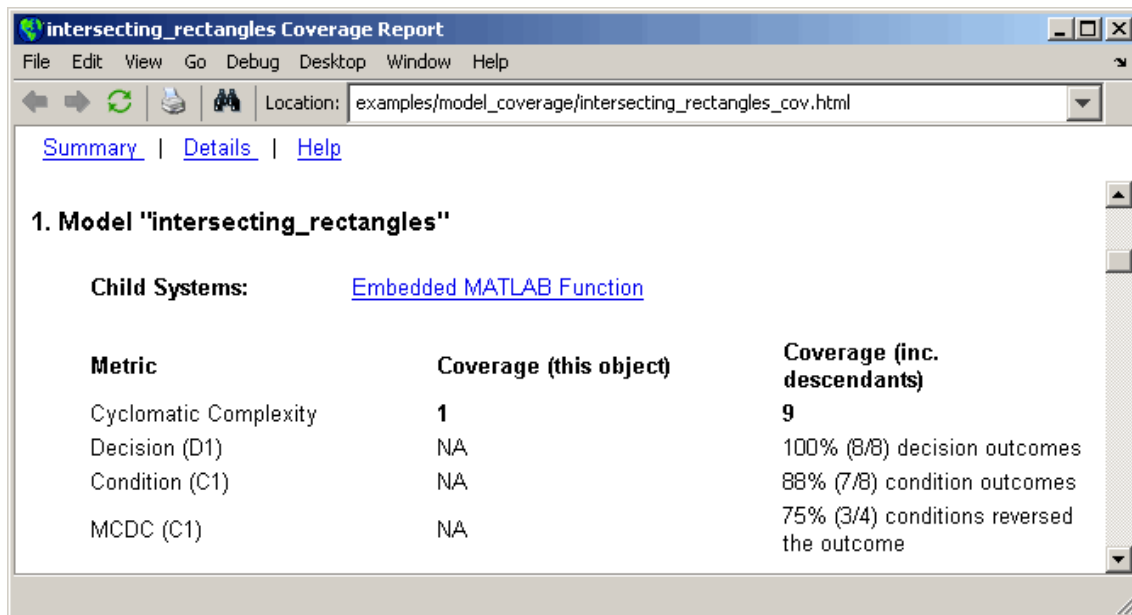
- There are four conditions reported for `run_intersect_test` in the line reports. Lines 27 and 30 each have two conditions, and each condition has two condition outcomes (true and false), for a total of eight condition outcomes in `run_intersect_test`. All conditions tested positive for both the true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

Model Coverage for the Embedded MATLAB Function Block and the Model

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



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.



The screenshot shows a web browser window titled "intersecting_rectangles Coverage Report". The address bar shows the location: "examples/model_coverage/intersecting_rectangles_cov.html". There are navigation links for "Summary", "Details", and "Help".

1. Model "intersecting_rectangles"

Child Systems: [Embedded MATLAB Function](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	9
Decision (D1)	NA	100% (8/8) decision outcomes
Condition (C1)	NA	88% (7/8) condition outcomes
MCDC (C1)	NA	75% (3/4) conditions reversed the outcome

Customizing the Model Advisor

- Chapter 6, “Overview of the Model Advisor”
- Chapter 7, “Authoring Custom Checks”
- Chapter 8, “Creating Custom Configurations by Organizing Checks and Folders”
- Chapter 9, “Deploying Custom Configurations”

Overview of the Model Advisor

- “Why Use and Customize the Model Advisor?” on page 6-2
- “Customizing and Using the Model Advisor Workflow” on page 6-4
- “Before Customizing the Model Advisor” on page 6-5

Why Use and Customize the Model Advisor?

In this section...
“About the Model Advisor” on page 6-2
“Customizing the Model Advisor” on page 6-2

About the Model Advisor

The Model Advisor is a GUI that provides a way for you to check a Simulink model or subsystem for consistent modeling guidelines, using MathWorks checks. Using the checks, you can easily apply these guidelines across projects and development teams. For more information, see “Consulting the Model Advisor” in the Simulink documentation.

The Model Advisor includes MathWorks checks, which help you define and implement consistent design guidelines. Running the checks reviews your model for conditions and configuration settings that cause inaccurate or inefficient simulation and code generation of the system that the model represents. The Model Advisor displays different MathWorks checks depending on which products you have installed. For more information on individual checks, see:

- “Simulink Checks”
- “Real-Time Workshop® Checks”
- “Simulink® Verification and Validation Checks” on page 14-2

Customizing the Model Advisor

The Simulink Verification and Validation product allows you to extend the capabilities of the Model Advisor. Using Model Advisor APIs and the Model Advisor Configuration Editor, you can:

- Customize the behavior of the Model Advisor by defining your own custom checks, and writing your own callback functions.
- Organize checks and folders to create custom Model Advisor configurations.

- Create multiple custom configurations that you use for different projects or modeling guidelines, and switch between these configurations in the Model Advisor.
- Deploy the custom configurations to your users.

For more information, see “Customizing and Using the Model Advisor Workflow” on page 6-4.

Customizing and Using the Model Advisor Workflow

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

- 1** Review the information in “Before Customizing the Model Advisor” on page 6-5.
- 2** Optionally, author custom checks in a customization file. For detailed information, see Chapter 7, “Authoring Custom Checks”.
- 3** Organize checks into new and existing folders to create custom configurations. To organize the Model Advisor, use the Model Advisor Configuration Editor or create M-code in a customization file. For detailed information, see Chapter 8, “Creating Custom Configurations by Organizing Checks and Folders”.
- 4** Optionally, deploy custom configurations. For detailed information, see Chapter 9, “Deploying Custom Configurations”.
- 5** Verify that models comply with modeling guidelines using the Model Advisor. For detailed information, see “Consulting the Model Advisor”.

Before Customizing the Model Advisor

Before customizing the Model Advisor:

- If you want to create custom checks, know how to create an M-file script. For more information, see “M-File Scripts” in the MATLAB documentation.
- If you want to create custom checks, understand how to access model constructs that you want to check. For example, know how to find block and model parameters. For more information on using utilities for creating check callbacks, see “Common Utilities for Authoring Checks” on page 7-23.
- Identify which MathWorks checks you want to include in your custom Model Advisor configuration.

When you are ready to create a custom configuration, follow the “Customizing and Using the Model Advisor Workflow” on page 6-4. Each section provides you with detailed examples of how to create custom checks and configurations in the Model Advisor.

Authoring Custom Checks

- “Authoring Checks Workflow” on page 7-2
- “Customization File Overview” on page 7-3
- “Register Checks and Process Callbacks” on page 7-6
- “Defining Custom Checks” on page 7-11
- “Creating Callback Functions and Results” on page 7-22

Authoring Checks Workflow

- 1** On your MATLAB path, create a *customization file* called `sl_customization.m`. In this file, create a `sl_customization()` function to register the custom checks that you create and optional process callbacks with the Model Advisor. For detailed information, see “Register Checks and Process Callbacks” on page 7-6.
- 2** Define custom checks and where they appear in the Model Advisor. For detailed information, see “Defining Custom Checks” on page 7-11.
- 3** Specify what actions you want the Model Advisor to take for the custom checks by creating a check callback function for each custom check. For detailed information, see “Creating Callback Functions and Results” on page 7-22.
- 4** Optionally, specify what automatic fix operations the Model Advisor performs by creating an action callback function. For detailed information, see “Action Callback Function” on page 7-37.
- 5** Optionally, specify startup and post-execution actions by creating a process callback function. For detailed information, see “Defining Startup and Post-Execution Actions Using Process Callback Functions” on page 7-8.

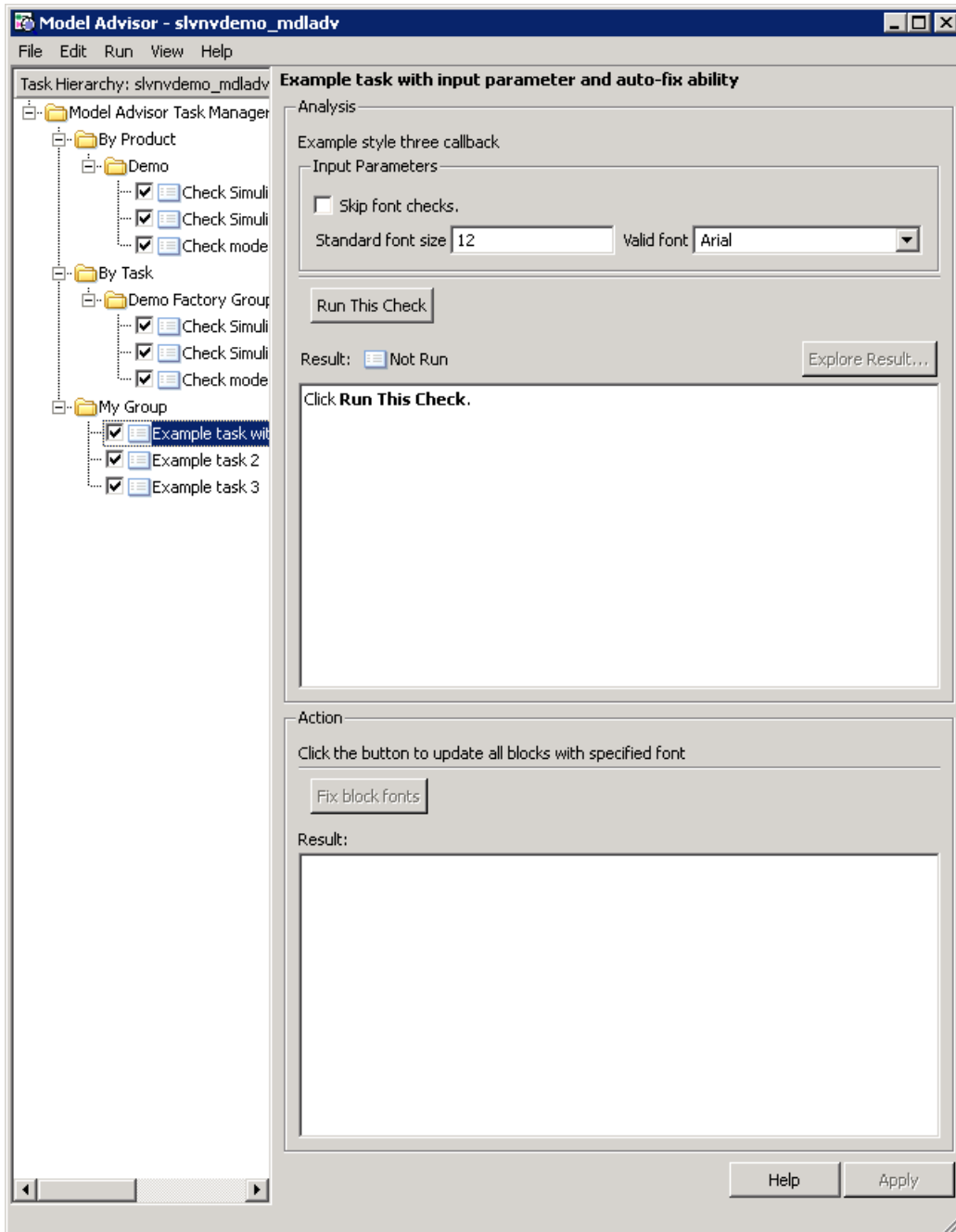
Customization File Overview

A *customization file* is an M-file that you create and name `sl_customization.m`. The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

Function	Description	When Required
<code>sl_customization()</code>	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at startup (see “Register Checks and Process Callbacks” on page 7-6).	Required for all customizations to the Model Advisor.
One or more check definitions	Defines all custom checks (see “Defining Custom Checks” on page 7-11).	Required for custom checks and to add custom checks to the By Product folder.
Check callback functions	Defines the actions of the custom checks (see “Creating Callback Functions and Results” on page 7-22).	Required for custom checks. You must write one callback function for each custom check.
One or more calls to check input parameters	Specifies input parameters to custom checks (see “Defining Check Input Parameters” on page 7-16).	Optional.
One or more calls to check list views	Specifies calls to the Model Advisor Result Explorer for custom checks (see “Defining Model Advisor Result Explorer Views” on page 7-18).	Optional.

Function	Description	When Required
One or more calls to check actions	Specifies actions the software performs for custom checks (see “Defining Check Actions” on page 7-19 and “Action Callback Function” on page 7-37).	Optional.
One process callback function	Specifies actions to be performed at startup and post-execution time (see “Defining Startup and Post-Execution Actions Using Process Callback Functions” on page 7-8).	Optional.

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



Register Checks and Process Callbacks

In this section...

“Create `sl_customization` Function” on page 7-6

“Registering Checks and Process Callbacks” on page 7-6

“Defining Startup and Post-Execution Actions Using Process Callback Functions” on page 7-8

Create `sl_customization` Function

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

Tip

- You can have more than one `sl_customization.m` file on your MATLAB path.
- Do not place an `sl_customization.m` file that customizes checks and folders in the Model Advisor in your root MATLAB folder or any of its subfolders, except for the `matlabroot/work` folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

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

Registering Checks and Process Callbacks

To register custom checks and process callbacks, the customization manager includes the following methods:

- `addModelAdvisorCheckFcn` (*@checkDefinitionFcn*)

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

The *checkDefinitionFcn* argument is a handle to the function that defines all custom checks that you want to add to the Model Advisor as instances of the `ModelAdvisor.Check` class (see “Defining Custom Checks” on page 7-11).

- `addModelAdvisorProcessFcn` (*@modelAdvisorProcessFcn*)

Registers the process callback function for the Model Advisor checks (see “Defining Startup and Post-Execution Actions Using Process Callback Functions” on page 7-8).

Caution The Model Advisor registers only one process callback function. If you have more than one `sl_customization.m` file on your MATLAB path, the Model Advisor registers the process callback function from the `sl_customization.m` file that has the highest priority.

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

Model Advisor Code Example: Registering Custom Checks and Process Callbacks

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

```
function sl_customization(cm)

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

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

Note If you add custom tasks and folders within the `sl_customization.m` file, include methods for registering the tasks and folders in the `sl_customization` function. For more information, see “Registering Tasks and Folders” on page 8-14.

Defining Startup and Post-Execution Actions Using Process Callback Functions

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

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

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

Process Callback Function Arguments

The process callback function takes the following arguments.

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

Model Advisor Code Example: Process Callback Function

The following code is an example of a process callback function that specifies actions in the `configure` stage, to make only custom checks visible. In the `process_results` stage, this function displays information at the MATLAB command line for checks that do not pass.

```
% Process Callback Function
% Defines actions to execute at startup and post-execution
function [checkCellArray taskCellArray] = ...
    ModelAdvisorProcessFunction(stage, system, checkCellArray, taskCellArray)
switch stage
    % Specify the appearance of the Model Advisor window at startup
```

```
case 'configure'
    for i=1:length(checkCellArray)
        % Hide all checks that do not belong to custom folder
        if isempty(strfind(checkCellArray{i}.ID, 'mathworks.example'))
            checkCellArray{i}.Visible = false;
            checkCellArray{i}.Value = false;
        end
    end
end
% Specify actions to perform after the Model Advisor completes execution
case 'process_results'
    for i=1:length(checkCellArray)
        % Print message if check does not pass
        if checkCellArray{i}.Selected && (strcmp(checkCellArray{i}.Title, ...
            'Check Simulink window screen color'))
            mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
            % Verify whether the check was run and if it failed
            if mdladvObj.verifyCheckRan(checkCellArray{i}.ID)
                if ~mdladvObj.getCheckResultStatus(checkCellArray{i}.ID)
                    % Display text in MATLAB Command Window
                    disp(['Example message from Model Advisor Process'...
                        ' callback.']);
                end
            end
        end
    end
end
end
end
```

Defining Custom Checks

In this section...

“About Custom Checks” on page 7-11

“Contents of Check Definitions” on page 7-11

“Displaying and Enabling Checks” on page 7-13

“Defining Where Custom Checks Appear” on page 7-14

“Model Advisor Code Example: Check Definition Function” on page 7-15

“Defining Check Input Parameters” on page 7-16

“Defining Model Advisor Result Explorer Views” on page 7-18

“Defining Check Actions” on page 7-19

About Custom Checks

You can create a custom check to use in the Model Advisor. Creating custom checks provides you with the ability to specify which conditions and configuration settings the Model Advisor reviews.

You define custom checks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Check` class. Define one instance of this class for each custom check that you want to add to the Model Advisor, and register the custom check as described in “Register Checks and Process Callbacks” on page 7-6.

Tip You can add a check to multiple folders by creating a *task*. For more information, see “Adding a Check to Custom or Multiple Folders Using Tasks” on page 8-16.

The following sections describe how to define custom checks.

Contents of Check Definitions

When you define a Model Advisor check, it contains the information listed in the following table.

Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this id to access the check.
Handle to check callback function (required)	Function that specifies the contents of a check.
Check name (recommended)	Creates a name for the check that the Model Advisor displays.
Check properties (optional)	<p>Creates a user interface with the check. When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for <code>Visible</code> and <code>LicenseName</code>. For more information, see <code>ModelAdvisor.Check</code> and <code>ModelAdvisor.Task</code>.</p> <hr/> <p>Tip When you add checks to the Model Advisor as tasks, specify only the required properties of a check, because the task definition includes the additional properties. For example, you define the description of the check in the task definition using the <code>ModelAdvisor.Task.Description</code> property instead of the <code>ModelAdvisor.Check.TitleTips</code> property.</p> <hr/>
Input Parameters (optional)	Adds input parameters that request input from the user. The Model Advisor uses the input to perform the check.

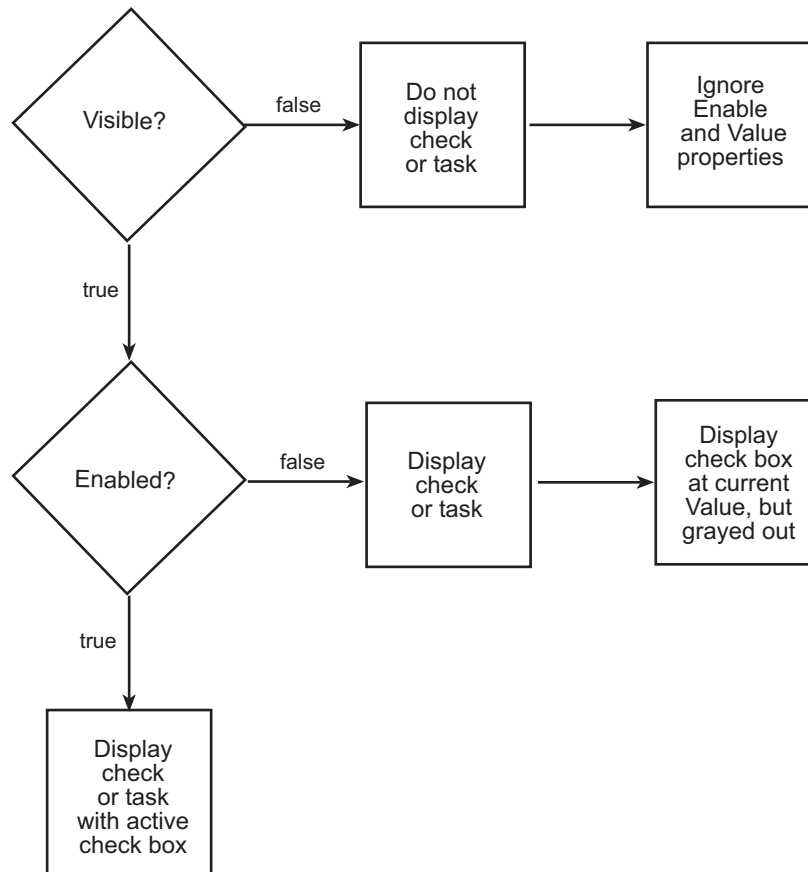
Contents	Description
Action (optional)	Adds automatic fixing action.
Explore Result button (optional)	Adds the Explore Result button that the user clicks to open the Model Advisor Result Explorer.

Displaying and Enabling Checks

You can create a check and specify how it appears in the Model Advisor. You can define when to display a check, or whether a user can select or clear a check using the `Visible`, `Enable`, and `Value` properties of the `ModelAdvisor.Check` class.

Note When adding checks to the Model Advisor as tasks, specify these properties in the `ModelAdvisor.Task` class. If you specify the properties in both `ModelAdvisor.Check` and `ModelAdvisor.Task`, the `ModelAdvisor.Task` properties take precedence, except for the `Visible` and `LicenseName` properties. For more information, see `ModelAdvisor.Task`.

Modify the behavior of the `Visible`, `Enable`, and `Value` properties in a process callback function (see “Defining Startup and Post-Execution Actions Using Process Callback Functions” on page 7-8). The following chart illustrates how these properties interact.



Defining Where Custom Checks Appear

Specify where the Model Advisor places custom checks using the following guidelines:

- To place a check in a new folder in the **Model Advisor** root, use the `ModelAdvisor.Group` class. See “Defining Custom Tasks” on page 8-15.
- To place a check in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See “Defining Custom Tasks” on page 8-15.

- To place a check in the **By Product** folder, use the `ModelAdvisor.Root.publish` method.

Model Advisor Code Example: Check Definition Function

The following is an example of a function that defines the custom checks associated with the callback functions described in “Creating Callback Functions and Results” on page 7-22. The check definition function returns a cell array of custom checks to be added to the Model Advisor.

The check definitions in the example use are used the tasks described in “Defining Custom Tasks” on page 8-15.

```
% Defines custom Model Advisor checks
function defineModelAdvisorChecks

% Sample check 1: Informational check
rec = ModelAdvisor.Check('mathworks.example.configManagement');
rec.Title = 'Informational check for model configuration management';
setCallbackFcn(rec, @modelVersionChecksumCallbackUsingFT,'None','StyleOne');
rec.CallbackContext = 'PostCompile';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample check 2: Basic Check with Pass/Fail Status
rec = ModelAdvisor.Check('mathworks.example.unconnectedObjects');
rec.Title = 'Check for unconnected objects';
setCallbackFcn(rec, @unconnectedObjectsCallbackUsingFT,'None','StyleOne');
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample Check 3: Check with Subchecks and Actions
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
rec.Title = 'Check safety-related optimization settings';
setCallbackFcn(rec, @OptimizationSettingCallback,'None','StyleOne');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
setCallbackFcn(modifyAction, @modifyOptmizationSetting);
modifyAction.Name = 'Modify Settings';
```

```
modifyAction.Description = ['Modify model configuration optimization' ...
                             ' settings that can impact safety.'];
modifyAction.Enable = true;
setAction(rec, modifyAction);
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);
```

Defining Check Input Parameters

With input parameters, the check author can request input from the user for a Model Advisor check. Define input parameters using the `ModelAdvisor.InputParameter` class inside a custom check function (see “Defining Custom Checks” on page 7-11). You must define one instance of this class for each input parameter that you want to add to a Model Advisor check.

Note You do not have to create input parameters for every custom check.

Specifying Input Parameter Layout

Specify the layout of input parameters in an input parameter definition. To place input parameters, use the following methods.

Method	Description
<code>ModelAdvisor.Check setInputParametersLayoutGrid</code>	Specifies the size of the input parameter grid.
<code>ModelAdvisor.InputParameter setRowSpan</code>	Specifies the number of rows the parameter occupies in the Input Parameter layout grid.
<code>ModelAdvisor.InputParameter setColSpan</code>	Specifies the number of columns the parameter occupies in the Input Parameter layout grid.

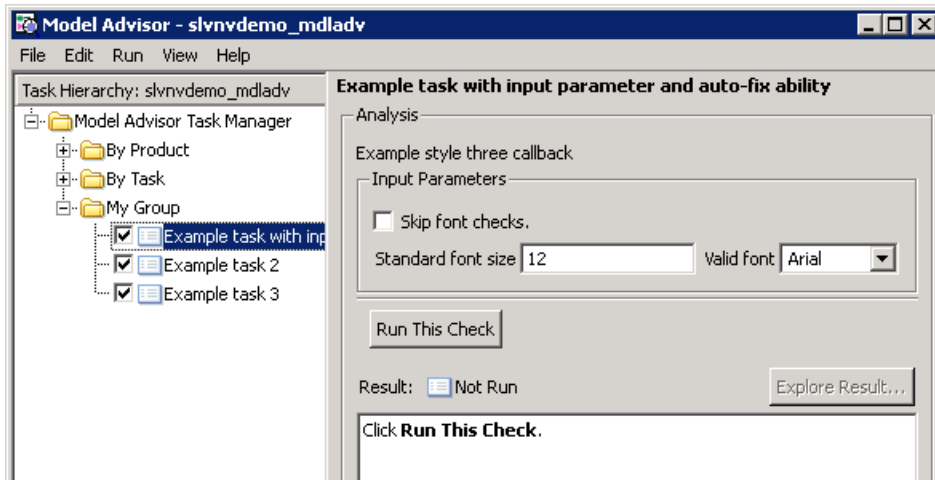
For information on using these methods, see the `ModelAdvisor.Check` and `ModelAdvisor.InputParameter` class documentation.

Model Advisor Code Example: Input Parameter Definition

The following is an example of defining input parameters that you add to a custom check. You must include input parameter definitions inside a custom check definition (see “Model Advisor Code Example: Check Definition Function” on page 7-15). The following code, when included in a custom check definition, creates three input parameters.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.setInputParametersLayoutGrid([3 2]);
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
```

The Model Advisor displays these input parameters in the right pane, in an **Input Parameters** box.



Defining Model Advisor Result Explorer Views

A *list view* provides a way for users to fix check warnings and failures using the Model Advisor Result Explorer. Creating a list view allows you to :

- Add the **Explore Result** button to the custom check in the Model Advisor window.
- Provide the information to populate the Model Advisor Result Explorer.

For information on using the Model Advisor Results Explorer, see “Batch-Fixing Warnings or Failures” in the Simulink documentation.

Define list views using the `ModelAdvisor.ListViewParameter` class inside a custom check function (see “Defining Custom Checks” on page 7-11). You must define one instance of this class for each list view that you want to add to a Model Advisor Result Explorer window.

Note You do not have to create list views for every custom check.

Model Advisor Code Example: List View Definition

The following is an example of defining list views. You must make the **Explore Result** button visible using the `ModelAdvisor.Check.ListViewVisible` property inside a custom check function, and include list view definitions inside a check callback function (see “Detailed Check Callback Function” on page 7-31).

The following code, when included in a check definition function, adds the **Explore Result** button to the check in the Model Advisor.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
% add 'Explore Result' button
rec.ListViewVisible = true;
```

The following code, when included in a check callback function, provides the information to populate the Model Advisor Result Explorer.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object');
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

Defining Check Actions

An *action* provides a way for you to specify an action that the Model Advisor performs to fix a Model Advisor check. When you define an action, the Model Advisor window includes an **Action** box below the **Analysis** box.

You define actions using the `ModelAdvisor.Action` class inside a custom check function (see “Defining Custom Checks” on page 7-11). You must define:

- One instance of this class for each action that you want to take.
- One action callback function for each action (see “Action Callback Function” on page 7-37).

Note

- Each check can contain only one action.
 - You do not have to create actions for every custom check.
-

Model Advisor Code Example: Action Definition

The following is an example of defining actions within a custom check. You must include action definitions inside a check definition function (see “Model Advisor Code Example: Check Definition Function” on page 7-15).

The following code, when included in a check definition function, provides the information to populate the **Action** box in the Model Advisor.

```
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
modifyAction.setCallbackFcn(@modifyOptimizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
                           ' settings that can impact safety'];
modifyAction.Enable = true;
rec.setAction(modifyAction);
```

The Model Advisor, in the right pane, displays an **Action** box.

Action

Modify model configuration optimization settings that can impact safety

Modify Settings

Result:

Creating Callback Functions and Results

In this section...

- “About Callback Functions” on page 7-22
- “Common Utilities for Authoring Checks” on page 7-23
- “Simple Check Callback Function” on page 7-23
- “Detailed Check Callback Function” on page 7-31
- “Check Callback Function with Hyperlinked Results” on page 7-33
- “Action Callback Function” on page 7-37
- “Formatting Model Advisor Results” on page 7-38

About Callback Functions

A *callback function* specifies the actions that the Model Advisor performs on a model or subsystem, based on the check or action that the user runs. You must create a callback function for each custom check and action so that the Model Advisor can execute the function when a user runs the check. There are several types of callback functions:

- “Simple Check Callback Function” on page 7-23
- “Detailed Check Callback Function” on page 7-31
- “Check Callback Function with Hyperlinked Results” on page 7-33
- “Action Callback Function” on page 7-37

All types of callback functions provide one or more return arguments for displaying the results after executing the check or action. In some cases, return arguments are strings or cell arrays of strings that support embedded HTML tags for text formatting. The MathWorks™ recommends that you use the Model Advisor Result Template API to format check results, as described in “Formatting Model Advisor Results” on page 7-38. Limit HTML tags to be compatible with alternate output formats.

Common Utilities for Authoring Checks

When you create a check, there are common Simulink utilities that you can use to make the check perform different actions. Following is a list of utilities and when to use them. In the Utility column, click the link for more information about the utility.

Utility	Used for...
find_system	Getting handle or path to: <ul style="list-style-type: none"> • Blocks • Lines • Annotations When getting the object, you can: <ul style="list-style-type: none"> • Specify a search depth • Search under masks and libraries
get_param / set_param	Getting and setting system and block parameter values.
inspect	Getting object properties. First you must get a handle to the object.
simget / simset	Getting and setting model simulation parameters.
evalin	Working in the base workspace.
Stateflow API	Programmatic access to Stateflow objects.

Simple Check Callback Function

Use a simple check callback function with results formatted using the Result Template API to indicate whether the model passed or failed the check, or to recommend correcting an issue. The keyword for this callback function is `StyleOne`. The check definition requires this keyword (see “Defining Custom Checks” on page 7-11).

The check callback function takes the following arguments.

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

Model Advisor Code Example: Informational Check Callback Function

The following code is an example of a callback function for a custom *informational* check that finds and displays the model configuration and checksum information. The informational check uses the Result Template API to format the check result.

An *informational* check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.

An informational check does not include the following items in the results:

- The check status. The Model Advisor displays the overall check status, but the status is not in the result.
- A description of the status.
- The recommended action to take when the check does not pass.
- Subcheck results.
- A line below the results.

```
% Sample Check 1 Callback Function: Informational Check
% Find and display model configuration and checksum information
% Informational checks do not have a passed or warning status in the results

function resultDescription = modelVersionChecksumCallbackUsingFT(system)
```



```

resultDescription = [];
system = getfullname(system);
model = bdroot(system);

% Format results in a list using Model Advisor Result Template API
ft = ModelAdvisor.FormatTemplate('ListTemplate');
% Add See Also section for references to standards
docLinkSfunction{1} = {'IEC 61508-3, Table A.8 (5)' ...
    ' 'Software configuration management' ' '};
setRefLink(ft,docLinkSfunction);

% Description of check in results
desc = 'Display model configuration and checksum information.';
% If running the Model Advisor on a subsystem, add note to description
if strcmp(system, model) == false
    desc = strcat(desc, ['<br/>NOTE: The Model Advisor is reviewing a ' ...
        ' sub-system, but these results are based on root-level settings.']);
end
setCheckText(ft, desc);

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% If err, use these values
mdlver = 'Error - could not retrieve Version';
mdlauthor = 'Error - could not retrieve Author';
mdldate = 'Error - could not retrieve Date';
mdlsum = 'Error - could not retrieve CheckSum';

% Get model configuration and checksum information
try
    mdlver = get_param(model,'ModelVersion');
    mdlauthor = get_param(model,'LastModifiedBy');
    mdldate = get_param(model,'LastModifiedDate');
    mdlsum = Simulink.BlockDiagram.getChecksum(model);
    mdlsum = [num2str(mdlsum(1)) ' ' num2str(mdlsum(2)) ' ' ...
        num2str(mdlsum(3)) ' ' num2str(mdlsum(4))];
    mdladvObj.setCheckResultStatus(true); % init to true
catch err
    mdladvObj.setCheckResultStatus(false);
    setSubResultStatusText(ft,err.message);
    resultDescription{end+1} = ft;

```

```
        return
    end

    % Display the results
    lbStr = '<br/>';
    resultStr = ['Model Version: ' mdlver lbStr 'Author: ' mdlauthor lbStr ...
        'Date: ' mdldate lbStr 'Model Checksum: ' mdlsum];
    setSubResultStatusText(ft,resultStr);

    % Informational checks do not have subresults, suppress line
    setSubBar(ft,false);
    resultDescription{end+1} = ft;
```

Model Advisor Code Example: Basic Check with Pass/Fail Status

Here is an example of a callback function for a custom *basic* check that finds and reports unconnected lines, input ports, and output ports.

A *basic* check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.
- The status of the check.
- A description of the status.
- Results for the check.
- The recommended actions to take when the check does not pass.

A basic check does not include the following items in the results:

- Subcheck results.
- A line below the results.

```
% Sample Check 2 Callback Function: Basic Check with Pass/Fail Status
% Find and report unconnected lines, input ports, and output ports
function ResultDescription = unconnectedObjectsCallbackUsingFT(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
```

```

% Initialize variables
mdladvObj.setCheckResultStatus(false);
ResultDescription = {};
ResultStatus = false; % Default check status is 'Warning'
system = getfullname(system);
isSubsystem = ~strcmp(bdroot(system), system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object
ft = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
if isSubsystem
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                  'output ports in the subsystem.'];
else
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                  'output ports in the model.'];
end
setCheckText(ft,checkDescStr);

% Add See Also section with references to applicable standards
checkStdRef = 'IEC 61508-3, Table A.3 (3) 'Language subset' ';
docLinkSfunction{1} = {checkStdRef};
setRefLink(ft,docLinkSfunction);

% Basic checks do not have subresults, suppress line
setSubBar(ft,false);

% Check for unconnected lines, inputs, and outputs
sysHandle = get_param(system, 'Handle');
uLines = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'line', ...
    'Connected', 'off');
uPorts = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'port', ...

```

```

        'Line', -1);

% Use parents of port objects for the correct highlight behavior
if ~isempty(uPorts)
    for i=1:length(uPorts)
        uPorts(i) = get_param(get_param(uPorts(i), 'Parent'), 'Handle');
    end
end

% Create cell array of unconnected object handles
modelObj = {};
searchResult = union(uLines, uPorts);
for i = 1:length(searchResult)
    modelObj{i} = searchResult(i);
end

% No unconnected objects in model
% Set result status to 'Pass' and display text describing the status
if isempty(modelObj)
    setSubResultStatus(ft,'Pass');
    if isSubsystem
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this subsystem.']);
    else
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this model.']);
    end
    ResultStatus = true;
% Unconnected objects in model
% Set result status to 'Warning' and display text describing the status
else
    setSubResultStatus(ft,'Warn');
    if ~isSubsystem
        setSubResultStatusText(ft,['The following lines, input ports, ' ...
            'or output ports are not properly connected in the system: ' system]);
    else
        setSubResultStatusText(ft,['The following lines, input ports, or ' ...
            'output ports are not properly connected in the subsystem: ' system]);
    end
    % Specify recommended action to fix the warning

```

```

        setRecAction(ft,'Connect the specified blocks.');
```

% Create a list of handles to problem objects

```

        setListObj(ft,modelObj);
        ResultStatus = false;
    end
    % Pass the list template object to the Model Advisor
    ResultDescription{end+1} = ft;
    % Set overall check status
    mdladvObj.setCheckResultStatus(ResultStatus);
```

Model Advisor Code Example: Check With Subchecks and Actions

Here is an example of a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting, and the second reviews the **Conditional input branch execution** optimization setting.

A check with subchecks includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.
- Recommended actions to take when the subcheck does not pass.
- A line between the subcheck results.

```

% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
function ResultDescription = OptmizationSettingCallback(system)
% Initialize variables
system =getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
```

```
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};
system = bdroot(system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,['Check model configuration for optimization settings that'...
    'can impact safety.']);

% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction setting');
setInformation(ft1,'Check whether the ''Block reduction'' check box is cleared.');
```

% Add See Also section with references to applicable standards

```
docLinks{1} = [['Reference DO-178B Section 6.3.4e - Source code ' ...
    'is traceable to low-level requirements']];
```

% Review 'Block reduction' optimization

```
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is cleared.');
```

ResultStatus = true;

```
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
```

setSubResultStatusText(ft1,'The ''Block reduction'' check box is selected.');

```
setRecAction(ft1,['Clear the ''Optimization > Block reduction'' ...
    'check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
```

```

setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check whether the ''Conditional input branch execution''...
    ' check box is cleared.'])
% Add See Also section and references to applicable standards
docLinks{1} = {'Reference DO-178B Section 6.4.4.2 - Test coverage ' ...
    'of software structure is achieved'}];
setRefLink(ft2,docLinks);

% Last subcheck, suppress line
setSubBar(ft2,false);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution'' ...
        ' check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution''...
        ' check box is selected.']);
    setRecAction(ft2,['Clear the ''Optimization > Conditional input branch ' ...
        'execution'' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);

```

Detailed Check Callback Function

Use the detailed check callback function to return and organize results as strings in a layered, hierarchical fashion. The function provides two output arguments so you can associate text descriptions with one or more paragraphs of detailed information. The keyword for the detailed callback function is

StyleTwo. The check definition requires this keyword (see “Defining Custom Checks” on page 7-11).

The detailed callback function takes the following arguments.

Argument	I/O Type	Description
system	Input	Path to the model or system analyzed by the Model Advisor.
ResultDescription	Output	Cell array of MATLAB strings that supports Model Advisor Formatting API calls or embedded HTML tags for text formatting. The Model Advisor concatenates the ResultDescription string with the corresponding array of ResultDetails strings.
ResultDetails	Output	Cell array of cell arrays, each of which contains one or more strings.

Note The ResultDetails cell array must be the same length as the ResultDescription cell array.

Here is an example of a detailed check callback function that checks optimization settings for simulation and code generation.

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

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

% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '...
'optimization settings:']);
```



```

if strcmp(get_param(model,'BlockReduction'),'off');
    ResultDetails{end+1} = {ModelAdvisor.Text(['It is recommended to '...
        'turn on Block reduction optimization option.',{'italic'}])};
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
else
    ResultDetails{end+1} = {ModelAdvisor.Text('Passed',{'pass'})};
end

% Check code generation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check code generation '...
    'optimization settings:']);
ResultDetails{end+1} = {};
if strcmp(get_param(model,'LocalBlockOutputs'),'off');
    ResultDetails{end}{end+1} = ModelAdvisor.Text(['It is recommended to'...
        ' turn on Enable local block outputs option.',{'italic'}]);
    ResultDetails{end}{end+1} = ModelAdvisor.LineBreak;
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if strcmp(get_param(model,'BufferReuse'),'off');
    ResultDetails{end}{end+1} = ModelAdvisor.Text(['It is recommended to'...
        ' turn on Reuse block outputs option.',{'italic'}]);
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if isempty(ResultDetails{end})
    ResultDetails{end}{end+1} = ModelAdvisor.Text('Passed',{'pass'});
end
end

```

Check Callback Function with Hyperlinked Results

This callback function automatically displays hyperlinks for every object returned by the check so that you can easily locate problem areas in your model or subsystem. The keyword for this type of callback function is `StyleThree`. The check definition requires this keyword (see “Defining Custom Checks” on page 7-11).

This callback function takes the following arguments.

Argument	I/O Type	Description
system	Input	Path to the model or system analyzed by the Model Advisor.
ResultDescription	Output	Cell array of MATLAB strings that supports the Model Advisor Formatting API calls or embedded HTML tags for text formatting.
ResultDetails	Output	Cell array of cell arrays, each of which contains one or more Simulink objects such as blocks, ports, lines, and Stateflow charts. The objects must be in the form of a handle or Simulink path.

Note The `ResultDetails` cell array must be the same length as the `ResultDescription` cell array.

The Model Advisor automatically concatenates each string from `ResultDescription` with the corresponding array of objects from `ResultDetails`. The Model Advisor displays the contents of `ResultDetails` as a set of hyperlinks, one for each object returned in the cell arrays. When you click a hyperlink, the Model Advisor displays the target object highlighted in your Simulink model.

The following is an example of a check callback function with hyperlinked results. This example checks a model for consistent use of font type and font size in its blocks. It also contains input parameters, actions, and a call to the Model Advisor Result Explorer, which are described in later sections.

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

    mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
    mdladvObj.setCheckResultStatus(true);
    needEnableAction = false;
```

```

% get input parameters
inputParams = mdladvObj.getInputParameters;
skipFontCheck = inputParams{1}.Value;
regularFontSize = inputParams{2}.Value;
regularFontName = inputParams{3}.Value;
if skipFontCheck
    ResultDescription{end+1} = ModelAdvisor.Paragraph('Skipped. ');
    ResultDetails{end+1} = {};
    return
end
regularFontSize = str2double(regularFontSize);
if regularFontSize<1 || regularFontSize>=99
    mdladvObj.setCheckResultStatus(false);
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['Invalid font size. '...
        'Please enter a value between 1 and 99']);
    ResultDetails{end+1} = {};
end

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

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

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

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font other than ' regularFontName ': ']);
    ResultDetails{end+1} = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult, 'object');
    myLVParam.Attributes = {'FontName'}; % name is default property

```

```

        mdladvObj.setListViewParameters({myLVParam});
        needEnableAction = true;
    else
        ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font names '...
            'are identical.']);
        ResultDetails{end+1}      = {};
    end

% find regular font size blocks
regularBlks = find_system(allBlks, 'FontSize', regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font size for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font size other than ' ...
        num2str(regularFontSize) ': ']);
    ResultDetails{end+1}      = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font size blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult, 'object');
    myLVParam.Attributes = {'FontSize'}; % name is default property
    mdladvObj.setListViewParameters...
        ({mdladvObj.getListViewParameters{:}, myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font sizes '...
        'are identical.']);
    ResultDetails{end+1}      = {};
end

mdladvObj.setActionEnable(needEnableAction);
mdladvObj.setCheckErrorSeverity(1);

```

In the Model Advisor, if you run **Example task with input parameter and auto-fix ability** for the `slvnvdemo_mdladv` model, you can view the hyperlinked results. Clicking the first hyperlink, `slvnvdemo_mdladv/Input`, displays the Simulink model with the Input block highlighted.

Action Callback Function

An *action callback function* specifies the actions that the Model Advisor performs on a model or subsystem when the user clicks the action button. You must create one callback function for the action that you want to take.

The action callback function takes the following arguments.

Argument	I/O Type	Description
taskobj	Input	The ModelAdvisor.Task object for the check that includes an action definition.
result	Output	MATLAB string that supports Model Advisor Formatting API calls or embedded HTML tags for text formatting.

Model Advisor Code Example: Action Callback Function

The following is an example of an action callback function that updates all of the blocks in the model with the font specified in the Input Parameter defined in “Model Advisor Code Example: Input Parameter Definition” on page 7-17.

```
% Sample Check 3 Action Callback Function: Check with Subresults and Actions
% Fix optimization settings
function result = modifyOptimizationSetting(taskobj)
% Initialize variables
result = ModelAdvisor.Paragraph();
mdladvObj = taskobj.MAObj;
system = bdroot(mdladvObj.System);

% 'Block reduction' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'BlockReduction'),'off')
    set_param(system,'BlockReduction','off');
    result.addItem(ModelAdvisor.Text( ...
        'Cleared the ''Block reduction'' check box.',{'Pass'}));
    result.addItem(ModelAdvisor.LineBreak);
end

% 'Conditional input branch execution' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
```

```
set_param(system, 'ConditionallyExecuteInputs', 'off');
result.addItem(ModelAdvisor.Text( ...
    'Cleared the ''Conditional input branch execution'' check box.', ...
    {'Pass'}));
end
```

Formatting Model Advisor Results

- “Overview of Displaying Results” on page 7-38
- “Formatting Model Advisor Results” on page 7-38
- “Formatting Text” on page 7-39
- “Formatting Lists” on page 7-40
- “Formatting Tables” on page 7-40
- “Formatting Paragraphs” on page 7-41
- “Model Advisor Code Example: Formatted Output” on page 7-41

Overview of Displaying Results

You can make all of the analysis results of your custom checks appear similar to each other with minimal scripting using the Model Advisor `ModelAdvisor.FormatTemplate` class, as described in `ModelAdvisor.FormatTemplate`. For examples of callback functions using the `ModelAdvisor.FormatTemplate` class, see “Simple Check Callback Function” on page 7-23.

If this format template does not meet your needs, or if you want to format action results, use the Model Advisor Formatting API, as described in the following sections.

Formatting Model Advisor Results

Use the Model Advisor Formatting API to produce formatted outputs in the Model Advisor. The following constructors of the `ModelAdvisor` class provide the ability to format the output. For more information on each constructor and associated methods, in the Constructor column, click the link.

Constructor	Description
<code>ModelAdvisor.Text</code>	Formats element text.
<code>ModelAdvisor.Paragraph</code>	Combines elements into paragraphs.
<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. You can format text in many different ways. The default text formatting is:

- Empty
- Default color (black)
- Unformatted (not bold, italicized, underlined, linked, subscripted, or superscripted)

To change text formatting, use the `ModelAdvisor.Text` constructor. When you want one type of formatting for all text, use this syntax:

```
ModelAdvisor.Text(content, {attributes})
```

When you want multiple types of formatting, you must build the text.

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' to ensure uniform appearance of model.');
```

```
result = [t1, t2, t3, t4, t5];
```

Add ASCII and Extended ASCII characters using the MATLAB `char` command. For more information, see the `ModelAdvisor.Text` class page.

Formatting Lists

You can create two types of lists: numbered and bulleted. The default list formatting is bulleted. Use the `ModelAdvisor.List` constructor to create and format lists (see `ModelAdvisor.List`). You can create lists with indented subsections, formatted as either numbered or bulleted.

```
subList = ModelAdvisor.List();
subList.setType('numbered');
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass', 'bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass', 'bold'}));

topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1', {'keyword', 'bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2', {'keyword', 'bold'}), subList]);
```

Formatting Tables

The default table formatting is:

- Default color (black)
- Left justified
- Bold title, row, and column headings

Change table formatting using the `ModelAdvisor.Table` constructor. The following example code creates a subtable within a table.

```
table1 = ModelAdvisor.Table(1,1);
table2 = ModelAdvisor.Table(2,3);
table2.setHeading('Table 2');
table2.setHeadingAlign('center');
table2.setColHeading(1, 'Header 1');
table2.setColHeading(2, 'Header 2');
table2.setColHeading(3, 'Header 3');
table1.setHeading('Table 1');
table1.setEntry(1,1,table2);
```


Table 1		
Table 2		
Header 1	Header 2	Header 3

Formatting Paragraphs

You must handle paragraphs explicitly because most markup languages do not support line breaks. The default paragraph formatting is:

- Empty
- Default color (black)
- Unformatted, (not bold, italicized, underlined, linked, subscripted, or superscripted)
- Aligned left

If you want to change paragraph formatting, use the `ModelAdvisor.Paragraph` class.

Model Advisor Code Example: Formatted Output

The following is the example from “Simple Check Callback Function” on page 7-23, 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

Creating Custom Configurations by Organizing Checks and Folders

- “Overview of Creating Custom Configurations” on page 8-2
- “Organizing Checks and Folders Using the Model Advisor Configuration Editor” on page 8-4
- “Organizing Checks and Folders Within a Customization File” on page 8-12
- “Verifying and Using Custom Configurations” on page 8-22

Overview of Creating Custom Configurations

In this section...
“About Creating Custom Configurations” on page 8-2
“Creating Custom Configurations Workflow” on page 8-2
“Using the Model Advisor Configuration Editor Versus Customization File” on page 8-3

About Creating Custom Configurations

The Simulink Verification and Validation product allows you to extend the capabilities of the Model Advisor. Using Model Advisor APIs and the Model Advisor Configuration Editor, you can:

- Customize the behavior of the Model Advisor by defining your own custom checks, and writing your own callback functions.
- Organize checks and folders to create custom Model Advisor configurations.
- Create multiple custom configurations that you use for different projects or modeling guidelines, and switch between these configurations in the Model Advisor.

Creating Custom Configurations Workflow

When you create custom configurations, you:

- 1** Optionally author custom checks, as described in Chapter 7, “Authoring Custom Checks”.
- 2** Identify which MathWorks checks you want to include in your custom Model Advisor configuration.
- 3** Organize checks and folders to create custom configurations. You can create custom configurations either using the Model Advisor Configuration Editor (see “Organizing Checks and Folders Using the Model Advisor Configuration Editor” on page 8-4), or within a customization file (see “Organizing Checks and Folders Within a Customization File” on page 8-12).

- 4 Verify the custom configuration, as described in “Verifying and Using Custom Configurations” on page 8-22.

Using the Model Advisor Configuration Editor Versus Customization File

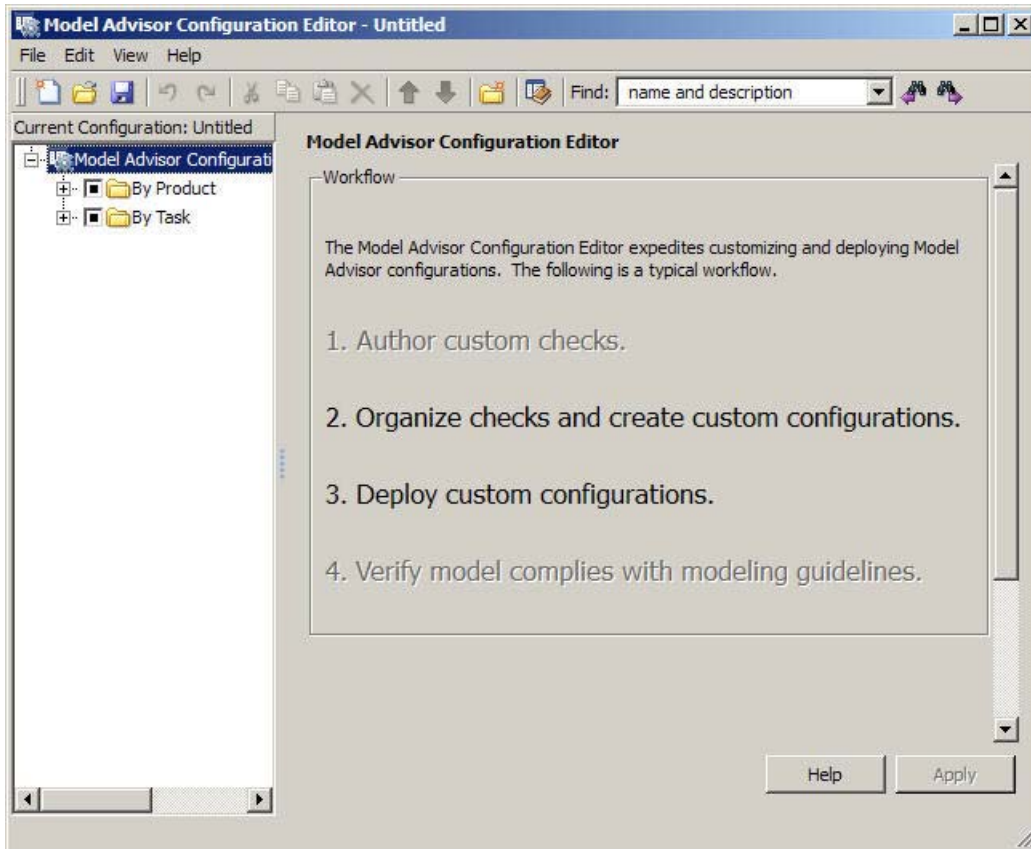
The Model Advisor Configuration Editor is a GUI that expedites creating and deploying custom configurations. While you can organize Model Advisor configurations in a customization file, The MathWorks recommends that you create custom configurations using the Model Advisor Configuration Editor. For more details, see “Organizing Checks and Folders Using the Model Advisor Configuration Editor” on page 8-4.

Organizing Checks and Folders Using the Model Advisor Configuration Editor

In this section...
“Overview of the Model Advisor Configuration Editor” on page 8-4
“Starting the Model Advisor Configuration Editor” on page 8-9
“How To Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-10

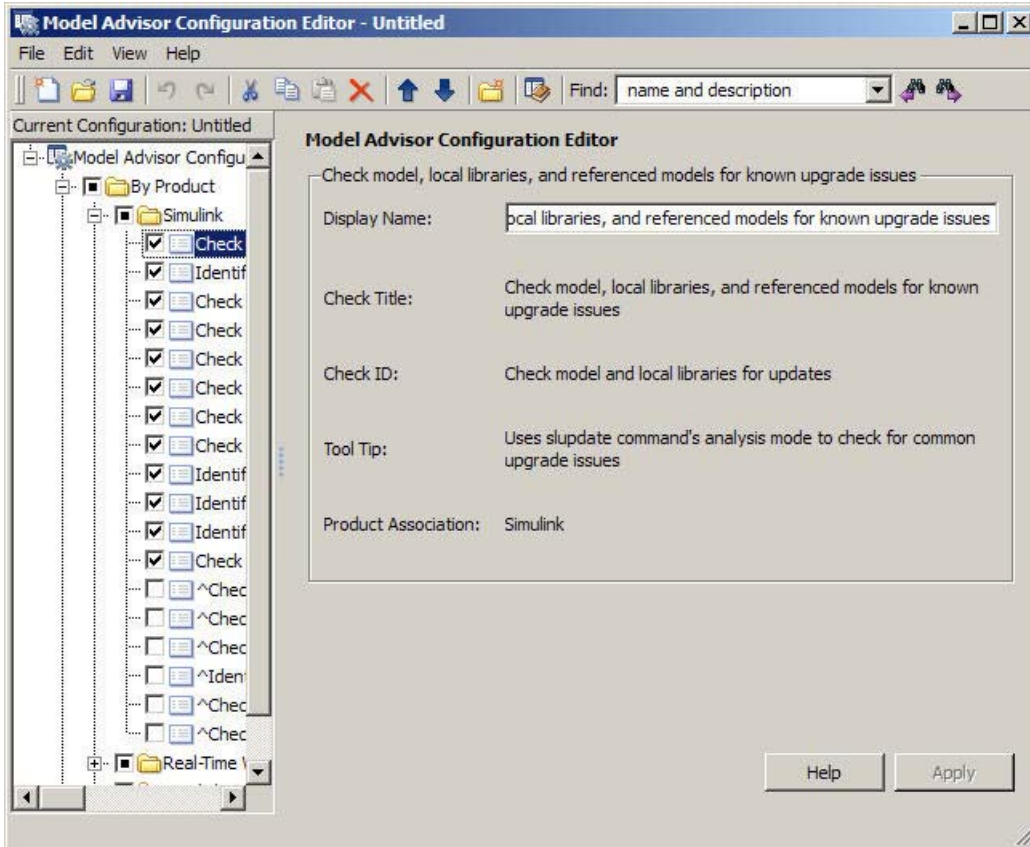
Overview of the Model Advisor Configuration Editor

When you start the Model Advisor Configuration Editor, two windows open; the Model Advisor Configuration Editor and the Model Advisor Check Browser. The Configuration Editor window consists of two panes: the Model Advisor Configuration Editor hierarchy and the Workflow. The Model Advisor Configuration Editor hierarchy lists the checks and folders in the current configuration. The Workflow on the right shows the common workflow you use to create a custom configuration.



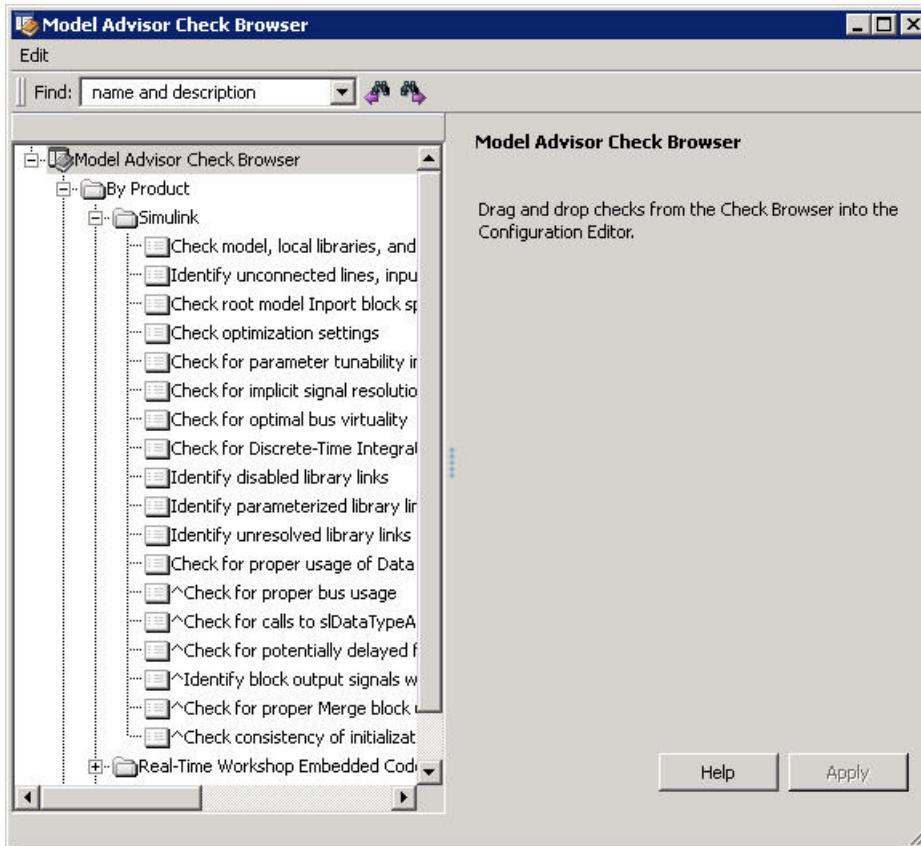
Model Advisor Configuration Editor

When you select a folder or check in the Model Advisor Configuration Editor hierarchy, the Workflow pane changes to display information about the check or folder. You can change the display name of the check or folder in this pane.



The Model Advisor Check Browser window includes a read-only list of available checks. If you delete a check in the Model Advisor Configuration Editor, you can retrieve a copy of it from the Model Advisor Check Browser.

Tip If you use a process callback function in a `sl_customization` file to hide checks and folders, the Model Advisor Configuration Editor and Model Advisor Check Browser do not display the hidden checks and folders. For a complete list of checks and folders, remove process callback functions and update the Simulink environment (see “Updating the Environment to Include Your `sl_customization` File” on page 8-22).



Model Advisor Check Browser

Using the Model Advisor Configuration Editor, you can perform the following actions.

To...	Select...
Create new configurations	File > New
Find checks and folders in the Model Advisor Check Browser	View > Check Browser

To...	Select...
Add checks and folders to the configuration	Edit > Copy Edit > Paste Edit > New folder The check or folder and drag and drop
Remove checks and folders from the configuration	Edit > Delete Edit > Cut
Reorder checks and folders	Edit > Move up Edit > Move down The check or folder and drag and drop
Rename checks and folders <hr/> Note The MathWorks folder display names are restricted. When you rename a folder, you cannot use the restricted display names.	The check or folder and edit Display Name in right pane.
Allow or gray out the check box control for checks and folders <hr/> Tip This capability is equivalent to enabling checks, described in “Displaying and Enabling Checks” on page 7-13.	Edit > Enable Edit > Disable
Save the configuration as a MAT file for use and distribution	File > Save File > Save As
Set the configuration so it opens by default in the Model Advisor	File > Set Current Configuration as Default
Restore the MathWorks default configuration	File > Restore Default Configuration
Load and edit saved configurations	File > Open

Starting the Model Advisor Configuration Editor

Note

- Before starting the Model Advisor Configuration Editor, ensure that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.
- The Model Advisor Configuration Editor uses the Simulink project (slprj) folder (for details about storing reports and other relevant information, see “Model Reference Simulation Targets”) in the current folder. If this folder does not exist in the current folder, the Model Advisor Configuration Editor creates it.

- 1 To include custom checks in the new Model Advisor configuration, update the Simulink environment to include your `sl_customization.m` file. For more information, see “Updating the Environment to Include Your `sl_customization` File” on page 8-22.
- 2 Start the Model Advisor Configuration Editor.

To start the Model Advisor Configuration Editor...	Do this:
Programmatically	At the MATLAB command line, enter <code>Simulink.ModelAdvisor.openConfigUI</code> . For more information, see the <code>Simulink.ModelAdvisor</code> function reference page.
From the Model Advisor	<ol style="list-style-type: none"> 1 Start the Model Advisor. 2 Select File > Open Configuration Editor.

The Model Advisor Configuration Editor and Model Advisor Check Browser windows open.

- 3 Optionally, to edit an existing configuration in the Model Advisor Configuration Editor window:
 - a Select **File > Open**.
 - b In the Open dialog box, navigate to the configuration file that you want to edit.
 - c Click **Open**.

How To Organize Checks and Folders Using the Model Advisor Configuration Editor

The following tutorial steps you through creating a custom configuration.

- 1 Open the Model Advisor Configuration Editor at the MATLAB command line by entering `Simulink.ModelAdvisor.openConfigUI` . For more options, see “Starting the Model Advisor Configuration Editor” on page 8-9.
- 2 In the Model Advisor Configuration Editor, in the left pane, delete the **By Product** and **By Task** folders, to start with a blank configuration.
- 3 Select the root node which is labeled Model Advisor Configuration Editor.
- 4 In the toolbar, click the **New Folder** button to create a folder.
- 5 In the left pane, select the new folder.
- 6 In the right pane, edit **Display Name** to rename the folder. For the purposes of this tutorial, rename the folder to **Review Optimizations**.
- 7 In the Model Advisor Check Browser window, in the **Find** field, enter optimization to find **Simulink > Check optimization settings**.
- 8 Drag and drop **Check optimization settings** into **Review Optimizations**.
- 9 In the Model Advisor Check Browser window, find **Simulink Verification and Validation > DO-178B Checks > Check safety-related optimization settings**.
- 10 Drag and drop **Check safety-related optimization settings** into **Review Optimizations**.

- 11** In the Model Advisor Configuration Editor window, expand **Review Optimizations**.
- 12** Rename **Check optimization settings** to **Check Simulink optimization settings**.
- 13** Select **File > Save As** to save the configuration.
- 14** Name the configuration `optimization_configuration.mat`.
- 15** Close the Model Advisor Configuration Editor window.

Organizing Checks and Folders Within a Customization File

In this section...

“Customization File Overview” on page 8-12

“Register Tasks and Folders” on page 8-13

“Defining Custom Tasks” on page 8-15

“Defining Custom Folders” on page 8-18

“Demo and Code Example” on page 8-20

Note While you can organize checks and folders within a customization file, The MathWorks recommends that you use the Model Advisor Configuration Editor. For more information, see “Using the Model Advisor Configuration Editor Versus Customization File” on page 8-3.

Customization File Overview

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

Function	Description	Required or Optional
<code>sl_customization()</code>	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at startup (see “Register Checks and Process Callbacks” on page 7-6).	Required for all customizations to the Model Advisor.
One or more check definitions	Defines all custom checks (see “Defining Custom Checks” on page 7-11).	Required for custom checks and to add custom checks to the By Product folder.

Function	Description	Required or Optional
One or more task definitions	Defines all custom tasks (see “Defining Custom Tasks” on page 8-15).	Required to add custom checks to the Model Advisor, except when adding the checks to the By Product folder. Write one task for each check that you add to the Model Advisor.
One or more groups	Defines all custom groups (see “Defining Custom Tasks” on page 8-15).	Required to add custom tasks to new folders in the Model Advisor, except when adding a new subfolder to the By Product folder. Write one group definition for each new folder.
One process callback function	Specifies actions that Simulink performs at startup and post-execution time (see “Defining Startup and Post-Execution Actions Using Process Callback Functions” on page 7-8).	Optional.

Register Tasks and Folders

- “Create `sl_customization` Function” on page 8-13
- “Registering Tasks and Folders” on page 8-14

Create `sl_customization` Function

To add tasks and folders to the Model Advisor, create the `sl_customization.m` file on your MATLAB path. Then create the `sl_customization()` function in the `sl_customization.m` file on your MATLAB path.

Tip

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

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

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

Registering Tasks and Folders

The customization manager provides the following methods for registering custom tasks and folders:

- `addModelAdvisorTaskFcn (@factorygroupDefinitionFcn)`

Registers the tasks that you define in *factorygroupDefinitionFcn* to the **By Task** folder of the Model Advisor.

The *factorygroupDefinitionFcn* argument is a handle to the function that defines the checks to add to the Model Advisor as instances of the `ModelAdvisor.FactoryGroup` class (see “Defining Custom Tasks” on page 8-15).

- `addModelAdvisorTaskAdvisorFcn (@taskDefinitionFcn)`

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

The *taskDefinitionFcn* argument is a handle to the function that defines all custom tasks and folders. Simulink adds the checks and folders to the Model Advisor as instances of the `ModelAdvisor.Task` or `ModelAdvisor.Group` classes (see “Defining Custom Tasks” on page 8-15).

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

Model Advisor Code Example: Registering Custom Tasks and Folders.

The following code example registers custom tasks and folders:

```
function sl_customization(cm)

% register custom factory group
cm.addModelAdvisorTaskFcn(@defineModelAdvisorTasks);

% register custom tasks.
cm.addModelAdvisorTaskAdvisorFcn(@defineTaskAdvisor);
```

Note If you add custom checks and process callbacks within the `sl_customization.m` file, include methods for registering the checks and process callbacks in the `sl_customization` function. For more information, see “Register Checks and Process Callbacks” on page 7-6.

Defining Custom Tasks

- “Adding a Check to Custom or Multiple Folders Using Tasks” on page 8-16
- “Creating Custom Tasks Using MathWorks Checks” on page 8-16
- “Displaying and Enabling Tasks” on page 8-17
- “Defining Where Tasks Appear” on page 8-17
- “Model Advisor Code Example: Task Definition Function” on page 8-17

Adding a Check to Custom or Multiple Folders Using Tasks

You can use custom tasks for adding checks to the Model Advisor, either in multiple folders or in a single, custom folder. You define custom tasks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Task` class. Define one instance of this class for each custom task that you want to add to the Model Advisor. Then register the custom task, as described in “Register Tasks and Folders” on page 8-13. The following sections describe how to define custom tasks.

To add a check to multiple folders or a single, custom folder:

- 1 Create a check using the `ModelAdvisor.Check` class, as described in “Defining Custom Checks” on page 7-11.
- 2 Register a task wrapper for the check, as described in “Register Tasks and Folders” on page 8-13.
- 3 If you want to add the check to folders that are not already present, register and create the folders using the `ModelAdvisor.Group` class.
- 4 Add a check to the task using the `ModelAdvisor.Task.setCheck` method.
- 5 Add the task to each folder using the `ModelAdvisor.Group.addTask` method and the task ID.

Creating Custom Tasks Using MathWorks Checks

You can add MathWorks checks to your custom folders by defining the checks as custom tasks. When you add the checks as custom tasks, you identify checks by the check ID.

To find MathWorks check IDs:

- 1 In the Model Advisor, select **View > Source Tab**.
- 2 Navigate to the folder that contains the MathWorks check.
- 3 In the right pane, click **Source**. The Model Advisor displays the **Title**, **TitleID**, and **Source** information for each check in the folder.
- 4 Select and copy the **TitleID** of the check that you want to add as a task.

Displaying and Enabling Tasks

The `Visible`, `Enable`, and `Value` properties interact the same way for tasks as they do for checks (see “Displaying and Enabling Checks” on page 7-13).

Defining Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor using the following guidelines:

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class. See “Defining Custom Folders” on page 8-18.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See “Defining Custom Folders” on page 8-18.

Model Advisor Code Example: Task Definition Function

The following is an example of a task definition function. This function defines three tasks. The tasks are derived from the checks defined in “Model Advisor Code Example: Check Definition Function” on page 7-15.

For an example of placing these tasks into a custom group, see “Model Advisor Code Example: Group Definition” on page 8-19.

```
% Defines Model Advisor tasks and a custom folder
% Add checks to a custom folder using task definitions
function defineTaskAdvisor
mdladvRoot = ModelAdvisor.Root;

% Define task that uses Sample Check 1: Informational check
MAT1 = ModelAdvisor.Task('mathworks.example.task.configManagement');
MAT1.DisplayName = 'Informational check for model configuration management';
MAT1.Description = 'Display model configuration and checksum information.';
setCheck(MAT1, 'mathworks.example.configManagement');
mdladvRoot.register(MAT1);

% Define task that uses Sample Check 2: Basic Check with Pass/Fail Status
MAT2 = ModelAdvisor.Task('mathworks.example.task.unconnectedObjects');
MAT2.DisplayName = 'Check for unconnected objects';
```

```
setCheck(MAT2, 'mathworks.example.unconnectedObjects');
MAT2.Description = ['Identify unconnected lines, input ports, and output ' ...
                  'ports in the model or subsystem.'];

mdladvRoot.register(MAT2);

% Define task that uses Sample Check 3: Check with Subresults and Actions
MAT3 = ModelAdvisor.Task('mathworks.example.task.optimizationSettings');
MAT3.DisplayName = 'Check safety-related optimization settings';
MAT3.Description = ['Check model configuration for optimization ' ...
                  'settings that can impact safety.'];
MAT3.setCheck('mathworks.example.optimizationSettings');
mdladvRoot.register(MAT3);

% Custom folder definition
MAG = ModelAdvisor.Group('mathworks.example.ExampleGroup');
MAG.DisplayName = 'My Group';
% Add tasks to My Group folder
addTask(MAG, MAT1);
addTask(MAG, MAT2);
addTask(MAG, MAT3);
% Add My Group folder to the Model Advisor under 'Model Advisor' (root)
mdladvRoot.publish(MAG);
```

Defining Custom Folders

- “About Custom Folders” on page 8-18
- “Adding Custom Folders” on page 8-19
- “Defining Where Custom Folders Appear” on page 8-19
- “Model Advisor Code Example: Group Definition” on page 8-19

About Custom Folders

Use folders to group checks in the Model Advisor by functionality or usage. You define custom folders in:

- A factory group definition function that specifies the properties of each instance of the `ModelAdvisor.FactoryGroup` class.

- A task definition function that specifies the properties of each instance of the `ModelAdvisor.Group` class. For more information about task definition functions, see “Adding a Check to Custom or Multiple Folders Using Tasks” on page 8-16.

Define one instance of the group classes for each folder that you want to add to the Model Advisor. Then register the custom folder, as described in “Register Tasks and Folders” on page 8-13. The following sections describe how to define custom groups.

Adding Custom Folders

To add a custom folder:

- 1 Create the folder using the `ModelAdvisor.Group` or `ModelAdvisor.FactoryGroup` classes.
- 2 Add the folder to the Model Advisor, as described in “Defining Custom Folders” on page 8-18.

Defining Where Custom Folders Appear

You can specify the location of custom folders within the Model Advisor using the following guidelines:

- To define a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To define a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Note To define a new folder in the **By Product** folder, use the `ModelAdvisor.Root.publish` method within a custom check. For more information, see “Defining Where Custom Checks Appear” on page 7-14.

Model Advisor Code Example: Group Definition

The following is an example of a group definition. The definition places the tasks defined in “Model Advisor Code Example: Task Definition Function” on

page 8-17 inside a folder called **My Group** under the **Model Advisor** root. The task definition function includes this group definition.

```
% Custom folder definition
MAG = ModelAdvisor.Group('mathworks.example.ExampleGroup');
MAG.DisplayName='My Group';
% Add tasks to My Group folder
MAG.addTask(MAT1);
MAG.addTask(MAT2);
MAG.addTask(MAT3);
% Add My Group folder to the Model Advisor under 'Model Advisor' (root)
mdladvRoot.publish(MAG);
```

The following is an example of a factory group definition function. The definition places the checks defined in “Model Advisor Code Example: Check Definition Function” on page 7-15 into a folder called **Demo Factory Group** inside of the **By Task** folder.

```
function defineModelAdvisorTasks
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='Demo Factory Group';
rec.Description='Demo Factory Group';
rec.addCheck('mathworks.example.configManagement');
rec.addCheck('mathworks.example.unconnectedObjects');
rec.addCheck('mathworks.example.optimizationSettings');
mdladvRoot.publish(rec); % publish inside By Task
```

Demo and Code Example

The Simulink Verification and Validation software provides a demo that shows how to customize the Model Advisor by adding:

- Custom checks
- Check input parameters
- Check actions
- Check list views to call the Model Advisor Result Explorer

- Custom tasks to include the custom checks in the Model Advisor
- Custom folders for grouping the checks
- A process callback function

The demo also provides the source code of the `sl_customization.m` file that executes the customizations.

To run the demo:

- 1** At the MATLAB command line, type `slvndemo_md1adv`.
- 2** Follow the instructions in the model.

Verifying and Using Custom Configurations

In this section...
“Updating the Environment to Include Your sl_customization File” on page 8-22
“Verifying Custom Configurations” on page 8-22

Updating the Environment to Include Your sl_customization File

When you start Simulink, it reads customization (`sl_customization.m`) files. If you change the contents of your customization file, update your environment by performing these tasks:

- 1 If you previously started the Model Advisor:
 - a Close the model from which you started the Model Advisor
 - b Clear the data associated with the previous Model Advisor session by removing the `slprj` folder from your working folder.

- 2 At the MATLAB command line, enter:

```
sl_refresh_customizations
```

- 3 Open your model.
- 4 Start the Model Advisor.

Verifying Custom Configurations

To verify a custom configuration:

- 1 If you created custom checks, or created the custom configuration using the `sl_customization` method, update the Simulink environment. For more information, see “Updating the Environment to Include Your sl_customization File” on page 8-22.
- 2 Open a model.
- 3 From the model window, start the Model Advisor.

- 4** Select **File > Load Configuration**. If you see a warning that the Model Advisor report corresponds to a different configuration, click **Load** to continue.
- 5** In the Open dialog box, navigate to and select your custom configuration. For example, if you created the custom configuration in “How To Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-10, select `optimization_configuration.mat`.
- 6** When the Model Advisor reopens, verify that the new configuration contains the appropriate folders and checks. For example, the **Review Optimizations** folder and the **Check Simulink optimization settings** and **Check safety-related optimization settings** checks.
- 7** Optionally, run the checks.

Deploying Custom Configurations

- “Overview of Deploying Custom Configurations” on page 9-2
- “How to Deploy Custom Configurations” on page 9-3
- “Loading and Setting the Default Configuration” on page 9-4

Overview of Deploying Custom Configurations

In this section...
“About Deploying Custom Configurations” on page 9-2
“Deploying Custom Configurations Workflow” on page 9-2

About Deploying Custom Configurations

When you create a custom configuration, often you *deploy* the custom configuration to your development group. Deploying the custom configuration allows your development group to review models using the same checks.

After you create a custom configuration, you can use it in the Model Advisor, or deploy the configuration to your users. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

Deploying Custom Configurations Workflow

When you deploy custom configurations, you:

- 1 Optionally author custom checks, as described in Chapter 7, “Authoring Custom Checks”.
- 2 Organize checks and folders to create custom configurations, as described in Chapter 8, “Creating Custom Configurations by Organizing Checks and Folders”.
- 3 Deploy the custom configuration to your users, as described in “How to Deploy Custom Configurations” on page 9-3.

How to Deploy Custom Configurations

To deploy a custom configuration:

- 1 Determine which files to distribute. You might need to distribute more than one file.

If You...	Using the...	Distribute...
Created custom checks	Customization file	<ul style="list-style-type: none"> • <code>sl_customization.m</code> • Files containing check and action callback functions (if separate)
Organized checks and folders	Model Advisor Configuration Editor	Configuration MAT file
	Customization file	<code>sl_customization.m</code>

- 2 Distribute the files and tell the user to include these files on the MATLAB path.
- 3 Instruct the user to load the custom configuration. For more details, see “Loading and Setting the Default Configuration” on page 9-4.

Loading and Setting the Default Configuration

When you use the Model Advisor, you can load any configuration. Once you load a configuration, you can set it to be the configuration that the Model Advisor uses every time you open the Model Advisor.

- 1 Open the Model Advisor.
- 2 Select **File > Load Configuration**.
- 3 In the Open dialog box, navigate to and select the configuration file that you want to edit.
- 4 Click **Open**.

Simulink reloads the Model Advisor using the new configuration.

- 5 Optionally, set the current configuration as the default when the Model Advisor opens by selecting **File > Set Current Configuration as Default**.

Tip You can restore the MathWorks default configuration by selecting **File > Restore Default Configuration**.

Function Reference

Requirements Management
Interface (p. 10-2)

Model Coverage (p. 10-3)

Model Advisor Customization API
(p. 10-5)

Model Advisor Result Template API
(p. 10-7)

Model Advisor Formatting API
(p. 10-8)

Access Requirements Management
Interface

Configure and execute model
coverage tests; store and report test
results

Customize the Model Advisor tree;
create new checks and folders

Template for formatting Model
Advisor results

Format Model Advisor outputs

Requirements Management Interface

rmi	Interact programmatically with Requirements Management Interface
rmidocrename	Update model requirements document paths and file names
rminav	Start Requirements Management Interface

Model Coverage

<code>add (cv.cvtestgroup)</code>	Add <code>cvtest</code> objects
<code>allNames (cv.cvdatagroup)</code>	Get names of all models associated with <code>cvdata</code> objects in <code>cv.cvdatagroup</code>
<code>allNames (cv.cvtestgroup)</code>	Get names of all models associated with <code>cvtest</code> objects in <code>cvtestgroup</code>
<code>conditioninfo</code>	Collect condition coverage information for model object
<code>cv.cvdatagroup</code>	Create collection of <code>cvdata</code> objects for model reference hierarchy
<code>cv.cvtestgroup</code>	Create collection of <code>cvtest</code> objects for model reference hierarchy
<code>cvexit</code>	Exit model coverage environment
<code>cvhtml</code>	Produce HTML report from model coverage objects
<code>cvload</code>	Load coverage tests and stored results into memory
<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>get (cv.cvdatagroup)</code>	Get <code>cvdata</code> object

<code>get (cv.cvtestgroup)</code>	Get <code>cvtest</code> objects
<code>getAll (cv.cvdatagroup)</code>	Get all <code>cvdata</code> objects
<code>getCoverageInfo</code>	Coverage information for Simulink Design Verifier blocks
<code>mcdeinfo</code>	Collect modified condition/decision coverage information for model object
<code>sigrangeinfo</code>	Collect signal range coverage information for model object
<code>tableinfo</code>	Display lookup table coverage information for model object

Model Advisor Customization API

addCheck (ModelAdvisor.FactoryGroup)	Add check to folder
addGroup (ModelAdvisor.Group)	Add subfolder to folder
addTask (ModelAdvisor.Group)	Add task to folder
getID (ModelAdvisor.Check)	Return check identifier
ModelAdvisor.Action	Add actions to custom checks
ModelAdvisor.Check	Create custom checks
ModelAdvisor.FactoryGroup	Define subfolder in By Task folder
ModelAdvisor.Group	Define custom folder
ModelAdvisor.InputParameter	Add input parameters to custom checks
ModelAdvisor.ListViewParameter	Add list view parameters to custom checks
ModelAdvisor.Root	Identify root node
ModelAdvisor.Task	Define custom tasks
publish (ModelAdvisor.Root)	Publish object in Model Advisor root
register (ModelAdvisor.Root)	Register object in Model Advisor root
setAction (ModelAdvisor.Check)	Specify action for check
setCallbackFcn (ModelAdvisor.Action)	Specify action callback function
setCallbackFcn (ModelAdvisor.Check)	Specify callback function for check
setCheck (ModelAdvisor.Task)	Specify check used in task
setColSpan (ModelAdvisor.InputParameter)	Specify number of columns for input parameter
setInputParameters (ModelAdvisor.Check)	Specify input parameters for check

setInputParametersLayoutGrid
(ModelAdvisor.Check)

Specify layout grid for input parameters

setRowSpan
(ModelAdvisor.InputParameter)

Specify rows for input parameter

Model Advisor Result Template API

<code>addRow</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add row to table
<code>ModelAdvisor.FormatTemplate</code>	Construct template object for formatting Model Advisor analysis results
<code>setCheckText</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add description of check to result
<code>setColTitles</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add column titles to table
<code>setInformation</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add description of subcheck to result
<code>setListObj</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add list of hyperlinks to model objects
<code>setRecAction</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add Recommended Action section and text
<code>setRefLink</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add See Also section and links
<code>setSubBar</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add line between subcheck results
<code>setSubResultStatus</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add status to check or subcheck result
<code>setSubResultStatusText</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add text below status in result
<code>setSubTitle</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add title for subcheck in result
<code>setTableInfo</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add data to table
<code>setTableTitle</code> (<code>ModelAdvisor.FormatTemplate</code>)	Add title to table

Model Advisor Formatting API

<code>addItem (ModelAdvisor.List)</code>	Add item to list
<code>addItem (ModelAdvisor.Paragraph)</code>	Add item to paragraph
<code>getEntry (ModelAdvisor.Table)</code>	Get table cell contents
<code>ModelAdvisor.Image</code>	Include image in Model Advisor output
<code>ModelAdvisor.LineBreak</code>	Insert line break
<code>ModelAdvisor.List</code>	Create list class
<code>ModelAdvisor.Paragraph</code>	Create and format paragraph
<code>ModelAdvisor.Table</code>	Create table
<code>ModelAdvisor.Text</code>	Create Model Advisor text output
<code>setAlign (ModelAdvisor.Paragraph)</code>	Specify paragraph alignment
<code>setBold (ModelAdvisor.Text)</code>	Specify bold text
<code>setColHeading (ModelAdvisor.Table)</code>	Specify table column title
<code>setColHeadingAlign (ModelAdvisor.Table)</code>	Specify column title alignment
<code>setColor (ModelAdvisor.Text)</code>	Specify text color
<code>setColWidth (ModelAdvisor.Table)</code>	Specify column widths
<code>setEntry (ModelAdvisor.Table)</code>	Add cell to table
<code>setEntryAlign (ModelAdvisor.Table)</code>	Specify table cell alignment
<code>setHeading (ModelAdvisor.Table)</code>	Specify table title
<code>setHeadingAlign (ModelAdvisor.Table)</code>	Specify table title alignment
<code>setHyperlink (ModelAdvisor.Image)</code>	Specify hyperlink location
<code>setHyperlink (ModelAdvisor.Text)</code>	Specify hyperlinked text
<code>setImageSource (ModelAdvisor.Image)</code>	Specify image location
<code>setItalic (ModelAdvisor.Text)</code>	Italicize text

<code>setRetainSpaceReturn</code> (ModelAdvisor.Text)	Retain spacing and returns in text
<code>setRowHeading</code> (ModelAdvisor.Table)	Specify table row title
<code>setRowHeadingAlign</code> (ModelAdvisor.Table)	Specify table row title alignment
<code>setSubscript</code> (ModelAdvisor.Text)	Specify subscripted text
<code>setSuperscript</code> (ModelAdvisor.Text)	Specify superscripted text
<code>setType</code> (ModelAdvisor.List)	Specify list type
<code>setUnderlined</code> (ModelAdvisor.Text)	Underline text

Class Reference

- “Model Coverage” on page 11-2
- “Model Advisor Customization API” on page 11-3
- “Model Advisor Result Template API” on page 11-4
- “Model Advisor Formatting API” on page 11-5

Model Coverage

cv.cvdatagroup

Collection of cvdata objects

cv.cvtestgroup

Collection of cvtest objects

Model Advisor Customization API

ModelAdvisor.Action	Add actions to custom checks
ModelAdvisor.Check	Create custom checks
ModelAdvisor.FactoryGroup	Define subfolder in By Task folder
ModelAdvisor.Group	Define custom folder
ModelAdvisor.InputParameter	Add input parameters to custom checks
ModelAdvisor.ListViewParameter	Add list view parameters to custom checks
ModelAdvisor.Root	Identify root node
ModelAdvisor.Task	Define custom tasks

Model Advisor Result Template API

ModelAdvisor.FormatTemplate

Template for formatting Model
Advisor analysis results

Model Advisor Formatting API

ModelAdvisor.Image	Include image in Model Advisor output
ModelAdvisor.LineBreak	Insert line break
ModelAdvisor.List	Create list class
ModelAdvisor.Paragraph	Create and format paragraph
ModelAdvisor.Table	Create table
ModelAdvisor.Text	Create Model Advisor text output

Alphabetical List

cv.cvtestgroup.add

Purpose Add cvtest objects

Syntax `add(cvtg, cvto1, cvto2, ...)`

Description `add(cvtg, cvto1, cvto2, ...)` adds the cvtest objects specified by the strings `cvto1`, `cvto2`, etc. to `cvtg`, which is an instantiation of the `cv.cvtestgroup` class.

Example Create two cvtest objects and add them to a newly created `cv.cvtestgroup` object:

```
cvto1 = cvtest;
cvto2 = cvtest;
cvtg = cv.cvtestgroup;
add(cvtg, cvto1, cvto2);
```


ModelAdvisor.FactoryGroup.addCheck

Purpose Add check to folder

Syntax addCheck(fg_obj, check_ID)

Description addCheck(fg_obj, check_ID) adds checks, identified by check_ID, to the folder specified by fg_obj, which is an instantiation of the ModelAdvisor.FactoryGroup class.

Examples Add three checks to rec:

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
.
.
.
addCheck(rec, 'com.mathworks.sample.Check1');
addCheck(rec, 'com.mathworks.sample.Check2');
addCheck(rec, 'com.mathworks.sample.Check3');
```

ModelAdvisor.Group.addGroup

Purpose Add subfolder to folder

Syntax `addGroup(group_obj, child_obj)`

Description `addGroup(group_obj, child_obj)` adds a new subfolder, identified by `child_obj`, to the folder specified by `group_obj`, which is an instantiation of the `ModelAdvisor.Group` class.

Examples Add three checks to rec:

```
group_obj = ModelAdvisor.Group('com.mathworks.sample.group');  
.  
.  
.  
addGroup(group_obj, 'com.mathworks.sample.subgroup1');  
addGroup(group_obj, 'com.mathworks.sample.subgroup2');  
addGroup(group_obj, 'com.mathworks.sample.subgroup3');
```

Purpose Add item to list

Syntax `addItem(element)`

Description `addItem(element)` adds items to the list created by the `ModelAdvisor.List` constructor.

Inputs *element* Specifies an element to be added to a list in one of the following:

- Element
- Cell array of elements. When you add a cell array to a list, they form different rows in the list.
- String

Example

```
subList = ModelAdvisor.List();
setType(subList, 'numbered')
addItem(subList, ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
addItem(subList, ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Paragraph.addItem

Purpose Add item to paragraph

Syntax addItem(text, element)

Description addItem(text, element) adds an element to text. element is one of the following:

- String
- Element
- Cell array of elements

Example Add two lines of text:

```
result = ModelAdvisor.Paragraph;  
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.FormatTemplate.addRow

Purpose Add row to table

Syntax `addRow(ft_obj, {item1, item2, ..., itemn})`

Description `addRow(ft_obj, {item1, item2, ..., itemn})` is an optional method that adds a row to the end of a table in the result. `ft_obj` is a handle to the template object previously created. `{item1, item2, ..., itemn}` is a cell array of strings and objects to add to the table. The order of the items in the array determines which column the item is in. If you do not add data to the table, the Model Advisor does not display the table in the result.

Note Before adding rows to a table, you must specify column titles using the `setColTitle` method.

Examples Find all of the blocks in the model and create a table of the blocks:

```
% Create FormatTemplate object, specify table format
ft = ModelAdvisor.FormatTemplate('TableTemplate');

% Add information to the table
setTableTitle(ft, {'Blocks in Model'});
setColTitles(ft, {'Index', 'Block Name'});
% Find all the blocks in the system and add them to a table.
allBlocks = find_system(system);
for inx = 2 : length(allBlocks)
    % Add information to the table
    addRow(ft, {inx-1,allBlocks(inx)});
end
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.Group.addTask

Purpose Add task to folder

Syntax `addTask(group_obj, task_obj)`

Description `addTask(group_obj, task_obj)` adds a task, specified by `task_obj`, to the folder `group_obj.group_obj` is an instantiation of the `ModelAdvisor.Group` class.

Example Add three tasks to MAG.

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');  
addTask(MAG, MAT1);  
addTask(MAG, MAT2);  
addTask(MAG, MAT3);
```

Purpose

Get names of all models associated with cvdata objects in cv.cvdatagroup

Syntax

```
models = allNames(cvdg)
```

Description

models = allNames(cvdg) returns a cell array of strings identifying all model names associated with the cvdata objects in cvdg, an instantiation of the cv.cvdatagroup class.

Examples

Add three cvdata objects to cvdg and return a cell array of model names:

```
a = cvdata;  
b = cvdata;  
c = cvdata;  
cvdg = cv.cvdatagroup;  
add (cvdg, a, b, c);  
model_names = allNames(cvdg)
```

cv.cvtestgroup.allNames

Purpose Get names of all models associated with `cvtest` objects in `cvtestgroup`

Syntax `models = allNames(cvtg)`

Description `models = allNames(cvtg)` returns a cell array of strings identifying all model names associated with the `cvtest` objects in `cvtg`, an instantiation of the `cv.cvtestgroup` class.

Examples Add three `cvtest` objects to `cvtg` and return a cell array of model names:

```
d = cvtest;
e = cvtest;
f = cvtes;
cvtg = cv.cvtestgroup;
add (cvtg, d, e, f);
model_names = allNames(cvtg)
```


Purpose

Collect condition coverage information for model object

Syntax

```
coverage = conditioninfo(cvdo, object)
coverage = conditioninfo(cvdo, object, ignore_descendants)
[coverage, description] = conditioninfo(cvdo, object)
```

Description

`coverage = conditioninfo(cvdo, object)` returns condition coverage results from the cvdata object `cvdo` for the model component specified by `object`.

`coverage = conditioninfo(cvdo, object, ignore_descendants)` returns condition coverage results for `object`, depending on the value of `ignore_descendants`.

`[coverage, description] = conditioninfo(cvdo, object)` returns condition coverage results and textual descriptions of each condition in `object`.

Inputs

`cvdo`

cvdata object

`ignore_descendants`

Logical value that specifies whether to ignore the coverage of descendant objects

1 to ignore coverage of descendant objects

0 (default) to collect coverage of descendant objects

`object`

An object in the Simulink model or Stateflow diagram that receives decision coverage. Valid values for `object` are as follows:

`BlockPath`

Full path to a Simulink model or block

`BlockHandle`

Handle to a Simulink model or block

conditioninfo

<code>s1Obj</code>	Handle to a Simulink API object
<code>sfID</code>	Stateflow ID
<code>sfObj</code>	Handle to a Stateflow API object
<code>{BlockPath, sfID}</code>	Cell array with the path to a Stateflow chart and the ID of an object contained in that chart
<code>{BlockPath, sfObj}</code>	Cell array with the path to a Stateflow chart and a Stateflow object API handle contained in that chart
<code>[BlockHandle, sfID]</code>	Array with a Stateflow block handle and the ID of an object contained in that chart

Outputs

`coverage`

The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`. `coverage` is empty if `cvdo` does not contain condition coverage results for object. The two elements are:

<code>covered_outcomes</code>	Number of condition outcomes satisfied for object
<code>total_outcomes</code>	Total number of condition outcomes for object

`description`

A structure array with the following fields:

text	String describing a condition or the block port to which it applies
trueCnts	Number of times the condition was true in a simulation
falseCnts	Number of times the condition was false in a simulation

Examples

The following example opens the `slvndemo_cv_small_controller` demo model, creates the test specification object `testObj`, enables condition coverage for `testObj`, and executes `testObj`. Then retrieve the condition coverage results for the Logic block (in the Gain subsystem) and determine its percentage of condition outcomes covered:

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.condition = 1;
data = cvsim(testObj)
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');
cov = conditioninfo(data, blk_handle)
percent_cov = 100 * cov(1) / cov(2)
```

Alternatives

To collect condition coverage for a model using the GUI:

- 1 Open the model for which you want condition coverage.
- 2 In the Model Editor, select **Tools > Coverage Settings**.
- 3 On the **Coverage** tab, under **Coverage Metrics**, select **Condition Coverage**.
- 4 View the **Results** and **Report** tab to specify the type of output you need.
- 5 Click **OK**.

conditioninfo

6 Simulate the model.

See Also

[decisioninfo](#) | [mcdcinfo](#) | [sigrangeinfo](#) | [tableinfo](#)

How To

- “Condition Coverage (CC)” on page 5-5

Purpose	Collection of cvdata objects						
Description	Instances of this class contain a collection of cvdata objects. For more information, see “Extracting Results from cv.cvdatagroup” on page 5-85.						
Construction	<table><tr><td>cv.cvdatagroup</td><td>Create collection of cvdata objects for model reference hierarchy</td></tr></table>	cv.cvdatagroup	Create collection of cvdata objects for model reference hierarchy				
cv.cvdatagroup	Create collection of cvdata objects for model reference hierarchy						
Methods	<table><tr><td>allNames</td><td>Get names of all models associated with cvdata objects in cv.cvdatagroup</td></tr><tr><td>get</td><td>Get cvdata object</td></tr><tr><td>getAll</td><td>Get all cvdata objects</td></tr></table>	allNames	Get names of all models associated with cvdata objects in cv.cvdatagroup	get	Get cvdata object	getAll	Get all cvdata objects
allNames	Get names of all models associated with cvdata objects in cv.cvdatagroup						
get	Get cvdata object						
getAll	Get all cvdata objects						
Properties	<table><tr><td>name</td><td>cv.cvdatagroup object name</td></tr></table>	name	cv.cvdatagroup object name				
name	cv.cvdatagroup object name						
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.						

cv.cvdatagroup

Purpose Create collection of cvdata objects for model reference hierarchy

Syntax `cvdg = cv.cvdatagroup(cvdo1, cvdo2, ...)`

Description `cvdg = cv.cvdatagroup(cvdo1, cvdo2, ...)` creates an instantiation of the `cv.cvdatagroup` class (`cvdg`) that contains the `cvdata` objects `cvdo1`, `cvdo2`, etc. A `cvdata` object contains results of the simulation runs.

Examples Create an instantiation of the `cv.cvdatagroup` class and add two `cvdata` objects to it:

```
a = cvdata;  
b = cvdata;  
cvdg = cv.cvdatagroup(a, b);
```

Purpose	Collection of <code>cvtest</code> objects
Description	Instances of this class contain a collection of <code>cvtest</code> objects. For more information, see “Creating a Test Group with <code>cv.cvtestgroup</code> ” on page 5-84.
Construction	<code>cv.cvtestgroup</code> Create collection of <code>cvtest</code> objects for model reference hierarchy
Methods	<code>add</code> Add <code>cvtest</code> objects <code>allNames</code> Get names of all models associated with <code>cvtest</code> objects in <code>cvtestgroup</code> <code>get</code> Get <code>cvtest</code> objects
Properties	<code>name</code> <code>cv.cvtestgroup</code> object name
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

cv.cvtestgroup

Purpose Create collection of cvtest objects for model reference hierarchy

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

Description `cvtg = cv.cvtestgroup(cvto1, cvto2, ...)` creates an instantiation of the `cv.cvtestgroup` class (`cvtg`) that contains the `cvtest` objects `cvto1`, `cvto2`, etc. A `cvtest` object is a test specification object for a Simulink model.

Examples Create an instantiation of the `cv.cvtestgroup` class and add two `cvtest` objects to it:

```
a = cvtest;  
b = cvtest;  
cvtg = cv.cvtestgroup(a, b);
```

See Also `cvtest`

Purpose	Exit model coverage environment
Syntax	<code>cvexit</code>
Description	<code>cvexit</code> exits the model coverage environment. Issuing this command closes the Coverage Display window and removes coloring from a block diagram that displays its model coverage results.

cvhtml

Purpose Produce HTML report from model coverage objects

Syntax

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

Description `cvhtml(file, cvdo)` creates an HTML report of the coverage results in the `cvdata` or `cv.cvdatagroup` object `cvdo` when you run model coverage in simulation. `cvhtml` saves the coverage results in `file`. The model must be open when you use `cvhtml` to generate its coverage report.

`cvhtml(file, cvdo1, cvdo2, ...)` creates a combined report of several `cvdata` objects. The results from each object appear in a separate column of the HTML report. Each `cvdata` object must correspond to the same root model or subsystem. Otherwise, the function fails.

`cvhtml(file, cvdo1, cvdo2, ..., options)` creates a combined report of several `cvdata` objects using the report options specified by `options`.

`cvhtml(file, cvdo1, cvdo2, ..., options, detail)` creates a combined report of several `cvdata` objects and specifies the detail level of the report with the value of `detail`.

Inputs

`cvdo`

A `cv.cvdatagroup` object

`detail`

Specifies the level of detail in the report. Set `detail` to an integer from 0 to 3. Greater numbers for `detail` indicate greater detail.

Default: 2

`file`

String specifying the HTML file in the MATLAB current folder where cvhtml stores the results

Default: []

options

Specify the report options that you specify in options:

- To enable an option, set it to 1 (e.g., ' -hTR=1 ').
- To disable an option, set it 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 ').

The following table lists all the options:

Option	Description	Default
-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

Examples

Make sure you have write access to the default MATLAB directory. Create a cumulative coverage report for the `slvndemo_cv_small_controller` mode and save it as `ratelim_coverage.html`:

```
model = 'slvndemo_cv_small_controller';
open_system(model);
cvt = cvtest(model);
cvd = cvsim(cvt);
outfile = 'ratelim_coverage.html';
cvhtml(outfile, cvd);
```

Alternatives

To create an HTML model coverage report:

- 1** Open the model for which you want model coverage.
- 2** In the Model Editor, select **Tools > Coverage Settings**.
- 3** On the **Report** tab of the Coverage Settings dialog box, select **Generate HTML report**.
- 4** Click **OK**.

See Also

`cv.cvdatagroup` | `cvsim`

How To

- “Producing HTML Reports with `cvhtml`” on page 5-77

Purpose Load coverage tests and stored results into memory

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

Description `[cvtos, cvdos] = cvload(filename)` loads the tests and data stored in the text file `filename.cvt`. `cvtos` is a cell array of `cvtest` objects that are successfully loaded. `cvdos` is a cell array of `cvdata` objects that are successfully loaded. `cvdos` has the same size as `cvtos`, but if a particular test has no results, `cvdos` can contain empty elements.

`[cvtos, cvdos] = cvload(filename, restoretotal)` restores or clears the cumulative results from prior runs, depending on the value of `restoretotal`. If `restoretotal` is 1, `cvload` restores the cumulative results from prior runs. If `restoretotal` is unspecified or 0, `cvload` clears the model's cumulative results.

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

- If a model with the same name exists in the coverage database, the software loads only the compatible results that reference the existing model 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, the software loads only the results that are consistent with the earlier files.

Examples Store coverage results in `cvtest` and `cvdata` objects:

```
(test_objects, data_objects) = cvload(test_results, 1);
```

See Also `cvsave`

How To • “Loading Stored Coverage Test Results with `cvload`” on page 5-79

cvmodelview

Purpose Display model coverage results with model coloring

Syntax cvmodelview(cvdo)

Description cvmodelview(cvdo) displays coverage results from the cvdata object cvdo by coloring the objects in the model that have model coverage results.

Examples Open the slvndemo_cv_small_controller demo model, create the test specification object testObj, and execute testObj to collect model coverage. Run cvmodelview to color the model objects for which you collect model coverage information:

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
data = cvsim(testObj)
cvmodelview(data)
```

Alternatives To display model coverage results by coloring objects:

- 1 Open the model.
- 2 Select **Tools > Coverage Settings**.
- 3 On the **Coverage** tab, select **Coverage for this model**.
- 4 On the **Results** tab, select **Display coverage results using model coloring**.
- 5 Click **OK** to close the Coverage Settings dialog box.
- 6 Simulate the model.

See Also cvsim

How To • “Enabling the Colored Diagram Display” on page 5-69

- “Displaying Model Coverage with Model Coloring” on page 5-70

Purpose Save coverage tests and results to file

Syntax

```
cvsave(filename, model)
cvsave(filename, cvto1, cvto2, ...)
cvsave(filename, cvdo1, cvdo2, ...)
```

Description `cvsave(filename, model)` saves all the tests (cvtest objects) and results (cvdata objects) related to `model` in the text file `filename.cvt`. `model` is a handle to or name of a Simulink model.

`cvsave(filename, cvto1, cvto2, ...)` saves multiple cvtest objects in the text file `filename.cvt`. `cvsave` also saves information about any referenced models.

`cvsave(filename, cvdo1, cvdo2, ...)` saves the tests and test results for multiple cvdata objects to the text file `filename.cvt`. `cvsave` also saves information about any referenced models.

Examples Save coverage results for the `slvndemo_cv_small_controller` model in `ratelim_testdata.cvt`:

```
model = 'slvndemo_cv_small_controller';
open_system(model);
cvt = cvtest(model);
cvd = cvsim(cvt);
cvsave('ratelim_testdata', model);
```

Alternatives To save cumulative coverage results:

- 1** In the Model Editor, select **Tools > Coverage Settings**.
- 2** On the **Results** tab:
 - a** Select **Save cumulative results in workspace variable**.
 - b** Select **Save last run in workspace variable**.
- 3** Click OK to close the Coverage Settings dialog box.

4 Simulate the model.

See Also

`cvload`

How To

- “Saving Test Runs to a File with `cvsave`” on page 5-78

Purpose Simulate and return model coverage results for test objects

Syntax

```
cvdo = cvsim(cvto)
[cvdo,t,x,y] = cvsim(cvto)
[cvdo,t,x,y] = cvsim(cvto, timespan, options)
[cvdo,t,x,y] = cvsim(cvto, label, setupcmd)
[cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)
```

Description

`cvdo = cvsim(cvto)` simulates the `cvtest` object `cvto` by starting a simulation run for the corresponding model. The software returns the results in the `cvdata` object `cvdo`. However, when recording coverage for multiple models in a hierarchy, `cvsim` returns its results in a `cv.cvdatagroup` object.

`[cvdo,t,x,y] = cvsim(cvto)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation.

`[cvdo,t,x,y] = cvsim(cvto, timespan, options)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation, and overrides default simulation values with the values for `timespan` and `options`.

`[cvdo,t,x,y] = cvsim(cvto, label, setupcmd)` creates the `cvtest` object `cvto` and simulates it in one command. The arguments `label` and `setupcmd` are passed directly to the `cvtest` function, which creates the `cvtest` object `cvto`.

`[cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)` executes the `cvtest` objects `cvto1`, `cvto2`, ... and returns the results in the set of `cvdata` objects `cvdo1`, `cvdo2`,

You do not have to enable model coverage reporting for the model to use the `cvsim` command.

Inputs

`cvto`
cvtest object

`label`

Label for test object (passed to cvtest)

Default: []

options

Optional simulation parameters specified as a structure created by the `simset` command.

setupcmd

Setup command used to create test object (passed to cvtest)

Default: []

timespan

Simulation start and stop time:

`tFinal` To specify the stop time. The start time is 0.

`[tStart tFinal]` To specify the start and stop times.

`[tStart OutputTimes tFinal]` To specify the start and stop times and time points to be returned in `t`.

Generally, `t` includes more time points. `OutputTimes` is equivalent to specifying **Configuration Parameters > Data Import/Export > Output options > Produce specified output only**.

Outputs

cvdo

cvdata object

t

The simulation's time vector

x

The simulation's state matrix consisting of continuous states followed by discrete states

y

The simulation's output matrix. Each column contains the output of a root-level Outport block, in port number order. If any Outport block has a vector input, its output takes the appropriate number of columns.

Examples

Simulate the `slvndemo_cv_small_controller` model, get the test data, and simulate the model with that test data. `cvsim` returns the time vector, matrix of state values, and matrix of output values:

```
model = 'slvndemo_cv_small_controller';  
open_system(model);  
testObj = cvtest(model); %Get test data  
[data, T, X, Y] = cvsim(testObj); %Get coverage data
```

See Also

`cv.cvdatagroup` | `cvtest` | `simset`

How To

- “Creating and Running Test Cases” on page 5-11

Purpose

Simulate and return model coverage results for referenced models

Syntax

```
cvdg = cvsimref(topModelName)
cvdg = cvsimref(topModelName, cvtg)
[cvdg,t,x,y] = cvsimref(topModelName, cvtg)
[cvdg,t,x,y] = cvsimref(topModelName, cvtg, timespan,
    options)
[cvdg1, cvdg2, ...] = cvsimref(topModelName, cvtg1, cvtg2,
    ...)
```

Description

`cvdg = cvsimref(topModelName)` simulates the top model and all referenced models in the hierarchy, collects model coverage data, and returns the results in the `cv.cvdagroup` object `cvdg`. 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, cvtg)` simulates `topModelName` and collects model coverage data by executing the `cv.cvtestgroup` object `cvtg`. `cvtg` contains `cvtest` specifications for the top-level model and all the referenced models in the hierarchy. `cvsimref` returns the model coverage results in `cvdg`.

`[cvdg,t,x,y] = cvsimref(topModelName, cvtg)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation.

`[cvdg,t,x,y] = cvsimref(topModelName, cvtg, timespan, options)` overrides default simulation values with the values in `timespan` and `options`.

`[cvdg1, cvdg2, ...] = cvsimref(topModelName, cvtg1, cvtg2, ...)` executes multiple `cv.cvtestgroup` objects and returns the results in a set of `cv.cvdagroup` objects.

Inputs

`cvtg`

`cv.cvtestgroup` object that contains test specifications for the referenced models in the hierarchy

options

Optional simulation parameters specified as a structure created by the `simset` command.

timespan

Simulation start and stop time:

<code>tFinal</code>	Specify the stop time. The start time is 0.
<code>[tStart tFinal]</code>	Specify the start and stop times.
<code>[tStart OutputTimes tFinal]</code>	Specify that <code>cvsimref</code> return the start and stop times and time points in <code>t</code> . Generally, <code>t</code> includes more time points. <code>OutputTimes</code> is equivalent to specifying Configuration Parameters > Data Import/Export > Output options > Produce specified output only .

topModelName

Name of the top-level model in the hierarchy

Outputs

cvdgc

`cv.cvdtagroup` object

t

The simulation time vector

x

The simulation state matrix consisting of continuous states followed by discrete states

y

The simulation output matrix. Each column contains the output of a root-level Outport block, in port number order. If any Outport block has a vector input, its output takes the appropriate number of columns.

Examples

Open and simulate the `slvndemo_ratelim_harness` model and its two subsystems:

```
topModel = 'slvndemo_cv_mutual_exclusion';  
load_system(topModel);  
% Make sure coverage is off for this run for the entire tree  
set_param(topModel, 'RecordCoverage', 'off');  
set_param(topModel, 'CovModelRefEnable', 'Off');  
[T1, X1, Y1] = sim(topModel);           % Normal data  
[allData, T2, X2, Y2] = cvsimref(topModel); % cvsimref data
```

See Also

`cv.cvdatagroup` | `cv.cvtestgroup` | `cvsim` | `cvtest` | `simset`

How To

- “Creating and Running Test Cases” on page 5-11
- “Using Model Coverage Commands for Referenced Models” on page 5-81

cvtest

Purpose Create model coverage test specification object

Syntax

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

Description `cvto = cvtest(root)` creates a test specification object with the handle `cvto`. Simulate `cvto` with the `cvsim` command.

`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`.

Inputs

`label`

Label for test object

`root`

The name of, or a handle to, a Simulink model or a subsystem. Only the specified model or subsystem and its descendants are subject to model coverage testing.

`setupcmd`

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

Outputs

`cvto`

A test specification object with the following structure:

Field	Description
<code>id</code>	Read-only internal ID

Field	Description
<code>modelcov</code>	Read-only internal ID
<code>rootPath</code>	Name of system or subsystem for analysis
<code>label</code>	String used when reporting results
<code>setupCmd</code>	Command executed in base workspace prior to simulation
<code>settings.condition</code>	Set to 1 for condition coverage.
<code>settings.decision</code>	Set to 1 for decision coverage.
<code>settings.designverifier</code>	Set to 1 for coverage for Simulink Design Verifier blocks.
<code>settings.mcdc</code>	Set to 1 for MC/DC coverage.
<code>settings.sigrange</code>	Set to 1 for signal range coverage.
<code>settings.sigsize</code>	Set to 1 for signal size coverage.
<code>settings.tableExec</code>	Set to 1 for lookup table coverage.
<code>modelRefSettings.enable</code>	<ul style="list-style-type: none"> • 'off' — Disable coverage for all referenced models. • 'all' or on — Enable coverage for all referenced models. • 'filtered' — Enable coverage only for referenced models not listed in the <code>excludedModels</code> subfield.
<code>modelRefSettings.excludeTopModel</code>	Set to 1 to exclude coverage for the top model.
<code>modelRefSettings.excludedModels</code>	String specifying a comma-separated list of referenced models for which coverage is disabled.

Field	Description
emlSettings. enableExternal	Set to 1 to enable coverage for external M-files called by Embedded MATLAB functions in your model.
options. forceBlockReduction	Set to 1 to override the Simulink Block reduction parameter if it is enabled.

Examples

Create a `cvtest` object of the Adjustable Rate Limiter block in the demo model `slvndemo_ratelim_harness` and display its contents:

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

See Also

`cv.cvtestgroup`

How To

- “Creating Tests with `cvtest`” on page 5-74
- “Creating a Test Group with `cv.cvtestgroup`” on page 5-84

Purpose

Display decision coverage information for model object

Syntax

```
coverage = decisioninfo(cvdo, object)
coverage = decisioninfo(cvdo, object, ignore_descendants)
[coverage, description] = decisioninfo(cvdo, object)
```

Description

`coverage = decisioninfo(cvdo, object)` returns decision coverage results from the cvdata object `cvdo` for the model component specified by `object`.

`coverage = decisioninfo(cvdo, object, ignore_descendants)` returns decision coverage results for `object`, depending on the value of `ignore_descendants`.

`[coverage, description] = decisioninfo(cvdo, object)` returns decision coverage results and text descriptions of decision points associated with `object`.

Inputs

`cvdo`

cvdata object

`ignore_descendants`

Specifies to ignore the coverage of descendant objects if `ignore_descendants` is set to 1.

`object`

The `object` argument specifies an object in the model or Stateflow chart that received decision coverage. Valid values for `object` include the following:

Object Specification

BlockPath

BlockHandle

s1obj

Description

Full path to a model or block

Handle to a model or block

Handle to a Simulink API object

Object Specification	Description
<code>sfID</code>	Stateflow ID
<code>sfObj</code>	Handle to a Stateflow API object
<code>{BlockPath, sfID}</code>	Cell array with the path to a Stateflow chart and the ID of an object contained in that chart
<code>{BlockPath, sfObj}</code>	Cell array with the path to a Stateflow chart and a Stateflow object API handle contained in that chart
<code>[BlockHandle, sfID]</code>	Array with a Stateflow chart handle and the ID of an object contained in that chart

Outputs

`coverage`

The value of `coverage` is a two-element vector of the form `[covered_outcomes total_outcomes]`. `coverage` is empty if `cvdo` does not contain decision coverage results for `object`. The two elements are:

<code>covered_outcomes</code>	Number of decision outcomes satisfied for <code>object</code>
<code>total_outcomes</code>	Number of decision outcomes for <code>object</code>

`description`

`description` is a structure array containing the following fields:

<code>decision.text</code>	String describing a decision point, e.g., 'U > LL'
<code>decision.outcome.text</code>	String describing a decision outcome, i.e., 'true' or 'false'
<code>decision.outcome.executionCount</code>	Number of times a decision outcome occurred in a simulation

Examples

Open the `slvndemo_cv_small_controller` model and create the test specification object `testObj`. Enable decision coverage for `testObj` and execute `testObj` using `cvsim`. Use `decisioninfo` to retrieve the decision coverage results for the Saturation block and determine the percentage of decision outcomes covered:

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.decision = 1;
data = cvsim(testObj)
blk_handle = get_param([mdl, '/Saturation'], 'Handle');
cov = decisioninfo(data, blk_handle)
percent_cov = 100 * cov(1) / cov(2)
```

Alternatives

To collect and display decision coverage results:

- 1 Open the model.
- 2 In the Model Editor, select **Tools > Coverage Settings**.
- 3 On the **Coverage** tab, under **Coverage Metrics**, select **Decision Coverage**.
- 4 Click **OK** to close the Coverage Settings dialog box.
- 5 Simulate the model and review the results.

See Also

[conditioninfo](#) | [cvsim](#) | [mcdcinfo](#) | [sigrangeinfo](#) | [tableinfo](#)

How To

- “Condition Coverage (CC)” on page 5-5

Purpose Get cvdata object

Syntax `get(cvdg, model_name)`

Description `get(cvdg, model_name)` returns the cvdata object in the `cv.cvdatagroup` object `cvdg` that corresponds to the model specified in `model_name`.

Example Get a cvdata object from the specified Simulink model:

```
get(cvdg, 'slvndemo_cv_small_controller');
```

cv.cvtestgroup.get

Purpose	Get cvtest objects
Syntax	<code>get(cvtg, model_name)</code>
Description	<code>get(cvtg, model_name)</code> returns the cvtest object in the <code>cv.cvtestgroup</code> object <code>cvtg</code> that corresponds to the model specified in <code>model_name</code> .
Example	Get a cvtest object from the specified Simulink model: <pre>get(cvtg, 'slvndemo_cv_small_controller');</pre>
See Also	<code>cvsimref</code> , <code>cvtest</code>

Purpose	Get all cvdata objects
Syntax	<code>getAll(cvdo)</code>
Description	<code>getAll(cvdo)</code> returns all cvdata objects in the <code>cv.cvdatagroup</code> object <code>cvdo</code> .
Example	Return all cvdata object from the specified Simulink model: <pre>getAll(cvdg, 'slvndemo_cv_small_controller');</pre>

getCoverageInfo

Purpose Coverage information for Simulink Design Verifier blocks

Syntax

```
[coverage, description] = getCoverageInfo(cvdo, object)
[coverage, description] = getCoverageInfo(cvdo, object,
    metric)
[coverage, description] = getCoverageInfo(cvdo, object,
    metric, ignore_descendants)
```

Description

[coverage, description] = getCoverageInfo(cvdo, object) collects Simulink Design Verifier coverage for *object*, based on coverage results in *cvdo*. *object* can be a handle to any block, subsystem, or Stateflow chart. `getCoverageData` returns coverage data only for Simulink Design Verifier library blocks in *object*'s hierarchy.

[coverage, description] = getCoverageInfo(cvdo, object, *metric*) returns coverage data for the block type specified in *metric*. If *object* does not match the block type, `getCoverageInfo` does not return any data.

[coverage, description] = getCoverageInfo(cvdo, object, *metric*, *ignore_descendants*) returns coverage data about *object*, omitting coverage data for its descendant objects if *ignore_descendants* equals 1.

Inputs

cvdo

cvdata object

object

In the model or Stateflow chart, object that received Simulink Design Verifier coverage. The following are valid values for *object*.

BlockPath	Full path to a model or block
BlockHandle	Handle to a model or block
s1obj	Handle to a Simulink API object

<code>sfID</code>	Stateflow ID from a singly instantiated Stateflow chart
<code>sfObj</code>	Handle to a Stateflow API object from a singly instantiated Stateflow chart
<code>{BlockPath, sfID}</code>	Cell array with the path to a Stateflow chart and the ID of an object in that chart
<code>{BlockPath, sfObj}</code>	Cell array with the path to a Stateflow chart and a handle to a Stateflow object in that chart
<code>[BlockHandle, sfID]</code>	Array with a Stateflow chart handle and the ID of an object in that chart

metric

`cvmetric.Sldv` enumeration object with values that correspond to Simulink Design Verifier library blocks.

<code>test</code>	Test Objective block
<code>proof</code>	Proof Objective block
<code>condition</code>	Test Condition block
<code>assumption</code>	Proof Assumption block

ignore_descendants

Boolean value that specifies to ignore the coverage of descendant objects if set to 1.

Outputs

coverage

Two-element vector of the form `[covered_outcomes total_outcomes]`.

getCoverageInfo

<i>covered_outcomes</i>	Number of test objectives satisfied for <i>object</i>
<i>total_outcomes</i>	Total number of test objectives for <i>object</i>

coverage is empty if *cvdo* does not contain decision coverage results for *object*.

description

Structure array containing descriptions of each test objective, and descriptions and execution counts for each outcome within *object*.

Examples

Collect and display coverage data for the Test Objective block named True in the `sldvdemo_debounce_testobjblks` model:

```
mdl = 'sldvdemo_debounce_testobjblks';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.designverifier = 1;
data = cvsim(testObj)
blk_handle = get_param([mdl, '/True'], 'Handle');
getCoverageInfo(data, blk_handle)
```

Alternatives

To collect and display coverage results for Simulink Design Verifier library blocks using the Coverage Settings dialog box:

- 1 Open the model.
- 2 In the Model Editor, select **Tools > Coverage Settings**.
- 3 On the **Coverage** tab, under **Coverage Metrics**, select **Simulink Design Verifier**.
- 4 Click **OK**.
- 5 Simulate the model and review the results.

See Also

[conditioninfo](#) | [cvsim](#) | [decisioninfo](#) | [mcdcinfo](#) | [sigrangeinfo](#) | [tableinfo](#)

How To

- “Simulink Design Verifier Coverage” on page 5-7

ModelAdvisor.Table.getEntry

Purpose	Get table cell contents	
Syntax	<code>content = getEntry(table, row, column)</code>	
Description	<code>content = getEntry(table, row, column)</code> gets the contents of the specified cell.	
Inputs	<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
	<code>row</code>	An integer specifying the row
	<code>column</code>	An integer specifying the column
Outputs	<code>content</code>	An element object or object array specifying the content of the table entry
Example	Get the content of the table cell in the third column, third row: <pre>table1 = ModelAdvisor.Table(4, 4); . . . content = getEntry(table1, 3, 3);</pre>	
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks	

Purpose

Return check identifier

Syntax

```
id = getID(check_obj)
```

Description

`id = getID(check_obj)` returns the ID of the check `check_obj`. `id` is a unique string that identifies the check.

You create this unique identifier when you create the check. This unique identifier is the equivalent of the `ModelAdvisor.Check ID` property.

See Also

“Defining Custom Checks” on page 7-11 — Describes how to create custom actions

Customizing the Model Advisor on page 1 — Describes how to create custom checks

mcdcinfo

Purpose Collect 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 (MC/DC) results from the cvdata object `cvdo` for the model component specified by `object`.

`coverage = mcdcinfo(cvdo, object, ignore_descendants)` returns MC/DC results for `object`, depending on the value of `ignore_descendants`.

`[coverage, description] = mcdcinfo(cvdo, object)` returns MC/DC results and text descriptions of each condition/decision in `object`.

Inputs

`cvdo`

cvdata object

`ignore_descendants`

Logical value specifying whether to ignore the coverage of descendant objects

1 — Ignore coverage of descendant objects

0 — Collect coverage for descendant objects

`object`

The `object` argument specifies an object in the Simulink model or Stateflow diagram that receives decision coverage. Valid values for `object` include the following:

Object Specification	Description
BlockPath	Full path to a model or block

Object Specification	Description
BlockHandle	Handle to a model or block
s1Obj	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 chart and the ID of an object contained in that chart
{BlockPath, sfObj}	Cell array with the path to a Stateflow chart and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow chart handle and the ID of an object contained in that chart

Outputs

coverage

Two-element vector of the form [covered_outcomes total_outcomes]. `coverage` is empty if `cvdo` does not contain modified condition/decision coverage results for object. The two elements are:

covered_outcomes	Number of condition/decision outcomes satisfied for object
total_outcomes	Total number of condition/decision outcomes for object

description

A structure array containing the following fields:

<code>text</code>	String denoting whether the condition/decision is associated with a block output or Stateflow transition
<code>condition.text</code>	String describing a condition/decision or the block port to which it applies
<code>condition.achieved</code>	Logical array indicating whether a condition case has been fully covered
<code>condition.trueRslt</code>	String representing a condition case expression that produces a true result
<code>condition.falseRslt</code>	String representing a condition case expression that produces a false result

Examples

Collect MC/DC coverage for the `slvndemo_cv_small_controller` model and determine the percentage of MC/DC coverage collected for the Logic block in the Gain subsystem:

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)           %Create test specification object
testObj.settings.mcdc = 1;     %Enable MC/DC coverage
data = cvsim(testObj)         %Simulate model
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');
cov = mcdcinfo(data, blk_handle) %Retrieve MC/DC results for Logic block
percent_cov = 100 * cov(1) / cov(2) %Percentage of MC/DC outcomes covered
```

Alternatives

To collect MC/DC coverage for a model:

- 1 Open the model.
- 2 In the Model Editor, select **Tools > Coverage Settings**.

3 On the **Coverage** tab, under **Coverage Metrics**, select **MCDC Coverage**.

4 On the **Results** and **Report** tabs, select the desired options.

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

6 Simulate the model and review the MC/DC coverage in the report.

See Also

[conditioninfo](#) | [cvsim](#) | [decisioninfo](#) | [sigrangeinfo](#) | [tableinfo](#)

How To

- “Modified Condition/Decision Coverage (MCDC)” on page 5-5
- “MCDC Analysis” on page 5-41

ModelAdvisor.Action class

Purpose	Add actions to custom checks	
Description	Instances of this class define actions you take when the Model Advisor checks do not pass. Users access actions by clicking the Action button that you define in the Model Advisor window.	
Construction	ModelAdvisor.Action	Add actions to custom checks
Methods	setCallbackFcn	Specify action callback function
Properties	Description	Message in Action box
	Name	Action button label
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.	
Example	<pre>% define action (fix) operation myAction = ModelAdvisor.Action; myAction.Name='Fix block fonts'; myAction.Description=... 'Click the button to update all blocks with specified font';</pre>	
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks	

Purpose Add actions to custom checks

Syntax `action_obj = ModelAdvisor.Action`

Description `action_obj = ModelAdvisor.Action` creates a handle to an action object.

Note

- Include an action definition in a check definition.
 - Each check can contain only one action.
-

Example

```
% define action (fix) operation  
myAction = ModelAdvisor.Action;
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Check class

Purpose Create custom checks

Description The `ModelAdvisor.Check` class creates a Model Advisor check object. All checks must have an associated `ModelAdvisor.Task` object to be displayed in the Model Advisor tree.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** is displayed in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When you use checks in task definitions, the following rules apply:

- If you define the properties of the check in the check definition and the task definition, the task definition takes precedence. The Model Advisor displays the information contained in the task definition. For example, if you define the name of the check in the task definition using the `ModelAdvisor.Task.DisplayName` property and in the check definition using the `ModelAdvisor.Check.Title` property, the Model Advisor displays the information provided in `ModelAdvisor.Task.DisplayName`.
- If you define the properties of the check in the check definition but not the task definition, the task uses the properties from the check. For example, if you define the name of the check in the check definition using the `ModelAdvisor.Check.Title` property, and you register the check using a task definition, the Model Advisor displays the information provided in `ModelAdvisor.Check.Title`.
- If you define the properties of the check in the task definition but not the check definition, the Model Advisor displays the information correctly as long as you register the task with the Model Advisor instead of the check. For example, if you define the name of the check in the task definition using the `ModelAdvisor.Task.DisplayName` property instead of the `ModelAdvisor.Check.Title` property, and you register the check

using a task definition, the Model Advisor displays the information provided in `ModelAdvisor.Task.DisplayName`.

Construction

<code>ModelAdvisor.Check</code>	Create custom checks
---------------------------------	----------------------

Methods

<code>getID</code>	Return check identifier
<code>setAction</code>	Specify action for check
<code>setCallbackFcn</code>	Specify callback function for check
<code>setInputParameters</code>	Specify input parameters for check
<code>setInputParametersLayoutGrid</code>	Specify layout grid for input parameters

Properties

<code>CallbackContext</code>	Model or subsystem context
<code>CallbackHandle</code>	Callback function handle for check
<code>CallbackStyle</code>	Callback function type
<code>Enable</code>	Indicate whether user can enable or disable check
<code>ID</code>	Identifier for check
<code>LicenseName</code>	Product license names required to display and run check
<code>ListViewVisible</code>	Status of button
<code>Result</code>	Results cell array
<code>Title</code>	Name of check

ModelAdvisor.Check class

TitleTips	Description of check
Value	Status of check
Visible	Indicate to display or hide check

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

Purpose Create custom checks

Syntax `check_obj = ModelAdvisor.Check(check_ID)`

Description `check_obj = ModelAdvisor.Check(check_ID)` creates a check object, `check_obj`, and assigns it a unique identifier, `check_ID`. `check_ID` must remain constant. To display checks in the Model Advisor tree, all checks must have an associated `ModelAdvisor.Task` or `ModelAdvisor.Root` object.

Note You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution appears** in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

Example `rec = ModelAdvisor.Check('com.mathworks.sample.Check1');`

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.FactoryGroup class

Purpose	Define subfolder in By Task folder	
Description	The ModelAdvisor.FactoryGroup class defines a new subfolder to add to the By Task folder.	
Construction	ModelAdvisor.FactoryGroup	Define subfolder in By Task folder
Methods	addCheck	Add check to folder
Properties	Description	Description of folder
	DisplayName	Name of folder
	ID	Identifier for folder
	MAObj	Model Advisor object
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.	
Example	<pre>% --- sample factory group rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');</pre>	
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks	

Purpose

Define subfolder in **By Task** folder

Syntax

```
fg_obj = ModelAdvisor.FactoryGroup(fg_ID)
```

Description

`fg_obj = ModelAdvisor.FactoryGroup(fg_ID)` creates a handle to a factory group object, `fg_obj`, and assigns it a unique identifier, `fg_ID`. `fg_ID` must remain constant.

Example

```
% --- sample factory group  
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.FormatTemplate class

Purpose Template for formatting Model Advisor analysis results

Description Use the `ModelAdvisor.FormatTemplate` class to format the result of a check in the analysis result pane of the Model Advisor for a uniform look and feel among the checks you create. There are two formats for the analysis result:

- Table
- List

Construction `ModelAdvisor.FormatTemplate` Construct template object for formatting Model Advisor analysis results

Methods

<code>addRow</code>	Add row to table
<code>setCheckText</code>	Add description of check to result
<code>setColTitles</code>	Add column titles to table
<code>setInformation</code>	Add description of subcheck to result
<code>setListObj</code>	Add list of hyperlinks to model objects
<code>setRecAction</code>	Add Recommended Action section and text
<code>setRefLink</code>	Add See Also section and links
<code>setSubBar</code>	Add line between subcheck results
<code>setSubResultStatus</code>	Add status to check or subcheck result
<code>setSubResultStatusText</code>	Add text below status in result

ModelAdvisor.FormatTemplate class

setSubTitle	Add title for subcheck in result
setTableInfo	Add data to table
setTableTitle	Add title to table

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

The following code creates two template objects, `ft1` and `ft2`, and uses them to format the result of running the check in a table and a list. The result identifies the blocks in the model. The graphics following the code display the output as it appears in the Model Advisor when the check passes and fails.

```
% Sample Check With Subchecks Callback Function
function ResultDescription = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

%Initialize variables
ResultDescription={};
ResultStatus = false; % Default check status is 'Warning'
mdladvObj.setCheckResultStatus(ResultStatus);

% Create FormatTemplate object for first subcheck, specify table format
ft1 = ModelAdvisor.FormatTemplate('TableTemplate');

% Add information describing the overall check
setCheckText(ft1, ['Find and report all blocks in the model. '...
    '(setCheckText method - Description of what the check reviews)']);

% Add information describing the subcheck
setSubTitle(ft1, 'Table of Blocks (setSubTitle method - Title of the subcheck)');
setInformation(ft1, ['Find and report all blocks in a table. '...
    '(setInformation method - Description of what the subcheck reviews)']);

% Add See Also section for references to standards
```

ModelAdvisor.FormatTemplate class

```
setRefLink(ft1, {'Standard 1 reference (setRefLink method)',
               {'Standard 2 reference (setRefLink method)'});

% Add information to the table
setTableTitle(ft1, {'Blocks in the Model (setTableTitle method)'});
setColTitles(ft1, {'Index (setColTitles method)',
                  'Block Name (setColTitles method)'});

% Perform the check actions
allBlocks = find_system(system);
if length(find_system(system)) == 1
    % Add status for subcheck
    setSubResultStatus(ft1, 'Warn');
    setSubResultStatusText(ft1, ['The model does not contain blocks. '...
                                '(setSubResultStatusText method - Description of result status)']);
    setRecAction(ft1, {'Add blocks to the model. '...
                      '(setRecAction method - Description of how to fix the problem)'});
    ResultStatus = false;
else
    % Add status for subcheck
    setSubResultStatus(ft1, 'Pass');
    setSubResultStatusText(ft1, ['The model contains blocks. '...
                                '(setSubResultStatusText method - Description of result status)']);
    for inx = 2 : length(allBlocks)
        % Add information to the table
        addRow(ft1, {inx-1,allBlocks(inx)});
    end
    ResultStatus = true;
end

% Pass table template object for subcheck to Model Advisor
ResultDescription{end+1} = ft1;

% Create FormatTemplate object for second subcheck, specify list format
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');

% Add information describing the subcheck
```

ModelAdvisor.FormatTemplate class

```
setSubTitle(ft2, 'List of Blocks (setSubTitle method - Title of the subcheck)');
setInformation(ft2, ['Find and report all blocks in a list. '...
    '(setInformation method - Description of what the subcheck reviews)']);

% Add See Also section for references to standards
setRefLink(ft2, {'Standard 1 reference (setRefLink method)'},
    {'Standard 2 reference (setRefLink method)'});


% Last subcheck, suppress line
setSubBar(ft2, false);

% Perform the subcheck actions
if length(find_system(system)) == 1
    % Add status for subcheck
    setSubResultStatus(ft2, 'Warn');
    setSubResultStatusText(ft2, ['The model does not contain blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    setRecAction(ft2, {'Add blocks to the model. '...
        '(setRecAction method - Description of how to fix the problem)'});
    ResultStatus = false;
else
    % Add status for subcheck
    setSubResultStatus(ft2, 'Pass');
    setSubResultStatusText(ft2, ['The model contains blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    % Add information to the list
    setListObj(ft2, allBlocks);
end

% Pass list template object for the subcheck to Model Advisor
ResultDescription{end+1} = ft2;
% Set overall check status
mdladvObj.setCheckResultStatus(ResultStatus);
```

ModelAdvisor.FormatTemplate class

The following graphic displays the output as it appears in the Model Advisor when the check passes.

Result:  Passed

Find and report all blocks in the model. (setCheckText method - Description of what the check reviews)

Table of Blocks (setSubTitle method - Title of the subcheck)
Find and report all blocks in a table. (setInformation method - Description of what the subcheck reviews)

See Also

- Standard 1 reference (setRefLink method)
- Standard 2 reference (setRefLink method)

Passed
The model contains blocks. (setSubResultStatusText method - Description of result status)

Blocks in the Model (setTableTitle method)

Index (setColTitles method)	Block Name (setColTitles method)
1	format template test/Constant
2	format template test/Constant1
3	format template test/Gain
4	format template test/Product
5	format template test/Out1

List of Blocks (setSubTitle method - Title of the subcheck)
Find and report all blocks in a list. (setInformation method - Description of what the subcheck reviews)

See Also


- Standard 1 reference (setRefLink method)
- Standard 2 reference (setRefLink method)

Passed
The model contains blocks. (setSubResultStatusText method - Description of result status)

- [format template test](#)
- [format template test/Constant](#)
- [format template test/Constant1](#)
- [format template test/Gain](#)
- [format template test/Product](#)
- [format template test/Out1](#)

ModelAdvisor.FormatTemplate class

The following graphic displays the output as it appears in the Model Advisor when the check fails.

Result:  Warning

Find and report all blocks in the model. (setCheckText method - Description of what the check reviews)

Table of Blocks (setSubTitle method - Title of the subcheck)
Find and report all blocks in a table. (setInformation method - Description of what the subcheck reviews)

See Also

- Standard 1 reference (setRefLink method)
- Standard 2 reference (setRefLink method)

Warning
The model does not contain blocks. (setSubResultStatusText method - Description of result status)

Recommended Action
Add blocks to the model.
(setRecAction method - Description of how to fix the problem)

List of Blocks (setSubTitle method - Title of the subcheck)
Find and report all blocks in a list. (setInformation method - Description of what the subcheck reviews)

See Also

- Standard 1 reference (setRefLink method)
- Standard 2 reference (setRefLink method)

Warning
The model does not contain blocks. (setSubResultStatusText method - Description of result status)

Recommended Action
Add blocks to the model.
(setRecAction method - Description of how to fix the problem)

Alternatives

Use the Model Advisor Formatting API to format check analysis results, however The MathWorks recommends that you use the

ModelAdvisor.FormatTemplate class

ModelAdvisor.FormatTemplate class for a uniform look and feel among the checks you create.

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

Purpose Construct template object for formatting Model Advisor analysis results

Syntax `obj = ModelAdvisor.FormatTemplate('type')`

Description `obj = ModelAdvisor.FormatTemplate('type')` creates a handle, *obj*, to an object of the `ModelAdvisor.FormatTemplate` class. *type* is a string identifying the format type of the template, either list or table. Valid values are `ListTemplate` and `TableTemplate`.

You must return the result object to the Model Advisor to display the formatted result in the analysis result pane.

Note Use the `ModelAdvisor.FormatTemplate` class in check callbacks.

Examples Create a template object, `ft`, and use it to create a list template:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.Group class

Purpose	Define custom folder								
Description	The <code>ModelAdvisor.Group</code> class defines a folder that is displayed in the Model Advisor tree. Use folders to consolidate checks by functionality or usage.								
Construction	<table><tr><td><code>ModelAdvisor.Group</code></td><td>Define custom folder</td></tr></table>	<code>ModelAdvisor.Group</code>	Define custom folder						
<code>ModelAdvisor.Group</code>	Define custom folder								
Methods	<table><tr><td><code>addGroup</code></td><td>Add subfolder to folder</td></tr><tr><td><code>addTask</code></td><td>Add task to folder</td></tr></table>	<code>addGroup</code>	Add subfolder to folder	<code>addTask</code>	Add task to folder				
<code>addGroup</code>	Add subfolder to folder								
<code>addTask</code>	Add task to folder								
Properties	<table><tr><td>Description</td><td>Description of folder</td></tr><tr><td><code>DisplayName</code></td><td>Name of folder</td></tr><tr><td>ID</td><td>Identifier for folder</td></tr><tr><td><code>MAObj</code></td><td>Model Advisor object</td></tr></table>	Description	Description of folder	<code>DisplayName</code>	Name of folder	ID	Identifier for folder	<code>MAObj</code>	Model Advisor object
Description	Description of folder								
<code>DisplayName</code>	Name of folder								
ID	Identifier for folder								
<code>MAObj</code>	Model Advisor object								
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation .								
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks								

Purpose

Define custom folder

Syntax

```
group_obj = ModelAdvisor.Group(group_ID)
```

Description

`group_obj = ModelAdvisor.Group(group_ID)` creates a handle to a group object, `group_obj`, and assigns it a unique identifier, `group_ID`. `group_ID` must remain constant.

Examples

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Image class

Purpose	Include image in Model Advisor output	
Description	The <code>ModelAdvisor.Image</code> class adds an image to the Model Advisor output.	
Construction	<code>ModelAdvisor.Image</code>	Include image in Model Advisor output
Methods	<code>setHyperlink</code>	Specify hyperlink location
	<code>setImageSource</code>	Specify image location
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.	
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks “Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results	

Purpose Include image in Model Advisor output

Syntax `object = ModelAdvisor.Image`

Description `object = ModelAdvisor.Image` creates a handle to an image object, object, that the Model Advisor displays in the output. The Model Advisor supports many image formats, including, but not limited to, JPEG, BMP, and GIF.

Examples `image_obj = ModelAdvisor.Image;`

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.InputParameter class

Purpose	Add input parameters to custom checks	
Description	Instances of the <code>ModelAdvisor.InputParameter</code> class specify the input parameters a custom check uses in analyzing the model. Access input parameters in the Model Advisor window.	
Construction	<code>ModelAdvisor.InputParameter</code>	Add input parameters to custom checks
Methods	<code>setColSpan</code>	Specify number of columns for input parameter
	<code>setRowSpan</code>	Specify rows for input parameter
Properties	Description	Description of input parameter
	Entries	Drop-down list entries
	Name	Input parameter name
	Type	Input parameter type
	Value	Value of input parameter
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation .	
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks	

Purpose Add input parameters to custom checks

Syntax `input_param = ModelAdvisor.InputParameter`

Description `input_param = ModelAdvisor.InputParameter` creates a handle to an input parameter object, `input_param`.

Note You must include input parameter definitions in a check definition.

Example

Note The following example is a fragment of code from the `sl_customization.m` file for the demo model, `slvndemo_mdadv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

ModelAdvisor.InputParameter

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.setInputParametersLayoutGrid([3 2]);
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.LineBreak class

Purpose	Insert line break
Description	Use instances of the <code>ModelAdvisor.LineBreak</code> class to insert line breaks in the Model Advisor outputs.
Construction	<code>ModelAdvisor.LineBreak</code> Insert line break
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks “Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.LineBreak

Purpose Insert line break

Syntax ModelAdvisor.LineBreak

Description ModelAdvisor.LineBreak inserts a line break into the Model Advisor output.

Example Add a line break between two lines of text:

```
result = ModelAdvisor.Paragraph;  
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

Purpose	Create list class
Description	Use instances of the <code>ModelAdvisor.List</code> class to create list-formatted outputs.
Construction	<code>ModelAdvisor.List</code> Create list class
Methods	<code>addItem</code> Add item to list <code>setType</code> Specify list type
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks “Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.List

Purpose Create list class

Syntax `list = ModelAdvisor.List`

Description `list = ModelAdvisor.List` creates a list object, `list`.

Example

```
subList = ModelAdvisor.List();
setType(subList, 'numbered')
addItem(subList, ModelAdvisor.Text('Sub entry 1', {'pass', 'bold'}));
addItem(subList, ModelAdvisor.Text('Sub entry 2', {'pass', 'bold'}));
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.ListViewParameter class

Purpose	Add list view parameters to custom checks	
Description	The Model Advisor uses list view parameters to populate the Model Advisor Result Explorer. Access the information in list views by clicking Explore Result in the Model Advisor window.	
Construction	ModelAdvisor.ListViewParameter	Add list view parameters to custom checks
Properties	Attributes	Attributes to display in Model Advisor Report Explorer
	Data	Objects in Model Advisor Result Explorer
	Name	Drop-down list entry
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.	

Example

Note The following example is a fragment of code from the `sl_customization.m` file for the demo model, `slvndemo_mdadv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object');
```

ModelAdvisor.ListViewParameter class

```
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.ListViewParameter

Purpose Add list view parameters to custom checks

Syntax `lv_param = ModelAdvisor.ListViewParameter`

Description `lv_param = ModelAdvisor.ListViewParameter` defines a list view, `lv_param`.

Note Include list view parameter definitions in a check definition.

See Also

- “Defining Model Advisor Result Explorer Views” on page 7-18 — Describes how to create check list views
- Customizing the Model Advisor on page 1 — Describes how to create custom checks
- “Batch-Fixing Warnings or Failures” — Describes how to use list views in the Model Advisor
- “Demo and Code Example” on page 8-20 — Describes how to run a demo that shows how to customize the Model Advisor
- “getListViewParameters” — Describes how to get list view parameters of a check
- “setListViewParameters” — Describes how to set list view parameters of a check

ModelAdvisor.Paragraph class

Purpose	Create and format paragraph	
Description	The ModelAdvisor.Paragraph class creates and formats a paragraph object.	
Construction	ModelAdvisor.Paragraph	Create and format paragraph
Methods	addItem	Add item to paragraph
	setAlign	Specify paragraph alignment
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.	
Example	<pre>% Check Simulation optimization setting ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '... 'optimization settings:']);</pre>	
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks “Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results	

Purpose Create and format paragraph

Syntax `para_obj = ModelAdvisor.Paragraph`

Description `para_obj = ModelAdvisor.Paragraph` defines a paragraph object `para_obj`.

Example

```
% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '...
'optimization settings:']);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Root class

Purpose	Identify root node
Description	The <code>ModelAdvisor.Root</code> class returns the root object.
Construction	<code>ModelAdvisor.Root</code> Identify root node
Methods	<code>publish</code> Publish object in Model Advisor root <code>register</code> Register object in Model Advisor root
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks

Purpose	Identify root node
Syntax	<code>root_obj = ModelAdvisor.Root</code>
Description	<code>root_obj = ModelAdvisor.Root</code> creates a handle to the root object, <code>root_obj</code> .
Example	<code>mdladvRoot = ModelAdvisor.Root;</code>
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Table class

Purpose	Create table																				
Description	Instances of the <code>ModelAdvisor.Table</code> class create and format a table. Specify the number of rows and columns in a table, excluding the table title and table heading row.																				
Construction	<code>ModelAdvisor.Table</code> Create table																				
Methods	<table><tr><td><code>getEntry</code></td><td>Get table cell contents</td></tr><tr><td><code>setColHeading</code></td><td>Specify table column title</td></tr><tr><td><code>setColHeadingAlign</code></td><td>Specify column title alignment</td></tr><tr><td><code>setColWidth</code></td><td>Specify column widths</td></tr><tr><td><code>setEntry</code></td><td>Add cell to table</td></tr><tr><td><code>setEntryAlign</code></td><td>Specify table cell alignment</td></tr><tr><td><code>setHeading</code></td><td>Specify table title</td></tr><tr><td><code>setHeadingAlign</code></td><td>Specify table title alignment</td></tr><tr><td><code>setRowHeading</code></td><td>Specify table row title</td></tr><tr><td><code>setRowHeadingAlign</code></td><td>Specify table row title alignment</td></tr></table>	<code>getEntry</code>	Get table cell contents	<code>setColHeading</code>	Specify table column title	<code>setColHeadingAlign</code>	Specify column title alignment	<code>setColWidth</code>	Specify column widths	<code>setEntry</code>	Add cell to table	<code>setEntryAlign</code>	Specify table cell alignment	<code>setHeading</code>	Specify table title	<code>setHeadingAlign</code>	Specify table title alignment	<code>setRowHeading</code>	Specify table row title	<code>setRowHeadingAlign</code>	Specify table row title alignment
<code>getEntry</code>	Get table cell contents																				
<code>setColHeading</code>	Specify table column title																				
<code>setColHeadingAlign</code>	Specify column title alignment																				
<code>setColWidth</code>	Specify column widths																				
<code>setEntry</code>	Add cell to table																				
<code>setEntryAlign</code>	Specify table cell alignment																				
<code>setHeading</code>	Specify table title																				
<code>setHeadingAlign</code>	Specify table title alignment																				
<code>setRowHeading</code>	Specify table row title																				
<code>setRowHeadingAlign</code>	Specify table row title alignment																				
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.																				
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks “Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results																				

Purpose Create table

Syntax `table = ModelAdvisor.Table(row, column)`

Description `table = ModelAdvisor.Table(row, column)` creates a table object (`table`). The Model Advisor displays the table object containing the specified number of rows (`row`) and columns (`column`).

Examples In the following example, you create two table objects, `table1` and `table2`. The Model Advisor displays `table1` in the results as a table with 1 row and 1 column. The Model Advisor display `table2` in the results as a table with 2 rows and 3 columns.

```
table1 = ModelAdvisor.Table(1,1);  
table2 = ModelAdvisor.Table(2,3);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Task class

Purpose Define custom tasks

Description The `ModelAdvisor.Task` class is a wrapper for a check so that you can access the check with the Model Advisor.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** is displayed in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Construction `ModelAdvisor.Task` Define custom tasks

Methods `setCheck` Specify check used in task

Properties

<code>Description</code>	Description of task
<code>DisplayName</code>	Name of task
<code>Enable</code>	Indicate if user can enable and disable task
<code>ID</code>	Identifier for task
<code>LicenseName</code>	Product license names required to display and run task
<code>MAObj</code>	Model Advisor object
<code>Value</code>	Status of task
<code>Visible</code>	Indicate to display or hide task

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');  
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
```

See Also

Chapter 7, “Authoring Custom Checks” — Describes how to create custom checks

ModelAdvisor.Task

Purpose Define custom tasks

Syntax `task_obj = ModelAdvisor.Task(task_ID)`

Description `task_obj = ModelAdvisor.Task(task_ID)` creates a task object, `task_obj`, with a unique identifier, `task_ID`. `task_ID` must remain constant. If you do not specify `task_ID`, the Model Advisor assigns a random `task_ID` to the task object.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** appears in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Examples In the following example, you create three task objects, `MAT1`, `MAT2`, and `MAT3`.

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');  
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

Purpose	Create Model Advisor text output	
Description	Instances of <code>ModelAdvisor.Text</code> class create formatted text for the Model Advisor output.	
Construction	<code>ModelAdvisor.Text</code>	Create Model Advisor text output
Methods	<code>setBold</code>	Specify bold text
	<code>setColor</code>	Specify text color
	<code>setHyperlink</code>	Specify hyperlinked text
	<code>setItalic</code>	Italicize text
	<code>setRetainSpaceReturn</code>	Retain spacing and returns in text
	<code>setSubscript</code>	Specify subscripted text
	<code>setSuperscript</code>	Specify superscripted text
	<code>setUnderlined</code>	Underline text
Copy Semantics	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.	
Examples	<pre>t1 = ModelAdvisor.Text('This is some text');</pre>	
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks “Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results	

ModelAdvisor.Text

Purpose Create Model Advisor text output

Syntax `text = ModelAdvisor.Text(content, attribute)`

Description `text = ModelAdvisor.Text(content, attribute)` creates a text object for the Model Advisor output.

Inputs

<code>content</code>	Optional string specifying the content of the text object. If <code>content</code> is empty, empty text is output.
----------------------	--

<code><i>attribute</i></code>	Optional string specifying the formatting of the content. If no <code>attribute</code> is specified, the output text has default coloring with no formatting. Possible formatting options include:
-------------------------------	--

- `normal` (default) — Text is default color and style.
- `bold` — Text is bold.
- `italic` — Text is italicized.
- `underlined` — Text is underlined.
- `pass` — Text is green.
- `warn` — Text is yellow.
- `fail` — Text is red.
- `keyword` — Text is blue.
- `subscript` — Text is subscripted.
- `superscript` — Text is superscripted.
- `retainspacereturn` — Text retains spacing and returns.

Outputs

text The text object you create

Example

```
text = ModelAdvisor.Text('Sub entry 1', {'pass', 'bold'})
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.Root.publish

Purpose Publish object in Model Advisor root

Syntax `publish(root_obj, check_obj, location)`
`publish(root_obj, group_obj)`
`publish(root_obj, fg_obj)`

Description `publish(root_obj, check_obj, location)` specifies where the Model Advisor places the check in the Model Advisor tree. `location` is either one of the subfolders in the **By Product** folder, or the name of a new subfolder to put in the **By Product** folder. Use a pipe-delimited string to indicate multiple subfolders. For example, to add a check to the **Simulink Verification and Validation > Modeling Standards** folder, use the following string: 'Simulink Verification and Validation|Modeling Standards'.

`publish(root_obj, group_obj)` specifies the `ModelAdvisor.Group` object to publish as a folder in the **Model Advisor Task Manager** folder.

`publish(root_obj, fg_obj)` specifies the `ModelAdvisor.FactoryGroup` object to publish as a subfolder in the **By Task** folder.

Example

```
% publish check into By Product > Demo group.  
mdladvRoot.publish(rec, 'Demo');
```

See Also

- “Defining Where Custom Checks Appear” on page 7-14 in the Simulink® Verification and Validation™ User’s Guide on page 1
- “Defining Where Tasks Appear” on page 8-17 in the Simulink® Verification and Validation™ User’s Guide on page 1
- “Defining Where Custom Folders Appear” on page 8-19 in the Simulink® Verification and Validation™ User’s Guide on page 1

Purpose

Register object in Model Advisor root

Syntax

```
register(MAobj, obj)
```

Description

`register(MAobj, obj)` registers the object, *obj*, in the root object *MAobj*.

In the Model Advisor memory, the `register` method registers the following types of objects:

- `ModelAdvisor.Check`
- `ModelAdvisor.Task`
- `ModelAdvisor.Group`
- `ModelAdvisor.FactoryGroup`

The `register` method places objects in the Model Advisor memory that you use in other functions. The `register` method does not place objects in the Model Advisor tree.

Example

```
mdladvRoot = ModelAdvisor.Root;

MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task with input parameter and auto-fix ability';
MAT1.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT1);

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';
MAT2.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT2);

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
MAT3.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT3)
```

Purpose

Interact programmatically with Requirements Management Interface

Syntax

```
rmi setup
reqlinks = rmi('createempty')
reqlinks = rmi('get', object)
reqlinks = rmi('get', object, group)
rmi('report', object)
rmi('set', object, reqlinks)
rmi('set', object, reqlinks, group)
rmi('cat', object, reqlinks)
cnt = rmi('count', object)
rmi('clearall', object)
rmi('clearAll', object, 'deep')
rmi register linktypename
rmi unregister linktypename
rmi linktypelist
cmdstr = rmi('navcmd', object)
[cmdstr, titlestr] = rmi('navcmd', object)
guidstr = rmi('guidget', object)
object = rmi('guidlookup', model, guidstr)
rmi('highlightModel', object)
rmi('unhighlightModel', object)
rmi('view', object, index)
dialog = rmi('edit', object)
rmi('copyObj', object)
```

Description

`rmi setup` configures RMI for use with your computer and installs the interface for use with the Telelogic® DOORS® software, if needed.

`reqlinks = rmi('createempty')` creates an empty instance of the requirement links data structure.

`reqlinks = rmi('get', object)` returns the requirement links data structure for `object`. `object` is the name or handle of a Simulink or Stateflow object with which requirements can be associated.

`reqlinks = rmi('get', object, group)` returns the requirement links data structure for the Signal Builder group specified by the index

group. In this case, `object` is the name or handle of a Signal Builder block whose signal groups are associated with requirements.

`rmi('report', object)` creates an HTML report that describes the requirements in `object`.

`rmi('set', object, reqlinks)` sets the requirement links data structure `reqlinks` to `object`.

`rmi('set', object, reqlinks, group)` sets the requirement links data structure `reqlinks` to the Signal Builder group specified by the index `group`. In this case, `object` is the name or handle of a Signal Builder block whose signal groups you want to associate with requirements.

`rmi('cat', object, reqlinks)` appends the requirement links data structure `reqlinks` to the end of the existing structure associated with `object`. If no structure exists, RMI sets `reqlinks` to `object`.

`cnt = rmi('count', object)` returns the number of requirement links associated with `object`.

`rmi('clearall', object)` removes the requirement links data structure associated with `object`, deleting its requirements.

`rmi('clearAll', object, 'deep')` deletes all requirements links in the model containing `object`.

`rmi register linktypename` registers the custom link type specified by the M-file function `linktypename`.

`rmi unregister linktypename` removes the custom link type specified by the M-file function `linktypename`.

`rmi linktypelist` displays a list of the currently registered link types. The list indicates whether each link type is built-in or custom, and provides the path to the M-file function used for its registration.

`cmdstr = rmi('navcmd', object)` returns the MATLAB command string used to navigate to `object`.

`[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.

`object = rmi('guidlookup', model, guidstr)` returns the object name in `model` that has the globally unique identifier `guidstr`.

`rmi('highlightModel', object)` highlights all of the objects in the parent model of `object` that have requirement links.

`rmi('unhighlightModel', object)` removes highlighting of objects in the parent model of `object` that have requirement links.

`rmi('view', object, index)` accesses the requirement numbered `index` in the requirements document associated with `object`. `index` is an integer that represents the n th requirement linked to `object`.

`dialog = rmi('edit', object)` displays the Requirements dialog box for `object` and returns the handle of the dialog box.

`rmi('copyObj', object)` resets the globally unique identifier for `object`, preserving its requirement links.

Inputs

`group`

Signal Builder group index

`guidstr`

Globally unique model identifier

`index`

Integer that represents the n th requirement linked to `object`

`object`

Name or handle of a Simulink or Stateflow object with which requirements can be associated.

reqlinks

Requirement links are represented using a MATLAB structure array with the following fields:

`doc` String identifying requirements document

`id` String defining location in requirements document. The first character specifies the identifier type:

First Character	Identifier	Example
?	Search text, the first occurrence of which is located in requirements document	'?Requirement 1'
@	Named item, such as bookmark in a Microsoft Word file or an anchor in an HTML file	'@my_req'
#	Page or item number	'#21'
>	Line number	'>3156'
\$	Worksheet range in a spreadsheet	'\$A2:C5'

`linked` Boolean value specifying whether the requirement link is accessible for report generation and highlighting:

1 (default). Highlight model object and include requirement link in reports.

0

`description` String describing the requirement
`keywords` Optional string supplementing description
`reqsys` String identifying the link type registration name; 'other' for built-in link types

Outputs

`cmdstr`
MATLAB command string

`cnt`
Number of requirement links associated with object

`dialog`
Handle for object

`guidstr`
Globally unique model identifier

`object`
Name or handle of a Simulink or Stateflow object with which requirements can be associated.

`reqlinks`
Requirement links are represented using a MATLAB structure array. See “Inputs” on page 12-100 for details.

`titlestr`
Descriptive text for object

Examples

Get a requirement associated with a block in the `slvndemo_fuelsys_htmreq` model, change its description, and save the requirement back to that block:

```
slvndemo_fuelsys_htmreq;  
blk_with_req = ['slvndemo_fuelsys_htmreq/fuel rate' 10 'controller/...  
Airflow calculation'];
```

```
reqts = rmi('get', blk_with_req);
reqts.description = 'Mass airflow estimation';
rmi('set', blk_with_req, reqts);
rmi('get', blk_with_req);
```

Add a new requirement to the block in the previous example:

```
new_req = rmi('createempty');
new_req.doc = 'fuelsys_requirements2.htm';
new_req.description = 'A new requirement';
rmi('cat',blk_with_req, new_req);
```

Create an HTML requirements report for the `slvndemo_fuelsys_htmreq` model:

```
rmi('report', 'slvndemo_fuelsys_htmreq');
```

How To

- Chapter 2, “Managing Model Requirements”
- “Linking to Custom Types of Requirements Documents” on page 2-40

rmidocrename

Purpose Update model requirements document paths and file names

Syntax `rmidocrename(model_handle, old_path, new_path)`
`rmidocrename(model_name, old_path, new_path)`

Description `rmidocrename(model_handle, old_path, new_path)` collectively updates the links from a Simulink model to requirements files whose names or locations have changed. `model_handle` is a handle to the model that contains links to the files that you have moved or renamed. `old_path` is a string that contains the existing full or partial file or path name. `new_path` is a string with the new full or partial file or path name.

`rmidocrename(model_name, old_path, new_path)` updates the links to requirements files associated with `model_name`. You can pass `rmidocrename` a model handle or a model file name.

When using the `rmidocrename` function, make sure to enter specific strings for the old document name fragments so that you do not inadvertently modify other links.

Examples For the current Simulink model, update all links to requirements files that contain the string 'project_0220', replacing them with 'project_0221':

```
rmidocrename(gcs, '00000220', '00000221')  
Processed 6 objects with requirements, 5 out of 13 links were modified.
```

Alternatives To update the requirements links one at a time, for each model object that has a link:

- 1 For each object with requirements, open the Requirements dialog box by right-clicking and selecting **Requirements > Edit/Add Links**.
- 2 Edit the **Document** field for each requirement that points to a moved or renamed document.
- 3 Click **Apply** to save the changes.

See Also rmi

rminav

Purpose

Start Requirements Management Interface

Syntax

Note rminav will be removed in a future release. Running rminav currently opens the Model Explorer.

Purpose Specify action for check

Syntax `setAction(check_obj, action_obj)`

Description `setAction(check_obj, action_obj)` returns the action object `action_obj` to use in the check `check_obj`. The `setAction` method identifies the action you want to use in a check.

See Also [ModelAdvisor.Action](#) — Create custom actions
[Customizing the Model Advisor on page 1](#) — Describes how to create custom checks

ModelAdvisor.Paragraph.setAlign

Purpose Specify paragraph alignment

Syntax `setAlign(paragraph, alignment)`

Description `setAlign(paragraph, alignment)` specifies the alignment of text. Possible values are:

- 'left' (default)
- 'right'
- 'center'

Example

```
report_paragraph = ModelAdvisor.Paragraph;  
setAlign(report_paragraph, 'center');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

Purpose Specify bold text

Syntax `setBold(text, mode)`

Description `setBold(text, mode)` specifies whether text should be formatted in bold font.

Inputs

`text`

Instantiation of the `ModelAdvisor.Text` class

`mode`

A Boolean value indicating bold formatting of text:

- `true` — Format the text in bold font.
- `false` — Do not format the text in bold font.

Example

```
t1 = ModelAdvisor.Text('This is some text');  
setBold(t1, 'true');
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Action.setCallbackFcn

Purpose Specify action callback function

Syntax setCallbackFcn(action_obj, @handle)

Description setCallbackFcn(action_obj, @handle) specifies the handle to the callback function, handle, to use with the action object, action_obj.

Example

Note The following example is a fragment of code from the `sl_customization.m` file for the demo model, `slvndemo_mdadv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

```
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
modifyAction.setCallbackFcn(@modifyOptimizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
                           ' settings that can impact safety'];
modifyAction.Enable = true;
rec.setAction(modifyAction);
```

See Also

“Defining Check Actions” on page 7-19 — Describes how to create custom actions

Customizing the Model Advisor on page 1 — Describes how to create custom checks

“setActionenable” — Set enable/disable status for check action

ModelAdvisor.Check.setCallbackFcn

Purpose	Specify callback function for check								
Syntax	<code>setCallbackFcn(check_obj, @handle, context, style)</code>								
Description	<code>setCallbackFcn(check_obj, @handle, context, style)</code> specifies the callback function to use with the check, <code>check_obj</code> .								
Inputs	<table><tr><td><code>check_obj</code></td><td>Instantiation of the <code>ModelAdvisor.Check</code> class</td></tr><tr><td><code>handle</code></td><td>Handle to a check callback function</td></tr><tr><td><code>context</code></td><td>Context for checking the model or subsystem:<ul style="list-style-type: none">• 'None' — No special requirements.• 'PostCompile' — The model must be compiled.</td></tr><tr><td><code>style</code></td><td>Type of callback function:<ul style="list-style-type: none">• 'StyleOne' — Simple check callback function, for formatting results using template• 'StyleTwo' — Detailed check callback function• 'StyleThree' — Check callback functions with hyperlinked results</td></tr></table>	<code>check_obj</code>	Instantiation of the <code>ModelAdvisor.Check</code> class	<code>handle</code>	Handle to a check callback function	<code>context</code>	Context for checking the model or subsystem: <ul style="list-style-type: none">• 'None' — No special requirements.• 'PostCompile' — The model must be compiled.	<code>style</code>	Type of callback function: <ul style="list-style-type: none">• 'StyleOne' — Simple check callback function, for formatting results using template• 'StyleTwo' — Detailed check callback function• 'StyleThree' — Check callback functions with hyperlinked results
<code>check_obj</code>	Instantiation of the <code>ModelAdvisor.Check</code> class								
<code>handle</code>	Handle to a check callback function								
<code>context</code>	Context for checking the model or subsystem: <ul style="list-style-type: none">• 'None' — No special requirements.• 'PostCompile' — The model must be compiled.								
<code>style</code>	Type of callback function: <ul style="list-style-type: none">• 'StyleOne' — Simple check callback function, for formatting results using template• 'StyleTwo' — Detailed check callback function• 'StyleThree' — Check callback functions with hyperlinked results								

Example

```
% --- sample check 1
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback, 'None', 'StyleThree');
```

ModelAdvisor.Check.setCallbackFcn

See Also

“Creating Callback Functions and Results” on page 7-22 — Describes how to create check callback functions

Customizing the Model Advisor on page 1 — Describes how to create custom checks

Purpose Specify check used in task

Syntax `setCheck(task, check_ID)`

Description `setCheck(task, check_ID)` specifies the check to use in the task.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution appears** in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Inputs

<code>task</code>	Instantiation of the <code>ModelAdvisor.Task</code> class
<code>check_ID</code>	A unique string that identifies the check to use in the task

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
setCheck(MAT1, 'com.mathworks.sample.Check1');
```

ModelAdvisor.FormatTemplate.setCheckText

Purpose Add description of check to result

Syntax `setCheckText(ft_obj, text)`

Description `setCheckText(ft_obj, text)` is an optional method that adds text or a model advisor template object as the first item in the report. Use this method to add information describing the overall check.

Inputs *ft_obj*

A handle to a template object.

text

A string or a handle to a formatting object.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

text appears as the first line in the analysis result.

Examples Create a list object, *ft*, and add a line of text to the result:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setCheckText(ft, ['Identify unconnected lines, input ports,...
    'and output ports in the model']);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.Table.setColHeading

Purpose

Specify table column title

Syntax

```
setColHeading(table, column, heading)
```

Description

`setColHeading(table, column, heading)` specifies that the column header of column is set to heading.

Inputs

<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
<code>column</code>	An integer specifying the column number
<code>heading</code>	A string, element object, or object array specifying the table column title

Examples

```
table1 = ModelAdvisor.Table(2, 3);  
setColHeading(table1, 1, 'Header 1');  
setColHeading(table1, 2, 'Header 2');  
setColHeading(table1, 3, 'Header 3');
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Table.setColHeadingAlign

Purpose Specify column title alignment

Syntax `setColHeadingAlign(table, column, alignment)`

Description `setColHeadingAlign(table, column, alignment)` specifies the alignment of the column heading.

Inputs

<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
<code>column</code>	An integer specifying the column number
<code><i>alignment</i></code>	Alignment of the column heading. <i>alignment</i> can have one of the following values: <ul style="list-style-type: none">• <code>left</code> (default)• <code>right</code>• <code>center</code>

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setColHeading(table1, 1, 'Header 1');
setColHeadingAlign(table1, 1, 'center');
setColHeading(table1, 2, 'Header 2');
setColHeadingAlign(table1, 2, 'center');
setColHeading(table1, 3, 'Header 3');
setColHeadingAlign(table1, 3, 'center');
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

Purpose Specify text color

Syntax `setColor(text, color)`

Description `setColor(text, color)` sets the text color to *color*.

Inputs

<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class
<code>color</code>	An enumerated string specifying the color of the text. Possible formatting options include: <ul style="list-style-type: none">• <code>normal</code> (default) — Text is default color.• <code>pass</code> — Text is green.• <code>warn</code> — Text is yellow.• <code>fail</code> — Text is red.• <code>keyword</code> — Text is blue.

Example

```
t1 = ModelAdvisor.Text('This is a warning');
setColor(t1, 'warn');
```

ModelAdvisor.InputParameter.setColSpan

Purpose Specify number of columns for input parameter

Syntax `setColSpan(input_param, [start_col end_col])`

Description `setColSpan(input_param, [start_col end_col])` specifies the number of columns that the parameter occupies. Use the `setColSpan` method to specify where you want an input parameter located in the layout grid when there are multiple input parameters.

Inputs	<code>input_param</code>	Instantiation of the <code>ModelAdvisor.InputParameter</code> class
	<code>start_col</code>	A positive integer representing the first column that the input parameter occupies in the layout grid
	<code>end_col</code>	A positive integer representing the last column that the input parameter occupies in the layout grid

Example

```
inputParam2 = ModelAdvisor.InputParameter;  
inputParam2.Name = 'Standard font size';  
inputParam2.Value='12';  
inputParam2.Type='String';  
inputParam2.Description='sample tooltip';  
inputParam2.setRowSpan([2 2]);  
inputParam2.setColSpan([1 1]);
```

ModelAdvisor.FormatTemplate.setColTitles

Purpose

Add column titles to table

Syntax

```
setColTitles(ft_obj, {col_title_1, col_title_2, ...})
```

Description

`setColTitles(ft_obj, {col_title_1, col_title_2, ...})` is method you must use when you create a template object that is a table type. Use it to specify the titles of the columns in the table.

Note Before adding data to a table, you must specify column titles.

Inputs

ft_obj

A handle to a template object.

col_title_N

A cell of strings or handles to formatting objects, specifying the column titles.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

The order of the *col_title_N* inputs determines which column the title is in. If you do not add data to the table, the Model Advisor does not display the table in the result.

Examples

Create a table object, `ft`, and specify two column titles:

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');  
setColTitle(ft, {'Index', 'Block Name'});
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.Table.setColWidth

Purpose Specify column widths

Syntax `setColWidth(table, column, width)`

Description `setColWidth(table, column, width)` specifies the column.

The `setColWidth` method specifies the table column widths relative to the entire table width. If column widths are [1 2 3], the second column is twice the width of the first column, and the third column is three times the width of the first column. Unspecified columns have a default width of 1. For example:

```
setColWidth(1, 1);  
setColWidth(3, 2);
```

specifies [1 1 2] column widths.

Inputs	<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
	<code>column</code>	An integer specifying column number
	<code>width</code>	An integer or array of integers specifying the column widths, relative to the entire table width

Example

```
table1 = ModelAdvisor.Table(2, 3)  
setColWidth(table1, 1, 1);  
setColWidth(table1, 3, 2);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

Purpose

Add cell to table

Syntax

```
setEntry(table, row, column, string)
setEntry(table, row, column, content)
```

Description

setEntry(table, row, column, string) adds a string to a cell in a table.

setEntry(table, row, column, content) adds an object specified by content to a cell in a table.

Inputs

table	Instantiation of the ModelAdvisor.Table class
row	An integer specifying the row
column	An integer specifying the column
string	A string representing the contents of the entry
content	An element object or object array specifying the content of the table entries

Example

Create two tables and insert table2 into the first cell of table1:

```
table1 = ModelAdvisor.Table(1, 1);
table2 = ModelAdvisor.Table(2, 3);
.
.
.
setEntry(table1, 1, 1, table2);
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Table.setEntryAlign

Purpose Specify table cell alignment

Syntax `setEntryAlign(table, row, column, alignment)`

Description `setEntryAlign(table, row, column, alignment)` specifies the cell alignment of the designated cell.

Inputs

<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
<code>row</code>	An integer specifying row number
<code>column</code>	An integer specifying column number
<code><i>alignment</i></code>	A string specifying the cell alignment. Possible values are: <ul style="list-style-type: none">• <code>left</code> (default)• <code>right</code>• <code>center</code>

Example

```
table1 = ModelAdvisor.Table(2,3);
setHeading(table1, 'New Table');
.
.
.
setEntry(table1, 1, 1, 'First Entry');
setEntryAlign(table1, 1, 1, 'center');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Table.setHeading

Purpose Specify table title

Syntax `setHeading(table, title)`

Description `setHeading(table, title)` specifies the table title.

Inputs

<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
<code>title</code>	A string, element object, or object array that specifies the table title

Example

```
table1 = ModelAdvisor.Table(2, 3);
setHeading(table1, 'New Table');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Table.setHeadingAlign

Purpose Specify table title alignment

Syntax `setHeadingAlign(table, alignment)`

Description `setHeadingAlign(table, alignment)` specifies the alignment for the table title.

Inputs

<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
<code><i>alignment</i></code>	A string specifying the table title alignment. Possible values are: <ul style="list-style-type: none">• <code>left</code> (default)• <code>right</code>• <code>center</code>

Example

```
table1 = ModelAdvisor.Table(2, 3);
setHeading(table1, 'New Table');
setHeadingAlign(table1, 'center');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Image.setHyperlink

Purpose Specify hyperlink location

Syntax `setHyperlink(image, url)`

Description `setHyperlink(image, url)` specifies the target location of the hyperlink associated with `image`.

Inputs

<code>image</code>	Instantiation of the <code>ModelAdvisor.Image</code> class
<code>url</code>	A string specifying the target URL

Example

```
matlab_logo=ModelAdvisor.Image;  
setHyperlink(matlab_logo, 'http://www.mathworks.com');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Text.setHyperlink

Purpose	Specify hyperlinked text				
Syntax	<code>setHyperlink(text, url)</code>				
Description	<code>setHyperlink(text, url)</code> creates a hyperlink from the text to the specified URL.				
Inputs	<table><tr><td><code>text</code></td><td>Instantiation of the <code>ModelAdvisor.Text</code> class</td></tr><tr><td><code>url</code></td><td>A string that specifies the target location of the URL</td></tr></table>	<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class	<code>url</code>	A string that specifies the target location of the URL
<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class				
<code>url</code>	A string that specifies the target location of the URL				
Examples	<pre>t1 = ModelAdvisor.Text('MathWorks home page'); setHyperlink(t1, 'http://www.mathworks.com');</pre>				
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks				

ModelAdvisor.Image.setImageSource

Purpose	Specify image location				
Syntax	<code>setImageSource(image_obj, source)</code>				
Description	<code>setImageSource(image_obj, source)</code> specifies the location of the image.				
Inputs	<table><tr><td><code>image_obj</code></td><td>Instantiation of the <code>ModelAdvisor.Image</code> class</td></tr><tr><td><code>source</code></td><td>A string specifying the location of the image file</td></tr></table>	<code>image_obj</code>	Instantiation of the <code>ModelAdvisor.Image</code> class	<code>source</code>	A string specifying the location of the image file
<code>image_obj</code>	Instantiation of the <code>ModelAdvisor.Image</code> class				
<code>source</code>	A string specifying the location of the image file				
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks				

ModelAdvisor.FormatTemplate.setInformation

Purpose Add description of subcheck to result

Syntax `setInformation(ft_obj, text)`

Description `setInformation(ft_obj, text)` is an optional method that adds *text* as the first item after the subcheck title. Use this method to add information describing the subcheck.

Inputs

ft_obj

A handle to a template object.

text

A string or a handle to a formatting object, that describes the subcheck.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

The Model Advisor displays *text* after the title of the subcheck.

Examples Create a list object, *ft*, and specify a subcheck title and description:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft, ['Check for constructs in the model '...
    'that are not supported when generating code']);
setInformation(ft, ['Identify blocks that should not '...
    'be used for code generation.']);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.Check.setInputParameters

Purpose Specify input parameters for check

Syntax setInputParameters(check_obj, params)

Description setInputParameters(check_obj, params) specifies ModelAdvisor.InputParameter objects (params) to be used as input parameters to a check (check_obj).

Inputs

check_obj	Instantiation of the ModelAdvisor.Check class
params	A cell array of ModelAdvisor.InputParameters objects

Examples

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
inputParam1 = ModelAdvisor.InputParameter;
inputParam2 = ModelAdvisor.InputParameter;
inputParam3 = ModelAdvisor.InputParameter;
setInputParameters(rec, {inputParam1,inputParam2,inputParam3});
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
ModelAdvisor.InputParameter— Add input parameters to custom checks

ModelAdvisor.Check.setInputParametersLayoutGrid

Purpose Specify layout grid for input parameters

Syntax `setInputParametersLayoutGrid(check_obj, [row col])`

Description `setInputParametersLayoutGrid(check_obj, [row col])` specifies the layout grid for input parameters in the Model Advisor. Use the `setInputParametersLayoutGrid` method if there are multiple input parameters.

Inputs	<code>check_obj</code>	Instantiation of the <code>ModelAdvisor.Check</code> class
	<code>row</code>	Number of rows in the layout grid
	<code>col</code>	Number of columns in the layout grid

Example

```
% --- sample check 1
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback, 'None', 'StyleThree');
rec.setInputParametersLayoutGrid([3 2]);
```

See Also `ModelAdvisor.InputParameter` — Add input parameters to custom checks
`Customizing the Model Advisor on page 1` — Describes how to create custom checks

Purpose Italicize text

Syntax `setItalic(text, mode)`

Description `setItalic(text, mode)` specifies whether text should be italicized.

Inputs

<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class
<code>mode</code>	A Boolean value indicating italic formatting of text: <ul style="list-style-type: none">• <code>true</code> — Italicize the text.• <code>false</code> — Do not italicize the text.

Example

```
t1 = ModelAdvisor.Text('This is some text');
setItalic(t1, 'true');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.FormatTemplate.setListObj

Purpose Add list of hyperlinks to model objects

Syntax `setListObj(ft_obj, {model_obj})`

Description `setListObj(ft_obj, {model_obj})` is an optional method that generates a bulleted list of hyperlinks to model objects. *ft_obj* is a handle to a list template object. *model_obj* is a cell array of handles or full paths to blocks, or model objects that the Model Advisor displays as a bulleted list of hyperlinks in the report.

Examples Create a list object, `ft`, and add a list of the blocks found in the model:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');

% Find all the blocks in the system
allBlocks = find_system(system);

% Add the blocks to a list
setListObj(ft, allBlocks);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setRecAction

Purpose	Add Recommended Action section and text
Syntax	<code>setRecAction(<i>ft_obj</i>, {<i>text</i>})</code>
Description	<code>setRecAction(<i>ft_obj</i>, {<i>text</i>})</code> is an optional method that adds a Recommended Action section to the report. Use this method to describe how to fix the check.
Inputs	<p><i>ft_obj</i></p> <p>A handle to a template object.</p> <p><i>text</i></p> <p>A cell array of strings or handles to formatting objects, that describes the recommended action to fix the issues reported by the check.</p> <p>Valid formatting objects are: <code>ModelAdvisor.Image</code>, <code>ModelAdvisor.LineBreak</code>, <code>ModelAdvisor.List</code>, <code>ModelAdvisor.Paragraph</code>, <code>ModelAdvisor.Table</code>, and <code>ModelAdvisor.Text</code>.</p> <p>The Model Advisor displays the recommended action as a separate section below the list or table in the report.</p>

Examples Create a list object, `ft`, find Gain blocks in the model, and recommend changing them:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
% Find all Gain blocks
gainBlocks = find_system(gcs, 'BlockType','Gain');

% Find Gain blocks with expression evaluates to 1
for idx = 1:length(gainBlocks)
    gainObj = get_param(gainBlocks(idx), 'Object');
    resGain = slResolve(gainObj.Gain, gainObj.getFullName);
    if ~isempty(resGain)
        % Find the first index that computes to 1
    end
end
```

ModelAdvisor.FormatTemplate.setRecAction

```
        if ~isempty(find(resGain == 1, 1))
            setRecAction(ft, {'If you are using these blocks '...
                'as buffers, you should replace them with '...
                'Signal Conversion blocks'});
        end
    end
end
```

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

Purpose Add See Also section and links

Syntax

```
setRefLink(ft_obj, {'standard'})  
setRefLink(ft_obj, {'url', 'standard'})
```

Description `setRefLink(ft_obj, {'standard'})` is an optional method that adds a See Also section above the table or list in the result. Use this method to add references to standards. `ft_obj` is a handle to a template object. `standard` is a cell array of strings that you want to display in the result. If you include more than one cell, the Model Advisor displays the strings in a bulleted list.

`setRefLink(ft_obj, {'url', 'standard'})` generates a list of links in the See Also section. `url` is a string that indicates the location to link to. You must provide the full link including the protocol. For example, `http:\www.mathworks.com` is a valid link, while `www.mathworks.com` is not a valid link. You can create a link to any protocol that is valid URL, such as a web site address, a full path to a file, or a relative path to a file.

Note `setRefLink` expects a cell array of cell arrays for the second input.

Examples Create a list object, `ft`, and add a related standard:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');  
setRefLink(ft, {'IEC 61508-3, Table A.3 (3) 'Language subset'});
```

Create a list object, `ft`, and add a list of related standards:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');  
setRefLink(ft, {  
    'IEC 61508-3, Table A.3 (2) 'Strongly typed programming language'',...  
    'IEC 61508-3, Table A.3 (3) 'Language subset''});
```

ModelAdvisor.FormatTemplate.setRefLink

See Also

Customizing the Model Advisor on page 1 — Describes how to create custom checks

“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.Text.setRetainSpaceReturn

Purpose	Retain spacing and returns in text				
Syntax	<code>setRetainSpaceReturn(text, mode)</code>				
Description	<code>setRetainSpaceReturn(text, mode)</code> specifies whether the text must retain the spaces and carriage returns.				
Inputs	<table><tr><td><code>text</code></td><td>Instantiation of the <code>ModelAdvisor.Text</code> class</td></tr><tr><td><code>mode</code></td><td>A Boolean value indicating whether to preserve spaces and carriage returns in the text:<ul style="list-style-type: none">• <code>true</code> (default) — Preserve spaces and carriage returns.• <code>false</code> — Do not preserve spaces and carriage returns.</td></tr></table>	<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class	<code>mode</code>	A Boolean value indicating whether to preserve spaces and carriage returns in the text: <ul style="list-style-type: none">• <code>true</code> (default) — Preserve spaces and carriage returns.• <code>false</code> — Do not preserve spaces and carriage returns.
<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class				
<code>mode</code>	A Boolean value indicating whether to preserve spaces and carriage returns in the text: <ul style="list-style-type: none">• <code>true</code> (default) — Preserve spaces and carriage returns.• <code>false</code> — Do not preserve spaces and carriage returns.				
Example	<pre>t1 = ModelAdvisor.Text('MathWorks home page'); setRetainSpaceReturn(t1, 'true');</pre>				
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks				

ModelAdvisor.Table.setRowHeading

Purpose	Specify table row title						
Syntax	<code>setRowHeading(table, row, heading)</code>						
Description	<code>setRowHeading(table, row, heading)</code> specifies a title for the designated table row.						
Inputs	<table><tr><td><code>table</code></td><td>Instantiation of the <code>ModelAdvisor.Table</code> class</td></tr><tr><td><code>row</code></td><td>An integer specifying row number</td></tr><tr><td><code>heading</code></td><td>A string, element object, or object array specifying the table row title</td></tr></table>	<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class	<code>row</code>	An integer specifying row number	<code>heading</code>	A string, element object, or object array specifying the table row title
<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class						
<code>row</code>	An integer specifying row number						
<code>heading</code>	A string, element object, or object array specifying the table row title						
Example	<pre>table1 = ModelAdvisor.Table(2,3); setRowHeading(table1, 1, 'Row 1 Title'); setRowHeading(table1, 2, 'Row 2 Title'); setRowHeading(table1, 3, 'Row 3 Title');</pre>						
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks						

ModelAdvisor.Table.setRowHeadingAlign

Purpose Specify table row title alignment

Syntax `setRowHeadingAlign(table, row, alignment)`

Description `setRowHeadingAlign(table, row, alignment)` specifies the alignment for the designated table row.

Inputs

<code>table</code>	Instantiation of the <code>ModelAdvisor.Table</code> class
<code>row</code>	An integer specifying row number.
<code>alignment</code>	A string specifying the cell alignment. Possible values are: <ul style="list-style-type: none">• <code>left</code> (default)• <code>right</code>• <code>center</code>

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setRowHeading(table1, 1, 'Row 1 Title');
setRowHeadingAlign(table1, 1, 'center');
setRowHeading(table1, 2, 'Row 2 Title');
setRowHeadingAlign(table1, 2, 'center');
setRowHeading(table1, 3, 'Row 3 Title');
setRowHeadingAlign(table1, 3, 'center');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.InputParameter.setRowSpan

Purpose Specify rows for input parameter

Syntax `setRowSpan(input_param, [start_row end_row])`

Description `setRowSpan(input_param, [start_row end_row])` specifies the number of rows that the parameter occupies. Specify where you want an input parameter located in the layout grid when there are multiple input parameters.

Inputs	<code>input_param</code>	The input parameter object
	<code>start_row</code>	A positive integer representing the first row that the input parameter occupies in the layout grid
	<code>end_row</code>	A positive integer representing the last row that the input parameter occupies in the layout grid

Examples

```
inputParam2 = ModelAdvisor.InputParameter;  
inputParam2.Name = 'Standard font size';  
inputParam2.Value='12';  
inputParam2.Type='String';  
inputParam2.Description='sample tooltip';  
inputParam2.setRowSpan([2 2]);  
inputParam2.setColSpan([1 1]);
```

ModelAdvisor.FormatTemplate.setSubBar

Purpose Add line between subcheck results

Syntax `setSubBar(ft_obj, value)`

Description `setSubBar(ft_obj, value)` is an optional method that adds lines between results for subchecks. *ft_obj* is a handle to a template object. *value* is a boolean value that specifies when the Model Advisor includes a line between subchecks in the check results. By default, the value is true, and the Model Advisor displays the bar. The Model Advisor does not display the bar when you set the value to false.

Examples Create a list object, `ft`, turn off the subbar:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubBar(ft, false);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setSubResultStatus

Purpose Add status to check or subcheck result

Syntax `setSubResultStatus(ft_obj, 'status')`

Description `setSubResultStatus(ft_obj, 'status')` is an optional method that displays the status in the result. Use this method to display the status of the check or subcheck in the result. *ft_obj* is a handle to a template object. *status* is a string identifying the status of the check. Valid strings are:

Pass

Warn

Fail

Examples Create a list object, *ft*, and add a passing status:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');  
setSubResultStatus(ft, 'Pass');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setSubResultStatusText

Purpose	Add text below status in result
Syntax	<code>setSubResultStatusText(<i>ft_obj</i>, <i>message</i>)</code>
Description	<code>setSubResultStatusText(<i>ft_obj</i>, <i>message</i>)</code> is an optional method that displays text below the status in the result. Use this method to describe the status.
Inputs	<p><i>ft_obj</i></p> <p>A handle to a template object.</p> <p><i>message</i></p> <p>A string or a handle to a formatting object that the Model Advisor displays below the status in the report.</p> <p>Valid formatting objects are: <code>ModelAdvisor.Image</code>, <code>ModelAdvisor.LineBreak</code>, <code>ModelAdvisor.List</code>, <code>ModelAdvisor.Paragraph</code>, <code>ModelAdvisor.Table</code>, and <code>ModelAdvisor.Text</code>.</p>
Examples	<p>Create a list object, <code>ft</code>, add a passing status and a description of why the check passed:</p> <pre>ft = ModelAdvisor.FormatTemplate('ListTemplate'); setSubResultStatus(ft, 'Pass'); setSubResultStatusText(ft, ['Constructs that are not supported when '... 'generating code were not found in the model or subsystem']);</pre>
See Also	<p>Customizing the Model Advisor on page 1 — Describes how to create custom checks</p> <p>“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results</p>

ModelAdvisor.Text.setSubscript

Purpose	Specify subscripted text				
Syntax	<code>setSubscript(text, mode)</code>				
Description	<code>setSubscript(text, mode)</code> indicates whether to make <code>text</code> subscript.				
Inputs	<table><tr><td><code>text</code></td><td>Instantiation of the <code>ModelAdvisor.Text</code> class</td></tr><tr><td><code>mode</code></td><td>A Boolean value indicating subscripted formatting of text:<ul style="list-style-type: none">• <code>true</code> — Make the text subscript.• <code>false</code> — Do not make the text subscript.</td></tr></table>	<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class	<code>mode</code>	A Boolean value indicating subscripted formatting of text: <ul style="list-style-type: none">• <code>true</code> — Make the text subscript.• <code>false</code> — Do not make the text subscript.
<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class				
<code>mode</code>	A Boolean value indicating subscripted formatting of text: <ul style="list-style-type: none">• <code>true</code> — Make the text subscript.• <code>false</code> — Do not make the text subscript.				
Example	<pre>t1 = ModelAdvisor.Text('This is some text'); setSubscript(t1, 'true');</pre>				
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks				

Purpose Specify superscripted text

Syntax `setSuperscript(text, mode)`

Description `setSuperscript(text, mode)` indicates whether to make text subscript.

Inputs

<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class
<code>mode</code>	A Boolean value indicating superscripted formatting of text: <ul style="list-style-type: none">• <code>true</code> — Make the text superscript.• <code>false</code> — Do not make the text superscript.

Example

```
t1 = ModelAdvisor.Text('This is some text');
setSuperscript(t1, 'true');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.FormatTemplate.setSubTitle

Purpose Add title for subcheck in result

Syntax `setSubTitle(ft_obj, title)`

Description `setSubTitle(ft_obj, title)` is an optional method that adds a subcheck result title. Use this method when you create subchecks to distinguish between them in the result.

Inputs *ft_obj*
A handle to a template object.

title
A string or a handle to a formatting object specifying the title of the subcheck.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

Examples Create a list object, `ft`, and add a subcheck title:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft, ['Check for constructs in the model '...
    'that are not supported when generating code']);
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setTableInfo

Purpose Add data to table

Syntax `setTableInfo(ft_obj, {data})`

Description `setTableInfo(ft_obj, {data})` is an optional method that creates a table. *ft_obj* is a handle to a table template object. *data* is a cell array of strings or objects specifying the information in the body of the table. The Model Advisor creates hyperlinks to objects. If you do not add data to the table, the Model Advisor does not display the table in the result.

Note Before creating a table, you must specify column titles using the `setColTitle` method.

Examples Create a table object, `ft`, add column titles, and add data to the table:

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setColTitle(ft, {'Index', 'Block Name'});
setTableInfo(ft, {'1', 'Gain'});
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

ModelAdvisor.FormatTemplate.setTableTitle

Purpose Add title to table

Syntax `setTableTitle(ft_obj, title)`

Description `setTableTitle(ft_obj, title)` is an optional method that adds a title to a table.

Inputs *ft_obj*

A handle to a template object.

title

A string or a handle to a formatting object specifying the title of the table.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

The title appears above the table. If you do not add data to the table, the Model Advisor does not display the table and title in the result.

Examples Create a table object, `ft`, and add a table title:

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setTitleTitle(ft, 'Table of fonts and styles used in model');
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks
“Formatting Model Advisor Results” on page 7-38 — Describes how to format Model Advisor results

Purpose Specify list type

Syntax setType(list_obj, listType)

Description setType(list_obj, listType) specifies the type of list the ModelAdvisor.List constructor creates.

Inputs

list_obj	Instantiation of the ModelAdvisor.List class
listType	Specifies the list type: <ul style="list-style-type: none">• numbered• bulleted

Example

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

See Also Customizing the Model Advisor on page 1 — Describes how to create custom checks

ModelAdvisor.Text.setUnderlined

Purpose	Underline text				
Syntax	<code>setUnderlined(text, mode)</code>				
Description	<code>setUnderlined(text, mode)</code> indicates whether to underline text.				
Inputs	<table><tr><td><code>text</code></td><td>Instantiation of the <code>ModelAdvisor.Text</code> class</td></tr><tr><td><code>mode</code></td><td>A Boolean value indicating underlined formatting of text:<ul style="list-style-type: none">• <code>true</code> — Underline the text.• <code>false</code> — Do not underline the text.</td></tr></table>	<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class	<code>mode</code>	A Boolean value indicating underlined formatting of text: <ul style="list-style-type: none">• <code>true</code> — Underline the text.• <code>false</code> — Do not underline the text.
<code>text</code>	Instantiation of the <code>ModelAdvisor.Text</code> class				
<code>mode</code>	A Boolean value indicating underlined formatting of text: <ul style="list-style-type: none">• <code>true</code> — Underline the text.• <code>false</code> — Do not underline the text.				
Example	<pre>t1 = ModelAdvisor.Text('This is some text'); setUnderlined(t1, 'true');</pre>				
See Also	Customizing the Model Advisor on page 1 — Describes how to create custom checks				

Purpose

Collect signal range coverage information for model object

Syntax

```
[min, max] = sigrangeinfo(cvdo, object)
[min, max] = sigrangeinfo(cvdo, object, portID)
```

Description

[min, max] = sigrangeinfo(cvdo, object) returns the minimum and maximum signal values output by the model component object within the cvdata object cvdo.

[min, max] = sigrangeinfo(cvdo, object, portID) returns the minimum and maximum signal values associated with the output port portID of the Simulink block object.

Inputs

cvdo

cvdata object

object

An object in the model or Stateflow chart that received decision coverage. Valid values for object include the following:

Object Specification**Description**

BlockPath

Full path to a model or block

BlockHandle

Handle to a model or block

s1Obj

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

{BlockPath, sfObj}

Description

Cell array with the path to a Stateflow chart and a Stateflow object API handle contained in that chart

[BlockHandle, sfID]

Array with a Stateflow chart handle and the ID of an object contained in that chart

portID

Output port of the block object

Outputs

max

Maximum signal values output by the model component object within the cvdata object cvdo. If object outputs a vector, min and max are also vectors.

min

Minimum signal values output by the model component object within the cvdata object cvdo. If object outputs a vector, min and max are also vectors.

Examples

Collect signal range data for the Product block in the slvndemo_cv_small_controller model:

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl) %Create test spec object
testObj.settings.sigrange = 1; %Enable MC/DC coverage
data = cvsim(testObj) %Simulate the model
blk_handle = get_param([mdl, '/Product'], 'Handle');
[minVal, maxVal] = sigrangeinfo(data, blk_handle) %Get signal range data
```

See Also

[conditioninfo](#) | [cvsim](#) | [decisioninfo](#) | [mcdcinfo](#) | [tableinfo](#)

Purpose	Display lookup table coverage information for model object
Syntax	<pre>coverage = tableinfo(cvdo, object) coverage = tableinfo(cvdo, object, ignore_descendants) [coverage, exeCounts] = tableinfo(cvdo, object) [coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)</pre>
Description	<p><code>coverage = tableinfo(cvdo, object)</code> returns lookup table coverage results from the cvdata object <code>cvdo</code> for the model component <code>object</code>.</p> <p><code>coverage = tableinfo(cvdo, object, ignore_descendants)</code> returns lookup table coverage results for <code>object</code>, depending on the value of <code>ignore_descendants</code>.</p> <p><code>[coverage, exeCounts] = tableinfo(cvdo, object)</code> returns lookup table coverage results and the execution count for each interpolation/extrapolation interval in the lookup table block <code>object</code>.</p> <p><code>[coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)</code> returns lookup table coverage results, the execution count for each interpolation/extrapolation interval, and the execution counts for breakpoint equality.</p>
Inputs	<p><code>cvdo</code></p> <p>cvdata object</p> <p><code>ignore_descendants</code></p> <p>Logical value specifying whether to ignore the coverage of descendant objects</p> <ul style="list-style-type: none">1 — Ignore coverage of descendant objects0 — Collect coverage for descendant objects <p><code>object</code></p> <p>Full path or handle to a lookup table block or a model containing a lookup table block.</p>

Outputs

`brkEquality`

A cell array containing vectors that identify the number of times during simulation that the lookup table block input was equivalent to a breakpoint value. Each vector represents the breakpoints along a different lookup table dimension.

`coverage`

The value of `coverage` is a two-element vector of form `[covered_intervals total_intervals]`, the elements of which are:

<code>covered_intervals</code>	Number of interpolation/extrapolation intervals satisfied for object
--------------------------------	--

<code>total_intervals</code>	Total number of interpolation/extrapolation intervals for object
------------------------------	--

`coverage` is empty if `cvdo` does not contain lookup table coverage results for object.

`execounts`

An array having the same dimensionality as the lookup table block; its size has been extended to allow for the lookup table extrapolation intervals.

Examples

Collect lookup table coverage for the `slvndemo_cv_small_controller` model and determine the percentage of interpolation/extrapolation intervals coverage collected for the Gain Table block in the Gain subsystem:

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl) %Create test spec object
testObj.settings.tableExec = 1; %Enable lookup table coverage
data = cvsim(testObj) %Simulate the model
```



```
blk_handle = get_param([mdl, '/Gain/Gain Table'], 'Handle');
cov = tableinfo(data, blk_handle) %Retrieve l/u table coverage
percent_cov = 100 * cov(1) / cov(2) %Percent MC/DC outcomes covered
```

Alternatives

To collect lookup coverage for a model:

- 1 Open the model.
- 2 In the Model Editor, select **Tools > Coverage Settings**.
- 3 On the **Coverage** tab, under **Coverage Metrics**, select **Look-up Table Coverage**.
- 4 On the **Results** and **Report** tabs, select the desired options.
- 5 Click **OK** to close the Coverage Settings dialog box.
- 6 Simulate the model and review the lookup table coverage in the report.

See Also

conditioninfo | cvsim | decisioninfo | mcdinfo | sigrangeinfo

How To

- “Lookup Table Coverage” on page 5-6

ModelAdvisor.ListViewParameter.Attributes property

Purpose Attributes to display in Model Advisor Report Explorer

Values Cell array
Default: {} (empty cell array)

Description The `Attributes` property specifies the attributes to display in the center pane of the Model Advisor Results Explorer.

Example

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object');
myLVParam.Attributes = {'FontName'}; % name is default property
```

ModelAdvisor.Check.CallbackContext property

Purpose Model or subsystem context

Values 'PostCompile'
'None' (default)

Description The `CallbackContext` property specifies the context for checking the model or subsystem.

'None' No special requirements for the model before checking.

'Postcompile' The model must be compiled.

ModelAdvisor.Check.CallbackHandle property

Purpose	Callback function handle for check
Values	Function handle. An empty handle [] is the default.
Description	The <code>CallbackHandle</code> property specifies the handle to the check callback function.

ModelAdvisor.Check.CallbackStyle property

Purpose	Callback function type
Values	'StyleOne' (default) 'StyleTwo' 'StyleThree'
Description	The CallbackStyle property specifies the type of the callback function. 'StyleOne' Simple check callback function 'StyleTwo' Detailed check callback function 'StyleThree' Check callback function with hyperlinked results

ModelAdvisor.ListViewParameter.Data property

Purpose Objects in Model Advisor Result Explorer

Values Array of Simulink objects
Default: [] (empty array)

Description The Data property specifies the objects displayed in the Model Advisor Result Explorer.

Example

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult, 'object');
```

ModelAdvisor.Action.Description property

Purpose

Message in **Action** box

Values

String

Default: '' (null string)

Description

The Description property specifies the message displayed in the Action box.

Example

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
%Specify a callback function for the action
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
myAction.Description=...
    'Click the button to update all blocks with specified font';
```

ModelAdvisor.FactoryGroup.Description property

Purpose Description of folder

Values String
Default: '' (null string)

Description The Description property provides information about the folder. Details about the folder are displayed in the right pane of the Model Advisor.

Example

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.Description='Demo Factory Group';
```


ModelAdvisor.Group.Description property

Purpose Description of folder

Values String

Default: '' (null string)

Description The Description property provides information about the folder. Details about the folder are displayed in the right pane of the Model Advisor.

Example

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');  
MAG.Description='This is my group';
```

ModelAdvisor.InputParameter.Description property

Purpose Description of input parameter

Values String.

Default: '' (null string)

Description The Description property specifies a description of the input parameter. Details about the check are displayed in the right pane of the Model Advisor.

Example

```
% define input parameters
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
```

ModelAdvisor.Task.Description property

Purpose Description of task

Values String

Default: '' (null string)

Description The Description property is a description of the task that the Model Advisor displays in the **Analysis** box.

When adding checks as tasks, the Model Advisor uses the task Description property instead of the check TitleTips property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
MAT1.DisplayName='Example task 1';  
MAT1.Description='This is the first example task.'
```

```
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');  
MAT2.DisplayName='Example task 2';  
MAT2.Description='This is the second example task.'
```

```
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');  
MAT3.DisplayName='Example task 3';  
MAT3.Description='This is the third example task.'
```

ModelAdvisor.FactoryGroup.DisplayName property

Purpose Name of folder

Values String
Default: '' (null string)

Description The DisplayName specifies the name of the folder that is displayed in the Model Advisor.

Examples

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='Demo Factory Group';
```

ModelAdvisor.Group.DisplayName property

Purpose	Name of folder
Values	String Default: '' (null string)
Description	The DisplayName specifies the name of the folder that is displayed in the Model Advisor.
Examples	<pre>MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample'); MAG.DisplayName='My Group';</pre>

ModelAdvisor.Task.DisplayName property

Purpose Name of task

Values String

Default: '' (null string)

Description The `DisplayName` property specifies the name of the task. The Model Advisor displays each custom task in the tree using the name of the task. Therefore, you should specify a unique name for each task. When you specify the same name for multiple tasks, the Model Advisor generates a warning.

When adding checks as tasks, the Model Advisor uses the task `DisplayName` property instead of the check `Title` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
MAT1.DisplayName='Example task with input parameter and auto-fix ability';
```

```
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');  
MAT2.DisplayName='Example task 2';
```

```
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');  
MAT3.DisplayName='Example task 3';
```

ModelAdvisor.Check.Enable property

Purpose

Indicate whether user can enable or disable check

Values

true (default)
false

Description

The Enable property specifies whether the user can enable or disable the check.

true	Display the check box control
false	Hide the check box control

ModelAdvisor.Task.Enable property

Purpose Indicate if user can enable and disable task

Values true (default)
false

Description The Enable property specifies whether the user can enable or disable a task.

true (default)	Display the check box control for task
false	Hide the check box control for task

When adding checks as tasks, the Model Advisor uses the task Enable property instead of the check Enable property.

Example

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
MAT1.Enable = 'false';
```


ModelAdvisor.InputParameter.Entries property

Purpose	Drop-down list entries
Values	Depends on the value of the Type property.
Description	<p>The Entries property is valid only when the Type property is one of the following:</p> <ul style="list-style-type: none">• Enum• ComboBox• PushButton
Examples	<pre>inputParam3 = ModelAdvisor.InputParameter; inputParam3.Name='Valid font'; inputParam3.Type='Combobox'; inputParam3.Description='sample tooltip'; inputParam3.Entries={'Arial', 'Arial Black'};</pre>

ModelAdvisor.Check.ID property

Purpose Identifier for check

Values String

Default: '' (null string)

Description The ID property specifies a permanent, unique identifier for the check. Note the following about the ID property:

- You must specify this property.
- The value of ID must remain constant.
- The Model Advisor generates an error if ID is not unique.
- Tasks and factory group definitions must refer to checks by ID.

ModelAdvisor.FactoryGroup.ID property

Purpose Identifier for folder

Values String

Description The ID property specifies a permanent, unique identifier for the folder.

Note

- You must specify this field.
 - The value of ID must remain constant.
 - The Model Advisor generates an error if ID is not unique.
 - Group definitions must refer to other groups by ID.
-

ModelAdvisor.Group.ID property

Purpose Identifier for folder

Values String

Description The ID property specifies a permanent, unique identifier for the folder.

Note

- You must specify this field.
 - The value of ID must remain constant.
 - The Model Advisor generates an error if ID is not unique.
 - Group definitions must refer to other groups by ID.
-

ModelAdvisor.Task.ID property

Purpose Identifier for task

Values String

Default: '' (null string)

Description The ID property specifies a permanent, unique identifier for the task.

Note

- The Model Advisor automatically assigns a string to ID if you do not specify it.
 - The value of ID must remain constant.
 - The Model Advisor generates an error if ID is not unique.
 - Group definitions must refer to tasks using ID.
-

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
MAT1.ID='Task_ID_1234';
```

ModelAdvisor.Check.LicenseName property

Purpose Product license names required to display and run check

Values Cell array of product license names

{}(empty cell array) (default)

Description The `LicenseName` property specifies a cell array of names for product licenses required to display and run the check.

When the Model Advisor starts, it tests whether the product license exists. If you do not meet the license requirements, the Model Advisor does not display the check.

The Model Advisor performs a checkout of the product licenses when you run the custom check. If you do not have the product licenses available, you see an error message that the required license is not available.

Tip To find the correct text for license strings, type `help license` at the MATLAB command line.

ModelAdvisor.Task.LicenseName property

Purpose

Product license names required to display and run task

Values

Cell array of product license names

Default: {} (empty cell array)

Description

The `LicenseName` property specifies a cell array of names for product licenses required to display and run the check.

When the Model Advisor starts, it tests whether the product license exists. If you do not meet the license requirements, the Model Advisor does not display the check.

The Model Advisor performs a checkout of the product licenses when you run the custom check. If you do not have the product licenses available, you see an error message that the required license is not available.

If you specify `ModelAdvisor.Check.LicenseName`, the Model Advisor displays the check when the union of both properties is true.

Tip To find the correct text for license strings, type `help license` at the MATLAB command line.

ModelAdvisor.Check.ListViewVisible property

Purpose Status of **Explore Result** button

Values false (default)
true

Description The `ListViewVisible` property is a Boolean value that sets the status of the **Explore Result** button.

true	Display the Explore Result button.
false	Hide the Explore Result button.

Example

```
% add 'Explore Result' button
rec.ListViewVisible = true;
```


ModelAdvisor.FactoryGroup.MAObj property

Purpose	Model Advisor object
Values	Handle to a Simulink.ModelAdvisor object
Description	The MAObj property specifies a handle to the current Model Advisor object.

ModelAdvisor.Group.MAObj property

Purpose	Model Advisor object
Values	Handle to <code>Simulink.ModelAdvisor</code> object
Description	The MAObj property specifies a handle to the current Model Advisor object.

ModelAdvisor.Task.MAObj property

Purpose	Model Advisor object
Values	Handle to a Simulink.ModelAdvisor object
Description	<p>The MAObj property specifies the current Model Advisor object.</p> <p>When adding checks as tasks, the Model Advisor uses the task MAObj property instead of the check MAObj property.</p>

cv.cvdatagroup.name property

Purpose	cv.cvdatagroup object name
Values	name
Description	The name property specifies the name of the cv.cvdatagroup object.
Examples	<pre>cvdg = cvsimref(topModelName, cvtg); cvdg.name = 'My_Data_Group';</pre>

Purpose	cv.cvtestgroup object name
Value	name
Description	The name property specifies the name of the cv.cvtestgroup object.
Examples	<pre>cvto1 = cvtest('TopModel'); cvto2 = cvtest('SubModel1'); cvto3 = cvtest('SubModel2'); cvtg = cv.cvtestgroup(cvto1, cvto2, cvto3); cvtg.name = 'My_Test_Group';</pre>
See Also	cvtest

ModelAdvisor.Action.Name property

Purpose Action button label

Values String
Default: '' (null string)

Description The Name property specifies the label for the action button. This property is required.

Example

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
%Specify a callback function for the action
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
```

ModelAdvisor.InputParameter.Name property

Purpose	Input parameter name
Values	String. Default: '' (null string)
Description	The Name property specifies the name of the input parameter in the custom check.
Examples	<pre>inputParam2 = ModelAdvisor.InputParameter; inputParam2.Name = 'Standard font size'; inputParam2.Value='12'; inputParam2.Type='String'; inputParam2.Description='sample tooltip';</pre>

ModelAdvisor.ListViewParameter.Name property

Purpose Drop-down list entry

Values String

Default: '' (null string)

Description The Name property specifies an entry in the **Show** drop-down list in the Model Advisor Result Explorer.

Examples

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
```


ModelAdvisor.Check.Result property

Purpose Results cell array

Values Cell array

Default: {} (empty cell array)

Description The `Result` property specifies the cell array for storing the results that are returned by the callback function specified in `CallbackHandle`.

Tip To set the icon associated with the check, use the `Simulink.ModelAdvisor` `setCheckResultStatus` and `setCheckErrorSeverity` methods.

ModelAdvisor.Check.Title property

Purpose Name of check

Values String
Default: '' (null string)

Description The Title property specifies the name of the check in the Model Advisor. The Model Advisor displays each custom check in the tree using the title of the check. Therefore, you should specify a unique title for each check. When you specify the same title for multiple checks, the Model Advisor generates a warning.

Example

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');  
rec.Title = 'Check Simulink block font';
```

ModelAdvisor.Check.TitleTips property

Purpose Description of check

Values String
Default: '' (null string)

Description The TitleTips property specifies a description of the check. Details about the check are displayed in the right pane of the Model Advisor.

Example

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');  
rec.Title = 'Check Simulink block font';  
rec.TitleTips = 'Example style three callback';
```

ModelAdvisor.InputParameter.Type property

Purpose Input parameter type

Values String.

Default: '' (null string)

Description The Type property specifies the type of input parameter.

Use the Type property with the Value and Entries properties to define input parameters.

Valid values are listed in the following table.

Type	Data Type	Default Value	Description
Bool	Boolean	false	A check box
ComboBox	Cell array	First entry in the list	A drop-down menu <ul style="list-style-type: none">• Use Entries to define the entries in the list.• Use Value to indicate a specific entry in the menu or to enter a value not in the list.
Enum	Cell array	First entry in the list	A drop-down menu <ul style="list-style-type: none">• Use Entries to define the entries in the list.• Use Value to indicate a specific entry in the list.

ModelAdvisor.InputParameter.Type property

Type	Data Type	Default Value	Description
PushButton	N/A	N/A	A button When you click the button, the callback function specified by <code>Entries</code> is called.
String	String	' ' (null string)	A text box

Examples

```
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
```

ModelAdvisor.Check.Value property

Purpose Status of check

Values 'true' (default)
'false'

Description The Value property specifies the initial status of the check.

'true'	Check is enabled
'false'	Check is disabled

Examples

```
% hide all checks that do not belong to Demo group
if ~(strcmp(checkCellArray{i}.Group, 'Demo'))
    checkCellArray{i}.Visible = false;
    checkCellArray{i}.Value = false;
end
```

ModelAdvisor.InputParameter.Value property

Purpose	Value of input parameter
Values	Depends on the Type property.
Description	<p>The Value property specifies the initial value of the input parameter. This property is valid only when the Type property is one of the following:</p> <ul style="list-style-type: none">• 'Bool'• 'String'• 'Enum'• 'ComboBox'
Example	<pre>% define input parameters inputParam1 = ModelAdvisor.InputParameter; inputParam1.Name = 'Skip font checks.'; inputParam1.Type = 'Bool'; inputParam1.Value = false;</pre>

ModelAdvisor.Task.Value property

Purpose	Status of task
Values	'true' (default) — Initial status of task is enabled 'false' — Initial status of task is disabled
Description	<p>The Value property indicates the initial status of a task—whether it is enabled or disabled.</p> <p>When adding checks as tasks, the Model Advisor uses the task Value property instead of the check Value property.</p>
Examples	<pre>MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1'); MAT1.Value = 'false';</pre>

ModelAdvisor.Check.Visible property

Purpose	Indicate to display or hide check				
Values	'true' (default) 'false'				
Description	The Visible property specifies whether the Model Advisor displays the check. <table><tr><td>'true'</td><td>Display the check</td></tr><tr><td>'false'</td><td>Hide the check</td></tr></table>	'true'	Display the check	'false'	Hide the check
'true'	Display the check				
'false'	Hide the check				
Examples	<pre>% hide all checks that do not belong to Demo group if ~(strcmp(checkCellArray{i}.Group, 'Demo')) checkCellArray{i}.Visible = false; checkCellArray{i}.Value = false; end</pre>				

ModelAdvisor.Task.Visible property

Purpose Indicate to display or hide task

Values 'true' (default) — Display task in the Model Advisor
'false' — Hide task

Description The `Visible` property specifies whether the Model Advisor displays the task.

Caution

When adding checks as tasks, you cannot specify both the task and check `Visible` properties, you must specify one or the other. If you specify both properties, the Model Advisor generates an error when the check `Visible` property is `false`.

Example

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');  
MAT1.Visible = 'false';
```

Block Reference

System Requirements

Purpose

List system requirements in Simulink diagrams

Library

Simulink Verification and Validation

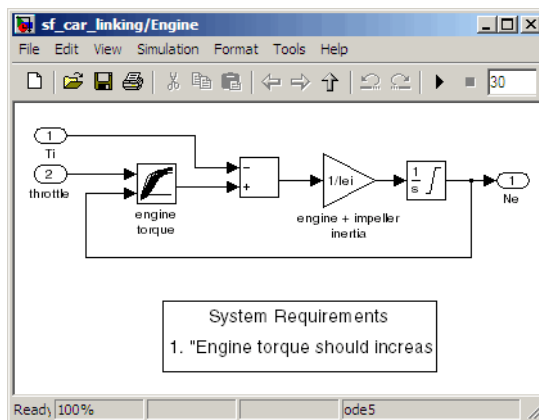
Description



The System Requirements block lists all the system requirements associated with the model or subsystem depicted in the current diagram. It does not list requirements associated with individual blocks in the diagram.

You can place this block anywhere in a diagram. It is not connected to other Simulink blocks. You can only have one System Requirements block in a diagram.

When you drag the System Requirements block from the Library Browser into your Simulink diagram, it is automatically populated with the system requirements, as shown.



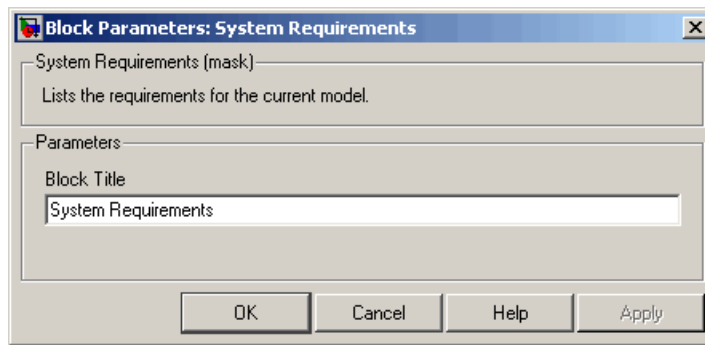
Each of the listed requirements is an active link to the actual requirements document. When you double-click on a requirement name, the associated requirements document opens in its editor window, scrolled to the target location.

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

For more information on using the System Requirements block, see “Using the System Requirements Block in a Model” on page 2-56.

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” on page 14-2
- “DO-178B Checks” on page 14-4
- “IEC 61508 Checks” on page 14-62
- “MathWorks Automotive Advisory Board Checks” on page 14-79
- “Requirements Consistency Checks” on page 14-145

Simulink Verification and Validation Checks

In this section...
“Simulink® Verification and Validation Checks Overview” on page 14-2
“Modeling Standards Checks Overview” on page 14-3

Simulink Verification and Validation Checks Overview

Simulink Verification and Validation checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements, modeling guidelines, or requirements consistency.

The Model Advisor performs a checkout of the Simulink Verification and Validation license when you run the Simulink Verification and Validation checks.

For descriptions of the modeling standards checks, see

- “DO-178B Checks” on page 14-4
- “IEC 61508 Checks” on page 14-62
- “MathWorks Automotive Advisory Board Checks” on page 14-79

For descriptions of the requirements consistency checks, see “Requirements Consistency Checks” on page 14-145.

See Also

- “Consulting the Model Advisor” in the Simulink documentation
- “Simulink Checks” in the Simulink reference documentation
- “Real-Time Workshop Checks” in the Real-Time Workshop documentation

Modeling Standards Checks Overview

Modeling standards checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements or MathWorks Automotive Advisory Board (MAAB) modeling guidelines.

The Model Advisor performs a checkout of the Simulink Verification and Validation license when you run the modeling standards checks.

For descriptions of the modeling standards checks, see

- “DO-178B Checks” on page 14-4
- “IEC 61508 Checks” on page 14-62
- “MathWorks Automotive Advisory Board Checks” on page 14-79

See Also

- Consulting the Model Advisor in the Simulink documentation
- Simulink Checks in the Simulink reference documentation
- Real-Time Workshop Checks in the Real-Time Workshop documentation

DO-178B Checks

In this section...

“DO-178B Checks Overview” on page 14-5

“Check safety-related optimization settings” on page 14-6

“Check safety-related diagnostic settings for solvers” on page 14-10

“Check safety-related diagnostic settings for sample time” on page 14-13

“Check safety-related diagnostic settings for signal data” on page 14-16

“Check safety-related diagnostic settings for parameters” on page 14-19

“Check safety-related diagnostic settings for data used for debugging” on page 14-22

“Check safety-related diagnostic settings for data store memory” on page 14-24

“Check safety-related diagnostic settings for type conversions” on page 14-26

“Check safety-related diagnostic settings for signal connectivity” on page 14-28

“Check safety-related diagnostic settings for bus connectivity” on page 14-30

“Check safety-related diagnostic settings that apply to function-call connectivity” on page 14-32

“Check safety-related diagnostic settings for compatibility” on page 14-34

“Check safety-related diagnostic settings for model initialization” on page 14-36

“Check safety-related diagnostic settings for model referencing” on page 14-38

“Check safety-related model referencing settings” on page 14-41

“Check safety-related code generation settings” on page 14-43

“Check safety-related diagnostic settings for saving” on page 14-50

“Check for model objects that do not link to requirements” on page 14-52

“Check for proper usage of Math blocks” on page 14-53

“Check for proper usage of For Iterator blocks” on page 14-54

In this section...

“Check for proper usage of While Iterator blocks” on page 14-55

“Display model version information” on page 14-57

“Check for proper usage of blocks that compute absolute values” on page 14-58

“Check for proper usage of Relational Operator blocks” on page 14-60

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.

The Model Advisor performs a checkout of the Simulink Verification and Validation license when you run the DO-178B checks.

See Also

- “Consulting the Model Advisor” in the Simulink documentation
- “Simulink Checks” in the Simulink reference documentation
- “Real-Time Workshop Checks” in the Real-Time Workshop documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related optimization settings

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Results and Recommended Actions

Condition	Recommended Action
Block reduction optimization is selected. This optimization can remove blocks from generated code, resulting in requirements with no associated code and violations for traceability requirements. (See DO-178B, Section 6.3.4e—Source code is traceable to low-level requirements.)	Clear the Block reduction check box on the Optimization pane of the Configuration Parameters dialog box or set the parameter <code>BlockReduction</code> to off.
Implementation of logic signals as Boolean data is cleared. Strong data typing is recommended for safety-related code. (See DO-178B, Section 6.3.1e—High-level requirements conform to standards, DO-178B, Section 6.3.2e—Low-level requirements conform to standards, and MISRA C® 2004, Rule 12.6.)	Select Implement logic signals as boolean data (vs. double) on the Optimization pane of the Configuration Parameters dialog box or set the parameter <code>BooleanDataType</code> to on.

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 set the parameter <code>LifeSpan</code> to <code>inf</code>.</p>
<p>The optimization that ignores integer downcasts in folded expressions is selected. This optimization can remove blocks that typecast data from generated code, resulting in incorrect behavior due to overflows of integer data and requirements without associated 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 10.1.)</p>	<p>If you have a Real-Time Workshop license, 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>on</code>.</p>
<p>The optimization that suppresses the generation of initialization code for root-level inports and outports that are set to zero is selected. For safety-related code, you should explicitly initialize all variables. (See DO-178B, Section 6.3.3b—Software architecture is consistent and MISRA C 2004, Rule 9.1.)</p>	<p>If you have a Real-Time Workshop Embedded Coder license, and you are using an ERT-based system target file, 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 selected. For safety-related code, you should explicitly initialize all variables. (See DO-178B, Section 6.3.3b—Software architecture is consistent and MISRA C 2004, Rule 9.1.)</p>	<p>If you have a Real-Time Workshop Embedded Coder license, and you are using an ERT-based system target file, clear the Remove internal data 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 cleared. You must avoid overflows for safety-related code. When this optimization is off and your model includes blocks that disable the Saturate on overflow parameter, the code generator wraps out-of-range values for those blocks. This can result in unreachable and, therefore, untestable code. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 12.11.)</p>	<p>If you have a Real-Time Workshop license, 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>The optimization that suppresses generation of code that guards against division by zero for fixed-point data is selected. You must avoid division-by-zero exceptions in safety-related code. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>If you have a Real-Time Workshop Embedded Coder license, and you are using an ERT-based system target file, 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>

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

See Also

- Optimization Pane in the Simulink graphical user interface documentation
- Optimizing a Model for Code Generation in the Real-Time Workshop documentation
- Tips for Optimizing the Generated Code in the Real-Time Workshop Embedded Coder documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for solvers

Check model configuration for diagnostic settings that apply to solvers and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to solvers are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
<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-related 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 set the parameter AlgebraicLoopMsg to error. Consider breaking such loops explicitly with Unit Delay blocks to ensure that execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.</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-related 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 set the parameter ArtificialAlgebraicLoopMsg to error. Consider breaking such loops explicitly with Unit Delay blocks to ensure that execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.</p>

Condition	Recommended Action
<p>The diagnostic for detecting potential conflict in block execution order is set to none or warning. For safety-related applications, block execution order must be predictable. A model developer needs to know when conflicting block priorities exist. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Block priority violation on the Diagnostics > Solver pane of the Configuration Parameters dialog box or set 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 set the parameter UnknownTs1nhSupMsg to error.</p>
<p>The diagnostic for detecting whether the Simulink software automatically modifies the solver, step size, or simulation stop time is set to none or warning. Such changes can affect the operation of generated code. For safety-related applications, it is better to detect such changes so a model developer can explicitly set the parameters to known values. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Automatic solver parameter selection on the Diagnostics > Solver pane of the Configuration Parameters dialog box or set 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-related applications, it is better to detect name clashes so a model developer can correct them. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set State name clash on the Diagnostics > Solver pane of the Configuration Parameters dialog box or set the parameter StateNameClashWarn to warning.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to solvers and that can impact safety.

See Also

- Diagnostics Pane: Solver in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for sample time

Check model configuration for diagnostic settings that apply to sample time and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to sample times are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
<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-related applications, source blocks should have explicit sample times to prevent incorrect execution sequencing. (See DO-178B, Section 6.3.3e – Software architecture conforms to standards.)</p>	<p>Set Source block specifies -1 sample time on the Diagnostics > Sample Time pane of the Configuration Parameters dialog box or set 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 set the parameter <code>DiscreteInheritContinuousMsg</code> to error.</p>

Condition	Recommended Action
<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 set the parameter <code>MultiTaskRateTransMsg</code> to error.</p>
<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 set 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 set the parameter <code>SigSpecEnsureSampleTimeMsg</code> to error.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to sample time and that can impact safety.

See Also

- [Diagnostics Pane: Sample Time in the Simulink graphical user interface documentation](#)
- [Diagnosing Simulation Errors in the Simulink documentation](#)
- [Radio Technical Commission for Aeronautics \(RTCA\) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard](#)

Check safety-related diagnostic settings for signal data

Check model configuration for diagnostic settings that apply to signal data and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal data are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that specifies how the Simulink software resolves signals associated with <code>Simulink.Signal</code> objects in the MATLAB workspace is set to <code>Explicit</code> and <code>implicit</code> or <code>Explicit</code> and <code>warn implicit</code>. For safety-related applications, model developers should be required to define signal resolution explicitly. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Signal resolution on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>SignalResolutionControl</code> to <code>Explicit</code> only. This provides predictable operation by requiring users to define each signal and block setting that must resolve to <code>Simulink.Signal</code> 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 <code>none</code> or <code>warning</code>. Division by a singular matrix can result in numeric exceptions when executing generated code. This is not acceptable in safety-related systems. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>Set Division by singular matrix on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>CheckMatrixSingularityMsg</code> to <code>error</code>.</p>

Condition	Recommended Action
<p>The diagnostic that detects when the Simulink software cannot infer the data type of a signal during data type propagation is set to none or warning. For safety-related applications, model developers must ensure that all data types are specified correctly. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards, DO-178B and Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set Underspecified data types on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>UnderSpecifiedDataTypeMsg</code> 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 unexpected 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 set the parameter <code>IntegerOverflowMsg</code> to error.</p>
<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-related applications. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>Set Inf or NaN block output on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>SignalInfNanChecking</code> to error.</p>

Condition	Recommended Action
<p>The diagnostic that detects Simulink object names that begin with <code>rt</code> is set to <code>none</code> or <code>warning</code>. This diagnostic prevents name clashes with generated signal names that have an <code>rt</code> prefix. (See DO-178B, Section 6.3.1e – High-level requirements conform to standards, and DO-178B, Section 6.3.2e – Low-level requirements conform to standards.)</p>	<p>Set "rt" prefix for identifiers on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>RTPrefix</code> to <code>error</code>.</p>
<p>The diagnostic that detects simulation range checking is set to <code>none</code> or <code>warning</code>. This diagnostic detects when signals exceed their specified ranges during simulation. Simulink compares the signal values that a block outputs with the specified range and the block data type. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>Set Simulation range checking on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>SignalRangeChecking</code> to <code>error</code>.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to signal data and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for parameters

Check model configuration for diagnostic settings that apply to parameters and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to parameters are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
<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 unexpected 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 set 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 set the parameter ParameterUnderflowMsg to error.</p>

Condition	Recommended Action
<p>The diagnostic that detects when a parameter overflow occurs is set to none or warning. Numeric overflows can result in unexpected application behavior and should be detected and corrected in safety-related applications. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>Set Detect overflow on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>ParameterOverflowMsg</code> to error.</p>
<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 set 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 set the parameter <code>ParameterTunabilityLossMsg</code> to error.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to parameters and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation

- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for data used for debugging

Check model configuration for diagnostic settings that apply to data used for debugging and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to debugging are set optimally for generating code for a safety-related application.

See

- DO-178B, Section 6.3.1e – High-level requirements conform to standards
- DO-178B and Section 6.3.2e – Low-level requirements conform to standards

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that enables model verification blocks is set to <code>Use local settings</code> or <code>Enable all</code>. Such blocks should be disabled because they are assertion blocks, which are for verification only. Model developers should not use assertions in embedded code.</p>	<p>Set Model Verification block enabling on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>AssertControl</code> to <code>Disable All</code>.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to data used for debugging and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for data store memory

Check model configuration for diagnostic settings that apply to data store memory and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to data store memory are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent.

Results and Recommended Actions

Condition	Recommended Action
<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 <code>Enable all as errors</code>. Reading data before it is written can result in use of stale data or data that is not initialized.</p>	<p>Set Detect read before write on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>ReadBeforeWriteMsg</code> to <code>Enable all as errors</code>.</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 <code>Enable all as errors</code>. Writing data after it is read can result in use of stale or incorrect data.</p>	<p>Set Detect write after read on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>WriteAfterReadMsg</code> to <code>Enable all as errors</code>.</p>

Condition	Recommended Action
The diagnostic that detects whether the model attempts to store data in a data store twice in succession in the current time step is set to a value other than <code>Enable all as errors</code> . Writing data twice in one time step can result in unpredictable data.	Set Detect write after write on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>WriteAfterWriteMsg</code> to <code>Enable all as errors</code> .
The diagnostic that detects when one task reads data from a Data Store Memory block to which another task writes data is set to <code>none</code> or <code>warning</code> . Reading or writing data in different tasks in multitask mode can result in corrupted or unpredictable data.	Set Multitask data store on the Diagnostics > Data Validity pane of the Configuration Parameters dialog box or set the parameter <code>MultiTaskDSMsg</code> to <code>error</code> .

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to data store memory and that can impact safety.

See Also

- Diagnostics Pane: Data Validity in the Simulink graphical user interface documentation
- Diagnosing Simulation Errors in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for type conversions

Check model configuration for diagnostic settings that apply to type conversions and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to type conversions are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects Data Type Conversion blocks used where no type conversion is necessary is set to none. The Simulink software might remove unnecessary Data Type Conversion blocks from generated code. This might result in requirements without 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 set 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 the Simulink software automatically converts between vector and matrix dimensions, unintended operations or unpredictable behavior can occur. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)</p>	<p>Set Vector/matrix block input conversion on the Diagnostics > Type Conversion pane of the Configuration Parameters dialog box or set the parameter VectorMatrixConversionMsg to error.</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 set the parameter Int32ToFloatConvMsg to warning.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to type conversions and that can impact safety.

See Also

- Diagnostics Pane: Type Conversion in the Simulink graphical user interface documentation
- Data Type Conversion block in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for signal connectivity

Check model configuration for diagnostic settings that apply to signal connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal connectivity are set optimally for generating code for a safety-related application.

See

- DO-178B, Section 6.3.1e – High-level requirements conform to standards
- DO-178B, Section 6.3.2e – Low-level requirements conform to standards

Results and Recommended Actions

Condition	Recommended Action
<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.</p>	<p>Set Signal label mismatch on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set the parameter <code>SignalLabelMismatchMsg</code> 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.</p>	<p>Set Unconnected block input ports on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set the parameter <code>UnconnectedInputMsg</code> to error.</p>

Condition	Recommended Action
The diagnostic that detects when the model contains a block with an unconnected output signal is set to none or warning. This must be detected because dead code can result from unconnected block output signals.	Set Unconnected block output ports on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set the parameter UnconnectedOutputMsg to error.
The diagnostic that detects unconnected signal lines and unmatched Goto or From blocks is set to none or warning. This error must be detected because code is not generated for unconnected lines.	Set Unconnected line on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set the parameter UnconnectedLineMsg to error.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to signal connectivity and that can impact safety.

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- Signal Basics in the Simulink documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for bus connectivity

Check model configuration for diagnostic settings that apply to bus connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to bus connectivity are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent.

Results and Recommended Actions

Condition	Recommended Action
<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.</p>	<p>Set Unspecified bus object at root Output block on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set 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.</p>	<p>Set Element name mismatch on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set 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 the Simulink software automatically converts a muxed signal to a bus, it is possible for</p>	<ul style="list-style-type: none"> • Set Mux blocks used to create bus signals on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box to error, or set the parameter <code>StrictBusMsg to ErrorOnBusTreatedAsVector</code>.

Condition	Recommended Action
<p>an unintended operation or unpredictable behavior to occur.</p>	<ul style="list-style-type: none"> • Set “Bus signal treated as vector” on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box to error, or the parameter <code>StrictBusMsg</code> to <code>ErrorOnBusTreatedAsVector</code>. <p>You can use the Model Advisor or the <code>sl_replace_mux</code> utility function to replace all Mux blocks used as bus creators with a Bus Creator block.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to bus connectivity and that can impact safety.

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- `Simulink.Bus` in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings that apply to function-call connectivity

Check model configuration for diagnostic settings that apply to function-call connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to function-call connectivity are set optimally for generating code for a safety-related application.

DO-178B, Section 6.3.3b – Software architecture is consistent

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects incorrect use of a function-call subsystem is set to none or warning. If this condition is undetected, incorrect code might be generated.	Set Invalid function-call connection on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set the parameter <code>InvalidFcnCallConMsg</code> to error.
The diagnostic that specifies whether the Simulink software has to compute inputs of a function-call subsystem directly or indirectly while executing the subsystem is set to <code>Use local settings</code> or <code>Disable all</code> . This diagnostic detects unpredictable data coupling between a function-call subsystem and the inputs of the subsystem in the generated code.	Set Context-dependent inputs on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box or set the parameter <code>FcnCallInpInsideContextMsg</code> to <code>Enable all</code> .

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to function-call connectivity and that can impact safety.

See Also

- Diagnostics Pane: Connectivity in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for compatibility

Check model configuration for diagnostic settings that affect compatibility and that might impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to compatibility are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA C 2004, Rule 9.1.

Results and Recommended Actions

Condition	Recommended Action
<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.</p>	<p>Set S-function upgrades needed on the Diagnostics > Compatibility pane of the Configuration Parameters dialog box or set the parameter <code>SFcnCompatibilityMsg</code> to error.</p>

Action Results

Clicking **Modify Settings** configures model diagnostic settings that affect compatibility and that might impact safety.

See Also

- Diagnosing Simulation Errors in the Simulink documentation
- Diagnostics Pane: Compatibility in the Simulink graphical user interface documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for model initialization

In the model configuration, check diagnostic settings that affect model initialization and might impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to initialization are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA C 2004, Rule 9.1.

Analysis Results and Recommended Actions

Condition	Recommended Action
<p>The Check undefined subsystem initial output diagnostic is cleared. This diagnostic specifies whether the Simulink software displays a warning if the model contains a conditionally executed subsystem, in which a block with a specified initial condition drives an Outport block with an undefined initial condition. A conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.</p>	<p>In the Configuration Parameters dialog box, on the Diagnostics > Data Validity pane, select Check undefined subsystem initial output or set the parameter <code>CheckSSInitialOutputMsg</code> to on.</p>

Condition	Recommended Action
<p>The diagnostic that detects potential initial output differences from earlier releases is cleared. A conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.</p>	<p>In the Configuration Parameters dialog box, on the Diagnostics > Compatibility pane, select Check preactivation output of execution context or set the parameter <code>CheckExecutionContextPreStartOutputMsg</code> to on.</p>
<p>The diagnostic that detects potential output differences from earlier releases is cleared. 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.</p>	<p>In the Configuration Parameters dialog box, on the Diagnostics > Compatibility pane, select Check runtime output of execution context or set the parameter <code>CheckExecutionContextRuntimeOutputMsg</code> to on.</p>

Action Results

To configure the diagnostic settings that affect model initialization and that might impact safety, click **Modify Settings**.

See Also

- “Diagnosing Simulation Errors” in the Simulink documentation
- “Diagnostics Pane: Data Validity” in the Simulink graphical user interface documentation
- For information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard, Radio Technical Commission for Aeronautics (RTCA)

Check safety-related diagnostic settings for model referencing

Check model configuration for diagnostic settings that apply to model referencing and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to model referencing are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects a mismatch between the version of the model that creates or refreshes a Model block and the current version of the referenced model is set to error or warning. The detection occurs during load and update operations. When you get the latest version of the referenced model from the software configuration management system, rather than an older version that was used in a previous simulation, if this diagnostic is set to error, the simulation is aborted. If the diagnostic is set to warning, a warning message is issued. To resolve the issue, the user must resave the model being simulated, which may not be the desired action. (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 set the parameter <code>ModelReferenceVersionMismatchMessage</code> to none.</p>

Condition	Recommended Action
<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 set the parameter <code>ModelReferenceIOMismatchMessage</code> to error.</p>
<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 set the parameter <code>ModelReferenceCSMismatchMessage</code> to warning.</p>
<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, the Simulink software might automatically insert hidden blocks into the model to correct the condition. The hidden blocks can result in generated code that has no traceable requirements. Setting the diagnostic to error forces model developers to correct the referenced models manually. (See DO-178B,</p>	<p>Set Invalid root Inport/Outport block connection on the Diagnostics > Model Referencing pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceIOMessage</code> to error.</p>

Condition	Recommended Action
Section 6.3.3b – Software architecture is consistent.)	
The diagnostic that detects whether To Workspace or Scope blocks are logging data in a referenced model is set to none or warning. Data logging is not supported for To Workspace and Scope blocks in referenced models. (See DO-178B, Section 6.3.1d – High-level requirements are verifiable and DO-178B, Section 6.3.2d – Low-level requirements are verifiable.)	Set Unsupported data logging on the Diagnostics > Model Referencing pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceDataLoggingMessage</code> to error. To log data, remove the blocks and log the referenced model signals. For more information, see “Logging Referenced Model Signals”.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to model referencing and that can impact safety.

See Also

- Diagnosing Simulation Errors in the Simulink documentation
- Diagnostics Pane: Model Referencing in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard
- “Logging Referenced Model Signals” in the Simulink documentation

Check safety-related model referencing settings

Check model configuration for model referencing settings that can impact safety.

Description

This check verifies that model configuration parameters for model referencing are set optimally for generating code for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
<p>The referenced model is configured such that its target is rebuilt whenever you update, simulate, or generate code for the model, or if the Simulink software detects any changes in known dependencies. These configuration settings can result in unnecessary regeneration of the code, resulting in changing only the date of the file and slowing down the build process when using model references. (See DO-178B, Section 6.3.1b – High-level requirements are accurate and consistent and DO-178B, Section 6.3.2b – Low-level requirements are accurate and consistent.)</p>	<p>Set Rebuild options on the Model Referencing pane of the Configuration Parameters dialog box or set 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 if targets require rebuild</code>. For safety-related applications, an error should alert model developers that the parent and referenced models are inconsistent. This diagnostic parameter is available only if Rebuild options is set to <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 set the parameter <code>CheckModelReferenceTargetMessage</code> to <code>Error if targets require rebuild</code>.</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 set the parameter <code>ModelReferencePassRootInputsByReference</code> to off.</p>
<p>The model is configured to minimize algebraic loop occurrences. This configuration is incompatible with the recommended setting of Single output/update function for embedded systems code. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set Minimize algebraic loop occurrences on the Model Referencing pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceMinAlgLoopOccurrences</code> to off.</p>

Action Results

Clicking **Modify Settings** configures model referencing settings that can impact safety.

See Also

- Model Dependencies in the Simulink documentation
- Model Referencing Pane in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related code generation settings

Check model configuration for code generation settings that can impact safety.

Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The option to include comments in the generated code is cleared. 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.)	Select Include comments on the Real-Time Workshop > Comments > pane of the Configuration Parameters dialog box or set the parameter <code>GenerateComments</code> to on.
The option to include comments that describe the code for blocks is cleared. 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.)	Select Simulink block / Stateflow object comments on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or set the parameter <code>SimulinkBlockComments</code> to on.
The option to include comments that describe the code for blocks eliminated from a model is cleared. 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.)	Select Show eliminated blocks on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or set the parameter <code>ShowEliminatedStatement</code> to on.

Condition	Recommended Action
<p>The option to include the names of parameter variables and source blocks as comments in the model parameter structure declaration in <i>model_prm.h</i> is cleared. 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>Select Verbose comments for SimulinkGlobal storage class on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or set the parameter <code>ForceParamTrailComments</code> to on.</p>
<p>The option to include requirement descriptions assigned to Simulink blocks as comments is cleared. 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>Select Requirements in block comments on the Real-Time Workshop > Comments pane of the Configuration Parameters dialog box or set the parameter <code>ReqsInCode</code> to on.</p>
<p>The option to generate nonfinite data and operations is selected. Support for nonfinite numbers is inappropriate for real-time embedded 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>Clear Support: non-finite numbers on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>SupportNonFinite</code> to off.</p>
<p>The option to generate and maintain integer counters for absolute and elapsed time is selected. Support for absolute time is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Clear Support: absolute time on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>SupportAbsoluteTime</code> to off.</p>

Condition	Recommended Action
<p>The option to generate code for blocks that use continuous time is selected. Support for continuous time is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Clear Support: continuous time on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>SupportContinuousTime</code> to off.</p>
<p>The option to generate code for noninlined S-functions is selected. This option requires support of nonfinite numbers, which is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Clear Support: non-inlined S-functions on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>SupportNonInlinedSFcns</code> to off.</p>
<p>The option to generate model function calls compatible with the main program module of the GRT target is selected. This option is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Clear GRT compatible call interface on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>GRTInterface</code> to off.</p>

Condition	Recommended Action
<p>The option to generate the <i>model_update</i> function is cleared. 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>Select Single output/update function on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>CombineOutputUpdateFcns</code> to on.</p>
<p>The option to generate the <i>model_terminate</i> function is selected. This function deallocates dynamic memory, which is not appropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Clear Terminate function required on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>IncludeMdlTerminateFcn</code> to off.</p>
<p>The option to log or monitor error status is cleared. If you do not select this option, the Real-Time Workshop product generates extra code that might not be reachable for testing. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Select Suppress error status in real-time model data structure on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>SuppressErrorStatus</code> to on.</p>

Condition	Recommended Action
<p>MAT-file logging is selected. 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>Clear MAT-file logging on the Real-Time Workshop > Interface pane of the Configuration Parameters dialog box or set the parameter <code>MatFileLogging</code> to off.</p>
<p>The option that specifies the style for parenthesis usage is set to <code>Minimum (Rely on C/C++ operators precedence)</code> or to <code>Nominal (Optimize for readability)</code>. For safety-related applications, explicitly specify precedence with parentheses. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer, DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer, and MISRA C 2004, Rule 12.1.)</p>	<p>Set Parenthesis level on the Real-Time Workshop > Code pane of the Configuration Parameters dialog box or set the parameter <code>ParenthesesLevel</code> to <code>Maximum (Specify precedence with parentheses)</code>.</p>
<p>The option that specifies whether to preserve operand order is cleared. 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>Select Preserve operand order in expression on the Real-Time Workshop > Code pane of the Configuration Parameters dialog box or set the parameter <code>PreserveExpressionOrder</code> to on.</p>

Condition	Recommended Action
<p>The option that specifies whether to preserve empty primary condition expressions in <code>if</code> statements is cleared. 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>Select Preserve condition expression in if statement on the Real-Time Workshop > Code pane of the Configuration Parameters dialog box or set 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. Use of this option assists with minimizing code differences between file versions, decreasing the effort to perform code reviews. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</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>

Action Results

Clicking **Modify Settings** configures model code generation settings that can impact safety.

Limitations

This check requires a Real-Time Workshop Embedded Coder license and an ERT-based system target file.

See Also

- Real-Time Workshop Pane: Comments in the Real-Time Workshop reference documentation
- Real-Time Workshop Pane: Symbols in the Real-Time Workshop reference documentation

- Real-Time Workshop Pane: Interface in the Real-Time Workshop reference documentation
- Real-Time Workshop Pane: Code Style in the Real-Time Workshop Embedded Coder reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check safety-related diagnostic settings for saving

Check model configuration for diagnostic settings that apply to saving model files

Description

This check verifies that model configuration parameters are set optimally for saving a model for a safety-related application.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether a model contains disabled library links before the model is saved is set to none or warning. If this condition is undetected, incorrect code might be generated. (See DO-178B, Section 6.3.3b - Software architecture is consistent.)	Set Block diagram contains disabled library links on the Diagnostics > Saving> pane of the Configuration Parameters dialog box or set the parameter <code>SaveWithDisabledLinkMsg</code> to error.
The diagnostic that detects whether a model contains library links that are using parameters not in a mask before the model is saved is set to none or warning. If this condition is undetected, incorrect code might be generated. (See DO-178B, Section 6.3.3b - Software architecture is consistent.)	Set Block diagram contains parameterized library links on the Diagnostics > Saving> pane of the Configuration Parameters dialog box or set the parameter <code>SaveWithParameterizedLinkMsg</code> to error.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to saving a model file.

See Also

- Disabling Links to Library Blocks in the Simulink documentation
- Identifying Disabled Library Links in the Simulink documentation
- Saving a Model in the Simulink documentation

- Model Parameters in the Simulink documentation
- Diagnostics Pane: Saving in the Simulink documentation

Check for model objects that do not link to requirements

Check whether Simulink blocks and Stateflow objects link to a requirements document.

Description

This check verifies whether Simulink blocks and Stateflow objects link to a document containing engineering requirements for traceability.

Analysis Results and Recommended Actions

Condition	Recommended Action
Blocks do not link to a requirements document. (See DO-178B, Section 6.3.1f - High-level requirements trace to system requirements, Section 6.3.2f - Low-level requirements trace to high-level requirements.)	Link to requirements document. See .

Limitations

When you run this check, the Model Advisor does not follow library links or look under masks. The Model Advisor reviews all top-level blocks in the system.

Tip

Run this check from the top model or subsystem that you want to check.

See Also

Chapter 2, “Managing Model Requirements”

Check for proper usage of Math blocks

Check whether math operators require nonfinite number support.

Description

This check verifies that Math Function blocks do not use math operations that need nonfinite number support with real-time embedded targets.

Analysis Results and Recommended Actions

Condition	Recommended Action
Math Function blocks using <code>log</code> (natural logarithm), <code>log10</code> (base 10 logarithm), and <code>rem</code> (Remainder) operators that require nonfinite number support. (See DO-178B, Section 6.3.1g - Algorithms are accurate, Section 6.3.2g - Algorithms are accurate, and MISRA C 2004, Rule 21.1)	<p>When using the Math Function block with a <code>log</code> or <code>log10</code> function, you must protect the input to the block in the model such that it is not less than or equal to zero. Otherwise, the output can produce a <code>NaN</code> or <code>Inf</code> and result in a run-time error in the generated code.</p> <p>When using the Math Function block with a <code>rem</code> function, you must protect the second input to the block such that it is not equal to zero. Otherwise the output can produce a <code>Inf</code> or <code>-Inf</code> and result in a run-time error in the generated code.</p>

Tips

With embedded systems, you must take care when using blocks that could produce nonfinite outputs such as `NaN`, `Inf` or `-Inf`. Your design must protect the inputs to these blocks in order to avoid run-time errors in the embedded system.

See Also

Math Function block in the Simulink documentation

Check for proper usage of For Iterator blocks

Check for For Iterator blocks that have variable loops.

Description

This check verifies that a model does not use variable loops with For Iterator blocks.

See

- DO-178B Section 6.3.1e – High-level requirements conform to standards
- DO-178B Section 6.3.2e – Low-level requirements conform to standards
- MISRA C 2004, Rule 13.6

Results and Recommended Actions

Condition	Recommended Action
<p>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.</p>	<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 in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check for proper usage of While Iterator blocks

Check for While Iterator blocks that cause infinite loops.

Description

This check verifies that a model does not include infinite loops with While Iterator blocks.

See

- DO-178B Section 6.3.1e – High-level requirements conform to standards
- DO-178B Section 6.3.2e – Low-level requirements conform to standards
- MISRA C 2004, Rule 21.1

Results and Recommended Actions

Condition	Recommended Action
<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.</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 in the Simulink reference documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Display model version information

Display model version information in your report.

Description

This check displays the following information for the current model:

- Version number
- Author
- Date
- Model checksum

Results and Recommended Actions

Condition	Recommended Action
Could not retrieve model version and checksum information.	This summary is provided for your information. No action is required.

See Also

- Validating Generated Code in the Real-Time Workshop documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check for proper usage of blocks that compute absolute values

Check for absolute value blocks that have unreachable code or produce overflows.

Description

This check verifies whether the model includes a block that attempts to compute the absolute value of a Boolean or unsigned integer value.

See

- DO-178B Section 6.3.1d – High-level requirements are verifiable
- DO-178B Section 6.3.2d – Low-level requirements are verifiable
- DO-178B Section 6.3.1g – Algorithms are accurate
- DO-178B Section 6.3.2g – Algorithms are accurate
- MISRA C 2004, Rule 14.1
- MISRA C 2004, Rule 21.1

Results and Recommended Actions

Condition	Recommended Action
<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 might result in code that is unreachable and cannot be tested. • Computes an absolute value of a signed integer and Saturate on integer overflow is not selected for that block. Taking the absolute value of full scale negative integer value results in an overflow. 	<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 in the Simulink reference documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

Check for proper usage of Relational Operator blocks

Check for relational operator blocks that compare data types or equate floating-point types.

Description

This check verifies that a model does not use the == or ~= operator with a relational operator block to compare floating-point signals.

See

- DO-178B Section 6.3.1g – Algorithms are accurate
- DO-178B Section 6.3.2g – Algorithms are accurate
- MISRA C 2004, Rule 12.6
- MISRA C 2004, Rule 13.3

Results and Recommended Actions

Condition	Recommended Action
The model includes a relational operator block that uses the == or ~= operator to compare floating-point signals. Because of floating-point precision issues, the use of these operators on floating-point signals is unreliable.	Change the data type of the signal or rework the model to eliminate the need to use the relational operator block with the == or ~= operator.

See Also

Descriptions of the following blocks in the Simulink reference documentation

- Relational Operator block in the Simulink reference documentation
- Compare To Constant block in the Simulink documentation
- Compare To Zero block in the Simulink documentation
- Detect Change block in the Simulink documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

IEC 61508 Checks

In this section...

“IEC 61508 Checks Overview” on page 14-62

“Display model metrics and complexity report” on page 14-64

“Check for unconnected objects” on page 14-65

“Check for fully defined interface” on page 14-66

“Check for questionable constructs” on page 14-68

“Check usage of Stateflow constructs” on page 14-70

“Check for model objects that do not link to requirements” on page 14-73

“Display configuration management data” on page 14-74

“Check usage of Simulink constructs” on page 14-75

IEC 61508 Checks Overview

IEC 61508 checks facilitate designing and troubleshooting Simulink models and subsystems and the code that you generate from it for applications that need to comply with IEC 61508-3.

The Model Advisor performs a checkout of the Simulink Verification and Validation license when you run the IEC 61508 checks.

Tips

If your model uses model referencing, run the IEC 61508 checks on all referenced models before running them on the top-level model.

See Also

- IEC 61508–3 Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 3: Software requirements
- Developing Models and Code That Comply with the IEC 16508 Standard in the Real-Time Workshop Embedded Coder documentation
- “Consulting the Model Advisor” in the Simulink documentation

- “Simulink Checks” in the Simulink reference documentation
- “Real-Time Workshop Checks” in the Real-Time Workshop documentation

Display model metrics and complexity report

Display number of elements and name, level, and depth of subsystems for the model or subsystem.

Description

The IEC 61508 standard recommends the usage of size and complexity metrics to assess the software under development. This check provides model metrics information for the model. The provided information can be used to inspect whether the size or complexity of the model or subsystem exceeds given limits. The check displays:

- A block count for each Simulink block type contained in the given model.
- The maximum subsystem depth of the given model.
- A count of Stateflow constructs in the given model (if applicable).
- Name, level, and depth of the subsystems contained in the given model (if applicable).

See IEC 61508-3, Table A.9 (5) – Software complexity metrics.

Results and Recommended Actions

Condition	Recommended Action
N/A	This summary is provided for your information. No action is required.

See Also

- `sldiagnostics` in the Simulink documentation
- “Cyclomatic Complexity” in the Stateflow documentation

Check for unconnected objects

Identify unconnected lines, input ports, and output ports in the model.

Description

Unconnected objects are likely to cause problems propagating signal attributes such as data, type, sample time, and dimensions.

Ports connected to Ground or Terminator blocks pass this check.

See IEC 61508-3, Table A.3 (3) — Language subset.

Results and Recommended Actions

Condition	Recommended Action
There are unconnected lines, input ports, or output ports in the model or subsystem.	<ul style="list-style-type: none">• Double-click an element in the list of unconnected items to locate the item in the model diagram.• Properly connect the objects identified in the results.

See Also

“Working with Signals” in the Simulink documentation

Check for fully defined interface

Identify root model Inport blocks that do not have fully defined attributes.

Description

Using root model Inport blocks that do not have fully define dimensions, sample time, or data type can lead to undesired simulation results. Simulink back-propagates dimensions, sample times, and data types from downstream blocks unless you explicitly assign these values.

See IEC 61508-3, Table B.9 (5) – Fully defined interface.

Results and Recommended Actions

Condition	Recommended Action
The model has root-level Inport blocks that have undefined attributes, such as an inherited sample time, data type, or port dimension.	Explicitly define all root-level Inport block attributes identified in the results. Double-click an element from the list of underspecified items to locate the condition.

Tips

The following configurations pass this check:

- Inport blocks with inherited port dimensions in conjunction with the usage of bus objects
- Inport blocks with automatically inherited data types in conjunction with bus objects
- Inport blocks with inherited sample times in conjunction with the **Periodic sample time constraint** menu set to Ensure sample time independent

See Also

- Working with Data Types in the Simulink documentation
- Determining Output Signal Dimensions in the Simulink documentation

- Specifying Sample Time in the Simulink documentation

Check for questionable constructs

Identify blocks not supported by code generation or not recommended for deployment.

Description

This check partially identifies model constructs that are not suited for code generation or not recommended for production code generation as identified in the Simulink Block Support tables for Real-Time Workshop and Real-Time Workshop Embedded Coder. If you are using blocks with support notes for code generation, review the information and follow the given advice.

See IEC 61508-3, Table A.3 (3) – Language subset.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for code generation.	Consider replacing the blocks listed in the results. Double-click an element from the list of questionable items to locate condition.
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Double-click an element from the list of questionable items to locate condition.
The model or subsystem contains Gain blocks whose value equals 1.	If you are using Gain blocks as buffers, consider replacing them with Signal Conversion blocks. Double-click an element from the list of questionable items to locate condition.

Limitation

This check might not identify all instances of noncompliance with the Real-Time Workshop and Real-Time Workshop Embedded Coder tables.

See Also

- tables in the Real-Time Workshop documentation for Real-Time Workshop and Real-Time Workshop Embedded Coder
- “Developing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation

Check usage of Stateflow constructs

Identify usage of Stateflow constructs that might impact safety.

Description

This check identifies instances of Stateflow software being used in a way that can impact an application's safety, including

- Use of strong data typing
- Port name mismatches
- Scope of data objects and events
- Formatting of state action statements

See

- IEC 61508-3, Table A.3 (2) – Strongly typed programming language
- IEC 61508-3, Table A.3 (3) – Language subset
- IEC 61508-3, Table B.9 (2) – Information hiding/encapsulation
- MISRA C 2004, Rule 10.1
- MISRA C 2004, Rule 10.2
- MISRA C 2004, Rule 10.3
- MISRA C 2004, Rule 10.4
- MAAB Control Algorithm Modeling Guidelines, db_0122: Stateflow and Simulink interface signals and parameters
- MAAB Control Algorithm Modeling Guidelines, db_0123: Stateflow port names
- MAAB Control Algorithm Modeling Guidelines, db_0125: Scope of internal signals and local auxiliary variables
- MAAB Control Algorithm Modeling Guidelines, db_0126: Scope of events
- MAAB Control Algorithm Modeling Guidelines, jc_0501: Format of entries in a State block

Results and Recommended Actions

Condition	Recommended Action
A Stateflow chart is not configured for strong data typing on boundaries between a Simulink model and the Stateflow chart.	Enable the option Use Strong Data Typing with Simulink I/O for the Stateflow chart. When you enable this option, the Stateflow chart accepts input signals of any data type that Simulink models support, provided that the type of the input signal matches the type of the corresponding Stateflow input data object.
Signals have names that differ from those of their corresponding Stateflow ports.	<ul style="list-style-type: none"> • Check whether the ports are connected properly and, if not, correct the connections. • Change the names of the signals or the Stateflow ports so that the names match.
Events are not defined in the Stateflow hierarchy at the chart level or below.	Define events at the chart level or below.
Local data is not defined in the Stateflow hierarchy at the chart level or below.	Define local data at the chart level or below.
<p>A new line is missing from a state action after</p> <ul style="list-style-type: none"> • An entry (en), during (du), or exit (ex) statement • The semicolon (;) at the end of an assignment statement 	Add missing new lines.

See Also

See the following topics in the Stateflow documentation

- “Strong Data Typing with Simulink I/O”
- “Property Fields”
- “Defining Events”
- “Defining Data”
- “Labeling States”

Check for model objects that do not link to requirements

Check whether Simulink blocks and Stateflow objects link to a requirements document.

Description

This check verifies whether Simulink blocks and Stateflow objects link to a document containing engineering requirements for traceability.

Analysis Results and Recommended Actions

Condition	Recommended Action
Blocks do not link to a requirements document. (See IEC 61508-3, Table A.1 (1) 'Computer-aided specification tools', Table A.2 (8) 'Computer-aided specification tools', and Table A.8 (1) 'Impact analysis'.)	Link to requirements document. See .

Limitations

When you run this check, the Model Advisor does not follow library links or look under masks. The Model Advisor reviews all top-level blocks in the system.

Tip

Run this check from the top model or subsystem that you want to check.

See Also

Chapter 2, "Managing Model Requirements"

Display configuration management data

Display model configuration and checksum information.

Description

This informer check displays the following information for the current model:

- Model version number
- Model author
- Date
- Model checksum

See IEC 61508-3, Table A.8 (5) – Software configuration management.

Results and Recommended Actions

Condition	Recommended Action
Could not retrieve model version and checksum information.	This summary is provided for your information. No action is required.

See Also

- “How Simulink Helps You Manage Model Versions” in the Simulink documentation
- Model Change Log in the Simulink Report Generator documentation
- Simulink.BlockDiagram.getChecksum in the Simulink documentation
- Simulink.SubSystem.getChecksum in the Simulink documentation

Check usage of Simulink constructs

Identify usage of Simulink constructs that might impact safety.

Description

Blocks that you use incorrectly can result in unreachable code, incorrect or unpredictable results, infinite loops, and unpredictable execution times in generated code.

This check inspects your model for proper usage of:

- Abs blocks
- Blocks that compute relational operators including Relational Operator, Compare To Constant, Compare To Zero, and Detect Change blocks
- While Iterator blocks
- For Iterator blocks

See

- IEC 61508-3, Table A.3 (2) – Strongly typed programming language
- IEC 61508-3, Table A.3 (3) – Language subset
- IEC 61508-3, Table A.4 (3) – Defensive programming
- IEC 61508-3, Table B.8 (3) – Control Flow Analysis
- MISRA C 2004, Rule 13.3
- MISRA C 2004, Rule 13.6
- MISRA C 2004, Rule 14.1
- MISRA C 2004, Rule 21.1

Results and Recommended Actions

Condition	Recommended Action
<p>The model or subsystem contains an Abs block that is operating on a Boolean or an unsigned input data type. This condition results in unreachable simulation pathways through the model and might result in unreachable code.</p>	<ul style="list-style-type: none"> • Change the input of the Abs block to a signed input type. • Remove the Abs from the model.
<p>The model or subsystem contains an Abs block that is operating on a signed integer value, and the Saturate on integer overflow check box is cleared. For signed data types, the absolute value of the most negative value is problematic since it is not representable by the data type. This condition results in an overflow in the generated code.</p>	<p>Select the Saturate on integer overflow check box of the specified Abs blocks.</p>
<p>The model or subsystem contains a block computing a relational operator that is operating on different data types. The condition can lead to unpredictable results in the generated code.</p>	<p>For the specified blocks, use common data types as inputs.</p>
<p>The model or subsystem contains a block computing a relational operator that is not generating Boolean data as its output. This condition violates strong data typing rules and can lead to unpredictable results in the generated code.</p>	<p>Set the Output data type to boolean in the Block Parameters > Signal Attributes pane for the specified blocks.</p>

Condition	Recommended Action
<p>The model or subsystem contains a block computing a relational operator that uses the == or ~= operator to compare floating-point signals. The use of these operators on floating-point signals is unreliable and unpredictable because of floating-point precision issues, and can lead to unpredictable results in the generated code.</p>	<p>For the specified blocks, do one of the following:</p> <ul style="list-style-type: none"> • Change the signal data type. • Rework the model to eliminate the need to use == or ~= operators on floating-point signals.
<p>The model or subsystem contains a While Iterator block that has unlimited iterations. This condition can lead to infinite loops in the generated code.</p>	<p>For the specified While Iterator blocks:</p> <ul style="list-style-type: none"> • Set the Maximum number of iterations (-1 for unlimited) parameter to a positive integer value. • Consider selecting the Show iteration number port check box and observe the iteration value during simulation.
<p>The model or subsystem contains a For Iterator block that has variable iterations. This condition can lead to unpredictable execution times or infinite loops in the generated code.</p>	<p>For the specified For Iterator blocks, do one of the following:</p> <ul style="list-style-type: none"> • Set the Iteration limit source parameter to <code>internal</code>. • If the Iteration limit source parameter must be <code>external</code>, use a Constant, Probe, or Width block as the source. • Clear the Set next i (iteration variable) externally check box. • Consider selecting the Show iteration variable check box and

Condition	Recommended Action
	observe the iteration value during simulation.

See Also

Descriptions of the following blocks in the Simulink reference documentation:

- Abs block
- Relational Operator block
- Compare To Constant block
- Compare To Zero block
- Detect Change block
- While Iterator block
- For Iterator block

MathWorks Automotive Advisory Board Checks

In this section...

“MathWorks Automotive Advisory Board Checks Overview” on page 14-81

“Check for difference in font and font sizes” on page 14-82

“Check transition orientations in flow charts” on page 14-84

“Check for display of nondefault block attributes” on page 14-85

“Check for proper labeling on signal lines” on page 14-86

“Check for propagated labels on signal lines” on page 14-88

“Check default transition placement in Stateflow charts” on page 14-90

“Check setting Stateflow graphical function return value” on page 14-91

“Check for blocks not using one-based indexing” on page 14-92

“Check for invalid file names” on page 14-94

“Check for invalid model directory names ” on page 14-96

“Check for blocks that are not discrete ” on page 14-97

“Check for prohibited sink blocks” on page 14-98

“Check for invalid port positioning and configuration” on page 14-99

“Check for mismatches between names of ports and corresponding signals” on page 14-101

“Check whether block names do not appear below blocks” on page 14-102

“Check for systems that mix primitive blocks and subsystems” on page 14-103

“Check whether model has unconnected block input ports, output ports, or signal lines” on page 14-105

“Check for improperly positioned Trigger and Enable blocks” on page 14-106

“Check whether annotations have drop shadows” on page 14-107

“Check whether tunable parameters specify expressions, data type conversions, or indexing operations” on page 14-108

In this section...

“Check whether Stateflow events are defined at the chart level or below” on page 14-110

“Check whether Stateflow data objects with local scope are defined at the chart level or below” on page 14-111

“Check interface signals and parameters” on page 14-112

“Check for exclusive states, default states, and substate validity” on page 14-113

“Check optimization parameters for Boolean data types” on page 14-115

“Check model diagnostic settings” on page 14-116

“Check the display attributes of block names” on page 14-119

“Check icon display attributes for port blocks” on page 14-120

“Check whether subsystem block names include invalid characters” on page 14-121

“Check whether Inport and Outport block names include invalid characters” on page 14-123

“Check whether signal line names include invalid characters” on page 14-125

“Check whether block names include invalid characters” on page 14-127

“Check Trigger and Enable block port names” on page 14-129

“Check for Simulink diagrams that have nonstandard appearance attributes” on page 14-130

“Check visibility of port block names” on page 14-133

“Check for direction of subsystem blocks” on page 14-135

“Check for proper position of constants used in Relational Operator blocks” on page 14-136

“Check for entry format in state blocks” on page 14-137

“Check for use of tunable parameters in Stateflow” on page 14-139

“Check for proper use of Switch blocks” on page 14-140

“Check for proper use of signal buses and Mux block usage” on page 14-141

In this section...

“Check for mismatches between Stateflow ports and associated signal names” on page 14-143

“Check for proper scope of From and Goto blocks” on page 14-144

MathWorks Automotive Advisory Board Checks Overview

MathWorks Automotive Advisory Board (MAAB) checks facilitate designing and troubleshooting models from which code is generated for automotive applications.

The Model Advisor performs a checkout of the Simulink Verification and Validation license when you run the MAAB checks.

See Also

- “Consulting the Model Advisor” in the Simulink documentation
- “Simulink Checks” in the Simulink reference documentation
- “Real-Time Workshop Checks” in the Real-Time Workshop documentation
- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation
- The MathWorks Automotive Advisory Board on the MathWorks Web site, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

Check for difference in font and font sizes

Check for difference in font and font sizes.

Description

With the exception of free text annotations within a model, text elements, such as block names, block annotations, and signal labels, must have the same font style and font size. Select a font style and font size that is legible and portable (convertible between platforms), such as Arial or Helvetica 12 point.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline db_0043: Simulink font and font size.

Input Parameters

Font Name

Apply the specified font to all text elements. Available fonts include Helvetica (default), Arial, Arial Black, Mangal, or Modern.

Font Size

Apply the specified font size to all text elements. Available sizes include -1, 6, 8, 9, 10 (default), 12, 14, 16, 18, 20, 22, and 24.

Font Angle

Apply the specified font angle to all text elements. Available angles include auto (default), normal, italic, and oblique.

Font Weight

Apply the specified font weight to all text elements. Available weights include auto (default), normal, light, demi , and bold.

Results and Recommended Actions

Condition	Recommended Action
The fonts or font sizes for text elements in the model are not consistent or portable.	Specify values for the font parameters and click Modify all Fonts , or manually change the fonts and font sizes of text elements in the model such that they are consistent and portable.

Action Results

Clicking **Modify all Fonts** changes the font and font size of all text elements in the model according to the values you specify for the font parameters.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check transition orientations in flow charts

Check transition orientations in flow charts.

Description

The following rules apply to transitions in flow charts:

- Draw transition conditions horizontally.
- Draw transitions with a condition action vertically.

Loop constructs are exceptions to these rules.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0132: Transitions in Flowcharts.

Results and Recommended Actions

Condition	Recommended Action
The model includes a transition with a condition that is not drawn horizontally or a transition action that is not drawn vertically.	Modify the model.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for display of nondefault block attributes

Check for display of nondefault block attributes.

Description

Model diagrams should display block parameters that have values other than default values. One way of displaying this information is by using the **Block Annotation** tab in the Block Properties dialog box.

This guideline facilitates

- Readability
- Verification and validation

See MAAB guideline db_0140: Display of basic block parameters.

Results and Recommended Actions

Condition	Recommended Action
Block parameters that have values other than default values do not appear in the model display.	Use the Block Annotation tab in the Block Properties dialog to add block parameter annotations.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for proper labeling on signal lines

Check for proper labeling on signal lines.

Description

You should use a label to identify:

- Signals originating from the following blocks (the block icon exception noted below applies to all blocks listed except Inport, Bus Selector, Demux, and Selector):

Bus Selector block (tool forces labeling)

Chart block (Stateflow)

Constant block

Data Store Read block

Demux block

From block

Inport block

Selector block

Subsystem block

Block Icon Exception If a signal label is visible in the display of the icon for the originating block, you do not have to display a label for the connected signal unless the signal label is needed elsewhere due to a rule for signal destinations.

- Signals connected to one of the following destination blocks (directly or indirectly with a basic block that performs an operation that is not transformative):

Bus Creator block

Chart block (Stateflow)

Data Store Write block

Goto block

Mux block

Outport block

Subsystem block

- Any signal of interest.

This guideline facilitates

- Readability
- Workflow
- Verification and validation
- Code generation

See MAAB guideline na_0008: Display of labels on signals.

Results and Recommended Actions

Condition	Recommended Action
Signals coming from Bus Selector, Chart, Constant, Data Store Read, Demux, From, Inport, or Selector blocks are not labeled.	Double-click the line that represents the signal. After the text cursor appears, enter a name and click anywhere outside the label to exit label editing mode.

See Also

- Signal Labels in the Simulink documentation
- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation

Check for propagated labels on signal lines

Check for propagated labels on signal lines.

Description

You should propagate a signal label from its source rather than enter the signal label explicitly (manually) if the signal originates from:

- An Inport block in a nested subsystem. However, if the nested subsystem is a library subsystem, you can explicitly label the signal coming from the Inport block to accommodate reuse of the library block.
- A basic block that performs a nontransformative operation.
- A Subsystem or Stateflow Chart block. However, if the connection originates from the output of an instance of the library block, you can explicitly label the signal to accommodate reuse of the library block.

This guideline facilitates

- Readability
- Workflow
- Verification and validation
- Code generation

See MAAB guideline na_0009: Entry versus propagation of signal labels.

Results and Recommended Actions

Condition	Recommended Action
The model includes signal labels that were entered explicitly, but should be propagated.	Use the open angle bracket (<) character to mark signal labels that should be propagated and remove the labels that were entered explicitly.

See Also

- Signal Labels in the Simulink documentation
- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation

Check default transition placement in Stateflow charts

Check default transition placement in Stateflow charts.

Description

In a Stateflow chart, you should connect the default transition at the top of the state and place the destination state of the default transition above other states in the hierarchy.

Properly position the default transition and its destination state for:

- Readability

See MAAB guideline jc_0531: Placement of the default transition.

Results and Recommended Actions

Condition	Recommended Action
The default transition for a Stateflow chart is not connected at the top of the state.	Move the default transition to the top of the state chart.
The destination state of a Stateflow chart's default transition is lower than other states in the same hierarchy.	Adjust the position of the default transition's destination state such that the state is above other states in the same hierarchy.

See Also

- “Defining Transitions Between States” in the Stateflow documentation
- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation

Check setting Stateflow graphical function return value

Check setting Stateflow graphic function return value.

Description

The return value from a Stateflow graphical function must be set in only one place.

This guideline facilitates

- Workflow
- Code generation

See MAAB guideline jc_0511: Setting the return value from a graphical function.

Results and Recommended Actions

Condition	Recommended Action
The return value from a Stateflow graphical function is set in multiple places.	Modify the function such that its return value is set in one place.

See Also

- “Graphical Functions” in the Stateflow documentation
- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation

Check for blocks not using one-based indexing

Check for blocks that do not use one-based indexing.

Description

One-based indexing ([1, 2, 3,...]) is used for the following:

Product	Items
MATLAB	<ul style="list-style-type: none"> • Workspace variables and structures • Local variables of MATLAB functions • Global variables
Simulink	<ul style="list-style-type: none"> • Signal vectors and matrices • Parameter vectors and matrices • S-function input and output signal vectors and matrices in M-code • S-function parameter vectors and matrices in M-code • S-function local variables in M-code
Stateflow	<ul style="list-style-type: none"> • Input and output signal vectors and matrices • Parameter vectors and matrices • Local variables

Zero-based indexing ([0, 1, 2, ...]) is used for the following:

Product	Items
Simulink	<ul style="list-style-type: none"> • Signal vectors and matrices • S-function input and output signal vectors and matrices in C code • S-function input parameters in C code • S-function parameter vectors and matrices in C code • S-function local variables in C code
Stateflow	<ul style="list-style-type: none"> • Variables and structures in custom C code
C code	<ul style="list-style-type: none"> • Local variables and structures • Global variables

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline db_0112: Indexing.

Results and Recommended Actions

Condition	Recommended Action
Blocks in your model are not configured for one-based indexing.	Using block parameters, configure all blocks for one-based indexing.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for invalid file names

Check for files residing in the same folder as the model that have illegal file names.

Description

This guideline facilitates

- Readability
- Workflow

See MAAB guideline ar_0001: Filenames.

Results and Recommended Actions

Condition	Recommended Action
The file name contains illegal characters.	Rename the file. Allowed characters are a–z, A–Z, 0–9, and underscore (_).
The file name starts with a number.	Rename the file.
The file name starts with an underscore ("_").	Rename the file.
The file name ends with an underscore ("_").	Rename the file.
The file extension contains one (or more) underscores.	Change the file extension.
The file name has consecutive underscores.	Rename the file.
The file name contains more than one dot (".").	Rename the file.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for invalid model directory names

Checks model directory and subdirectory names for invalid characters.

Description

This guideline facilitates

- Readability
- Workflow

See MAAB guideline ar_0002: Directory names.

Results and Recommended Actions

Condition	Recommended Action
The directory name contains illegal characters.	Rename the directory. Allowed characters are a–z, A–Z, 0–9, and underscore (_).
The directory name starts with a number.	Rename the directory.
The directory name starts with an underscore ("_").	Rename the directory.
The directory name ends with an underscore ("_").	Rename the directory.
The directory name has consecutive underscores.	Rename the directory.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for blocks that are not discrete

Check for blocks that are not discrete.

Description

You cannot include continuous blocks in controller models.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jm_0001: Prohibited Simulink standard blocks inside controllers.

Results and Recommended Actions

Condition	Recommended Action
Continuous blocks — Derivative, Integrator, State-Space, Transfer Fcn, Transfer Delay, Variable Time Delay, Variable Transport Delay, and Zero-Pole — are not permitted in models representing discrete controllers.	Replace continuous blocks with the equivalent blocks discretized in the s-domain by using the Discretizing library, as explain in “How to Discretize Blocks from the Simulink Model” in the Simulink documentation.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for prohibited sink blocks

Check for prohibited Simulink sink blocks.

Description

You must design controller models from discrete blocks. Sink blocks, such as the Scope block, are not allowed.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline hd_0001: Prohibited Simulink sinks.

Results and Recommended Actions

Condition	Recommended Action
Sink blocks are not permitted in discrete controllers.	Remove sink blocks from the model.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for invalid port positioning and configuration

Check whether the model contains ports with invalid position and configuration.

Description

In models, ports must comply with the following rules:

- Place Inport blocks on the left side of the diagram. Move the Inport block right only to prevent signal crossings.
- Place Outport blocks on the right side of the diagram. Move the Outport block left only to prevent signal crossings.
- Avoid using duplicate Inport blocks at the subsystem level if possible.
- Do not use duplicate Inport blocks at the root level.

This guideline facilitates

- Readability

See MAAB guideline db_0042: Port block in Simulink models.

Results and Recommended Actions

Condition	Recommended Action
Inport blocks are too far to the right and result in left-flowing signals.	Move the specified Inport blocks to the left.
Outport blocks are too far to the left and result in right-flowing signals.	Move the specified Output blocks to the right.

Condition	Recommended Action
Ports do not have the default orientation.	Modify the model diagram such that signal lines for output ports enter the side of the block and signal lines for input ports exit the right side of the block.
Ports are duplicate Inport blocks.	<ul style="list-style-type: none"><li data-bbox="872 475 1317 569">• If the duplicate Inport blocks are in a subsystem, remove them where possible.<li data-bbox="872 586 1317 647">• If the duplicate Inport blocks are at the root level, remove them.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for mismatches between names of ports and corresponding signals

Check for mismatches between names of ports and corresponding signals.

Description

Use matching names for ports and their corresponding signals.

This guideline facilitates

- Readability
- Workflow
- Simulation

See MAAB guideline jm_0010: Port block names in Simulink models.

Prerequisite

Prerequisite MAAB guidelines for this check are:

- db_0042: Port block in Simulink models
- na_0005: Port block name visibility in Simulink models

Results and Recommended Actions

Condition	Recommended Action
Ports have names that differ from their corresponding signals.	Change the port name or the signal name to match the correct name for the signal.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether block names do not appear below blocks

Check whether block names do not appear below blocks.

Description

If shown, the name of all blocks should appear below the blocks.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline db_0142: Position of block names.

Results and Recommended Actions

Condition	Recommended Action
Blocks have names that do not appear below the blocks.	Set the name of the block to appear below the blocks.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for systems that mix primitive blocks and subsystems

Check for systems that mix primitive blocks and subsystems.

Description

You must design every level of a model with building blocks of the same type, for example, only subsystems or only primitive (basic) blocks.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0143: Similar block types on the model levels.

Results and Recommended Actions

Condition	Recommended Action
A level in the model includes both subsystem blocks and primitive blocks.	<ul style="list-style-type: none"> • Move nonvirtual blocks into the subsystem. • If possible, replace blocks at the identified level of the model hierarchy with blocks that you can place at any module level. Such blocks include Inport, Outport, Enable (not at highest model level), Trigger (not at highest model level), Mux, Demux, Bus Selector, Bus Creator, Selector, Ground, Terminator, From, Goto, Switch, Multiport Switch, Merge, Unit Delay, Rate Transition, Type Conversion, Data Store Memory, If, and Switch Case.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether model has unconnected block input ports, output ports, or signal lines

Check whether model has unconnected input ports, output ports, or signal lines.

Description

All unconnected inputs should be connected to ground blocks. All unconnected outputs should be connected to terminator blocks. Respecting the guideline eliminates error messages.

See MAAB guideline db_0081: Unconnected signals, block inputs and block outputs.

Results and Recommended Actions

Condition	Recommended Action
Blocks have unconnected inputs or outputs.	Connect unconnected lines to blocks specified by the design or to Ground or Terminator blocks.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for improperly positioned Trigger and Enable blocks

Check for improperly positioned Trigger and Enable blocks.

Description

Locate blocks that define subsystems as conditional or iterative at the top of the subsystem diagram.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0146: Triggered, enabled, conditional Subsystems.

Results and Recommended Actions

Condition	Recommended Action
Trigger , Enable, and Action Port blocks are not centered in the upper third of the model diagram.	Move the Trigger, Enable, and Action Port blocks to the correct area of the model diagram.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether annotations have drop shadows

Check whether annotations have drop shadows.

Description

Annotations should not have a drop shadow for readability.

This guideline facilitates

- Readability

See MAAB guideline jm_0013: Annotations.

Results and Recommended Actions

Condition	Recommended Action
Annotations display drop shadows.	Clear the Format > Show Drop Shadow menu option.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether tunable parameters specify expressions, data type conversions, or indexing operations

Check whether tunable parameters specify expressions, data type conversions, or indexing operations.

Description

To ensure that a parameter is tunable, you must enter the basic block without the use of MATLAB calculations or scripting. For example, omit

- Expressions
- Data type conversions
- Selections of rows or columns

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline db_0110: Tunable parameters in basic blocks.

Results and Recommended Actions

Condition	Recommended Action
Blocks have a tunable parameter that specifies an expression, data type conversion, or indexing operation.	In each case, move the calculation outside of the block, for example, by performing the calculation with a series of Simulink blocks, or precompute the value in the base workspace as a new variable.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether Stateflow events are defined at the chart level or below

Check whether Stateflow events are defined at the chart level or below.

Description

All events of a Stateflow chart must be defined at the chart level or lower. Events cannot be at the machine level; that is, charts cannot interact with local events.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0126: Scope of events.

Results and Recommended Actions

Condition	Recommended Action
An event in a chart is not defined at the chart level or below.	Define the event at the chart level or below.

See Also

- “Defining Events” in the Stateflow documentation.
- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation

Check whether Stateflow data objects with local scope are defined at the chart level or below

Check whether Stateflow data objects with local scope are defined at the chart level or below.

Description

You must define all local data of a Stateflow block on the chart level or below in the object hierarchy. You cannot define local variables on the machine level; however, parameters and constants are allowed at the machine level.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0125: Scope of internal signals and local auxiliary variables.

Results and Recommended Actions

Condition	Recommended Action
Local data is not defined in the Stateflow hierarchy at the chart level or below.	Define local data at the chart level or below.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check interface signals and parameters

Check whether labeled Stateflow and Simulink input and output signals are strongly typed.

Description

Strong data typing between Stateflow and Simulink input and output signals is required.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0122: Stateflow and Simulink interface signals and parameters.

Results and Recommended Actions

Condition	Recommended Action
A Stateflow chart does not use strong data typing with Simulink.	Select the Use Strong Data Typing with Simulink I/O check box for the specified block.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for exclusive states, default states, and substate validity

Check states in state machines.

Description

In state machines:

- There must be at least two exclusive states.
- A state cannot have only one substate.
- The initial state of a hierarchical level with exclusive states is clearly defined by a default transition.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db_0137: States in state machines.

Prerequisite

A prerequisite MAAB guideline for this check is db_0149: Flowchart patterns for condition actions.

Results and Recommended Actions

Condition	Recommended Action
A system is underspecified.	Validate that the intended design is properly represented in the Stateflow diagram.
Chart has only one exclusive (OR) state.	Make the state a parallel state, or add another exclusive (OR) state.

Condition	Recommended Action
Chart does not have a default state defined.	Define a default state.
Chart has multiple default states defined.	Define only one default state. Make the others nondefault.
State has only one exclusive (OR) substate.	Make the state a parallel state, or add another exclusive (OR) state.
State does not have a default substate defined.	Define a default substate.
State has multiple default substates defined.	Define only one default substate, make the others nondefault.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check optimization parameters for Boolean data types

Check the optimization parameter for Boolean data types.

Description

Optimization for Boolean data types is required

This guideline facilitates

- Workflow
- Code generation

See MAAB guideline jc_0011: Optimization parameters for Boolean data types.

Prerequisite

A prerequisite MAAB guideline for this check is na_0002: Appropriate implementation of fundamental logical and numerical operations.

Results and Recommended Actions

Condition	Recommended Action
Configuration setting for Implement logic signals as boolean data (vs. double) is incorrect.	Select the Implement logic signals as boolean data (vs. double) check box in the Configuration Parameters dialog box Optimization pane.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check model diagnostic settings

Check the model diagnostics configuration parameter settings.

Description

You should enable the following diagnostics:

Algebraic loop

Minimize algebraic loop

Inf or NaN block output

Duplicate data store names

Unconnected block input ports

Unconnected block output ports

Unconnected line

Unspecified bus object at root Outport block

Mux blocks used to create bus signals

Element name mismatch

Invalid function-call connection

This guideline facilitates

- Workflow
- Code generation

Diagnostics not listed in the Results and Recommended Actions section below can be set to any value.

See MAAB guideline jc_0021: Model diagnostic settings.

Results and Recommended Actions

Condition	Recommended Action
Algebraic loop is set to none.	Set Algebraic loop on the Diagnostics > Solver pane of the Configuration Parameters dialog box to error or warning. Otherwise, Simulink might attempt to automatically break the algebraic loops, which can affect execution order of the blocks.
Minimize algebraic loop is set to none.	Set Minimize algebraic loop on the Diagnostics > Solver pane of the Configuration Parameters dialog box to error or warning. Otherwise, Simulink might attempt to automatically break the algebraic loops for reference models and atomic subsystems, which can affect the execution order for those models or subsystems.
Inf or NaN block output is set to none, which can result in numerical exceptions in the generated code.	Set Inf or NaN block output on the Diagnostics > Data Validity > Signals pane of the Configuration Parameters dialog box to error or warning.
Duplicate data store names is set to none, which can result in nonunique variable naming in the generated code.	Set Duplicate data store names on the Diagnostics > Data Validity > Signals pane of the Configuration Parameters dialog box to error or warning.
Unconnected block input ports is set to none, which prevents code generation.	Set Unconnected block input ports on the Diagnostics > Data Validity > Signals pane of the Configuration Parameters dialog box to error or warning.
Unconnected block output ports is set to none, which can lead to dead code.	Set Unconnected block output ports on the Diagnostics > Data Validity > Signals pane of the Configuration Parameters dialog box to error or warning.

Condition	Recommended Action
<p>Unconnected line is set to none, which prevents code generation.</p>	<p>Set Unconnected line on the Diagnostics > Connectivity > Signals pane of the Configuration Parameters dialog box to error or warning.</p>
<p>Unspecified bus object at root Output block is set to none, which can lead to an unspecified interface if the model is referenced from another model.</p>	<p>Set Unspecified bus object at root Output block on the Diagnostics > Connectivity > Buses pane of the Configuration Parameters dialog box to error or warning.</p>
<p>Mux blocks used to create bus signals is set to none, which can lead to an unintended bus being created in the model.</p>	<p>Set Mux blocks used to create bus signals on the Diagnostics > Connectivity > Buses pane of the Configuration Parameters dialog box to error or warning.</p>
<p>Element name mismatch is set to none, which can lead to an incorrect interface in the generated code.</p>	<p>Set Element name mismatch on the Diagnostics > Connectivity > Buses pane of the Configuration Parameters dialog box to error or warning.</p>
<p>Invalid function-call connection is set to none, which can lead to an error in the operation of the generated code.</p>	<p>Set Invalid function-call connection on the Diagnostics > Connectivity > Function Calls pane of the Configuration Parameters dialog box to error or warning, since this condition can lead to an error in the operation of the generated code.</p>

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check the display attributes of block names

Check the display attributes of block names.

Description

Block names should be displayed when providing descriptive information. Block names should not be displayed if the block function is known from its appearance.

This guideline facilitates

- Readability

See MAAB guideline jc_0061: Display of block names.

Results and Recommended Actions

Condition	Recommended Action
Block name is not descriptive.	These block names should be modified to be more descriptive or not be shown.
Block name is not displayed.	These block names should be shown since they appear to have a descriptive name.
Block name is obvious.	These block names should not be displayed.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check icon display attributes for port blocks

Check the **Icon display** setting for Inport and Outport blocks.

Description

The **Icon display** setting is required.

This guideline facilitates

- Readability

See MAAB guideline jc_0081: Icon display for Port block.

Results and Recommended Actions

Condition	Recommended Action
The Icon display setting is incorrect.	Set the Icon display to Port number for the specified Inport and Outport blocks.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether subsystem block names include invalid characters

Check whether subsystem block names include invalid characters.

Description

The names of all subsystem blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0201: Usable characters for Subsystem names.

Results and Recommended Actions

Condition	Recommended Action
The subsystem name contains illegal characters.	Rename the subsystem. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The subsystem name starts with a number.	Rename the subsystem.
The subsystem name starts with an underscore ("_").	Rename the subsystem.
The subsystem name ends with an underscore ("_").	Rename the subsystem.
The subsystem name contains consecutive underscores.	Rename the subsystem.
The subsystem name has consecutive underscores.	Rename the subsystem.
The subsystem name has blank spaces.	Rename the subsystem.

Tips

Use underscores to separate parts of a subsystem name instead of spaces.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether Inport and Outport block names include invalid characters

Check whether Inport and Outport block names include invalid characters.

Description

The names of all Inport and Outport blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0211: Usable characters for Inport blocks and Outport blocks.

Results and Recommended Actions

Condition	Recommended Action
The block name contains illegal characters.	Rename the block. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The block name starts with a number.	Rename the block.
The block name starts with an underscore ("_").	Rename the block.
The block name ends with an underscore ("_").	Rename the block.
The block name contains consecutive underscores.	Rename the block.
The block name has consecutive underscores.	Rename the block.
The block name has blank spaces.	Rename the block.

Tips

Use underscores to separate parts of a block name instead of spaces.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether signal line names include invalid characters

Check whether signal line names include invalid characters.

Description

The names of all signal lines are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0221: Usable characters for signal line names.

Results and Recommended Actions

Condition	Recommended Action
The signal line name contains illegal characters.	Rename the signal line. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The signal line name starts with a number.	Rename the signal line.
The signal line name starts with an underscore ("_").	Rename the signal line.
The signal line name ends with an underscore ("_").	Rename the signal line.
The signal line name contains consecutive underscores.	Rename the signal line.
The signal line name has consecutive underscores.	Rename the signal line.

Condition	Recommended Action
The signal line name has blank spaces.	Rename the signal line.
The signal line name has control characters.	Rename the signal line.

Tips

Use underscores to separate parts of a signal line name instead of spaces.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check whether block names include invalid characters

Check whether block names include invalid characters.

Description

The names of all blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

This guideline does not apply to subsystem blocks.

See MAAB guideline jc_0231: Usable characters for block names.

Prerequisite

A prerequisite MAAB guideline for this check is jc_0201: Usable characters for Subsystem names.

Results and Recommended Actions

Condition	Recommended Action
The block name contains illegal characters.	Rename the block. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The block name starts with a number.	Rename the block.
The block name has blank spaces.	Rename the block.
The block name has double byte characters.	Rename the block.

Tips

Carriage returns are allowed in block names.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check Trigger and Enable block port names

Check Trigger and Enable block port names.

Description

Block port names should match the name of the signal triggering the subsystem.

This guideline facilitates

- Readability

See MAAB guideline jc_0281: Naming of Trigger Port block and Enable Port block.

Results and Recommended Actions

Condition	Recommended Action
Trigger block does not match the name of the signal to which it is connected.	Match Trigger block names to the connecting signal.
Enable block does not match the name of the signal to which it is connected.	Match Enable block names to the connecting signal.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for Simulink diagrams that have nonstandard appearance attributes

Check model appearance setting attributes.

Description

Model appearance settings are required to conform to the guidelines when the model is released.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline na_0004: Simulink model appearance.

Results and Recommended Actions

Condition	Recommended Action
Diagrams do not have white backgrounds.	Select Format > Screen Color > Automatic .
Diagrams do not have zoom factor set to 100%.	Select View > Normal (100%) .
The toolbar is not visible.	Select View > Toolbar .
The status bar is not visible.	Select View > Status Bar .
Block backgrounds are not white.	Blocks should have black foregrounds with white backgrounds. Click the specified block and select Format > Foreground Color > Black and Format > Background Color > White .
Wide Nonscalar Lines is cleared.	Select Format > Port/Signal Displays > Wide Nonscalar Lines .

Condition	Recommended Action
Viewer Indicators is cleared.	Select Format > Port/Signal Displays > Viewer Indicators .
Testpoint Indicators is cleared.	Select Format > Port/Signal Displays > Testpoint Indicators .
Port Data Types is selected.	Clear Format > Port/Signal Displays > Port Data Types .
Storage Class is selected.	Clear Format > Port/Signal Displays > Storage Class .
Signal Dimensions is selected.	Clear Format > Port/Signal Displays > Signal Dimensions .
Model Browser is selected.	Clear View > Model Browser Options > Model Browser .
Sorted Order is selected.	Clear Format > Block Displays > Sorted Order .
Model Block Version is selected.	Clear Format > Block Displays > Model Block Version .
Model Block I/O Mismatch is selected.	Clear Format > Block Displays > Model Block I/O Mismatch .
Execution Context Indicator is selected.	Clear Format > Block Displays > Execution Context Indicator .
Sample Time Colors is selected.	Clear Format > Port/Signal Displays > Sample Time Colors .
Library Link Display is set to User or All .	Select Format > Library Link Display > None .
Linearization Indicators is cleared.	Select Format > Port/Signal Displays > Linearization Indicators .

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check visibility of port block names

Check the visibility of port block names.

Description

An organization applying the MAAB guidelines must select one of the following alternatives to enforce:

- The name of port blocks are not hidden.
- The name of port blocks must be hidden.

This guideline facilitates

- Readability

Note This check does not look in masked subsystems.

See MAAB guideline na_0005: Port block name visibility in Simulink models.

Input Parameters

All Port names should be shown (Format/Show Name)

Select this check box if all ports should show the name, including subsystems.

Results and Recommended Actions

Condition	Recommended Action
Blocks do not show their name and the All Port names should be shown (Format/Show Name) check box is selected.	Change the format of the specified blocks to show names according to the input requirement.

Condition	Recommended Action
Blocks show their name and the All Port names should be shown (Format/Show Name) check box is cleared.	Change the format of the specified blocks to hide names according to the input requirement.
Subsystem blocks do not show their port names.	Set the subsystem parameter Show port labels to a value other than none.
Subsystem blocks show their port names.	Set the subsystem parameter Show port labels to none.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for direction of subsystem blocks

Check the orientation of subsystem blocks.

Description

Subsystem inputs must be located on the left side of the block, and outputs must be located on the right side of the block.

This guideline facilitates

- Readability

See MAAB guideline jc_0111: Direction of Subsystem.

Results and Recommended Actions

Condition	Recommended Action
Subsystem blocks are not in the correct orientation.	Change the subsystem blocks to have the correct orientation, with inports on the left and outports on the right.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for proper position of constants used in Relational Operator blocks

Check the position of Constant blocks used in Relational Operator blocks.

Description

When the relational operator is used to compare a signal to a constant value, the constant input should be the second, lower input.

This guideline facilitates

- Readability
- Code generation

See MAAB guideline jc_0131: Use of Relational Operator block.

Results and Recommended Actions

Condition	Recommended Action
Relational Operator blocks have a Constant block on the first, upper input.	Move the Constant block to the second, lower input.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for entry format in state blocks

Check the format of entries in state blocks.

Description

A new line should be started after the entry, during, and exit action statements and after the completion of an assignment statement “;”.

This guideline facilitates

- Readability

See MAAB guideline jc_0501: Format of entries in a State block.

Results and Recommended Actions

Condition	Recommended Action
An entry in a state block is not formatted correctly.	Validate that the intended design is properly represented in the Stateflow diagram.
An entry action statement is not by itself.	Add a new line.
Multiple entry action statements found on one line.	Add a new line between entry action statements.
An during action statement is not by itself.	Add a new line.
Multiple during action statements found on one line.	Add a new line between during action statements.
An exit action statement is not by itself.	Add a new line.
Multiple exit action statements found on one line.	Add a new line between exit action statements.

Condition	Recommended Action
Multiple action statements found on one line.	Add a new line between action statements.
Potential misuse of semicolon (;) on a line.	Correct the use of the semicolon where specified.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for use of tunable parameters in Stateflow

Check for use of tunable parameters in Stateflow charts.

Description

Include tunable parameters in a Stateflow chart as inputs from the Simulink model.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc_0541: Use of tunable parameters in Stateflow.

Results and Recommended Actions

Condition	Recommended Action
Stateflow charts reference Simulink data objects, which should be used as inputs from the Simulink model.	Make the Simulink data objects inputs from the Simulink model to the specified Stateflow chart.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for proper use of Switch blocks

Check for proper use of Switch blocks.

Description

This check verifies that the Switch block's control input (the second input) is a Boolean value and that the block is configured to pass the first input when the control input is nonzero.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline jc_0141: Use of the Switch block.

Results and Recommended Actions

Condition	Recommended Action
The Switch block's control input (second input) is not a Boolean value.	Change the data type of the control input to Boolean.
The Switch block is not configured to pass the first input when the control input is nonzero.	Set the block parameter Criteria for passing first input to <code>u2 ~=0</code> .

See Also

- See the description of the Switch block in the Simulink documentation.
- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation

Check for proper use of signal buses and Mux block usage

Check for proper use of signal busses and Mux block usage.

Description

This check verifies whether a model is using signal buses and Mux blocks properly.

This guideline facilitates

- Readability
- Workflow

See MAAB guidelinenena_0010: Grouping data flows into signals.

Results and Recommended Actions

Condition	Recommended Action
The individual scalar input signals for a Mux block do not have common functionality, data types, dimensions, and units.	Modify the scalar input signals such that the specifications match.
The output of a Mux block is not a vector.	Change the output of the Mux block to a vector.
All inputs to a Mux block are not scalars.	Make sure that all input signals to Mux blocks are scalars.
The input for a Bus Selector block is not a bus signal.	Make sure that the input for all Bus Selector blocks is a bus signal.

See Also

- Using Composite Signals in the Simulink documentation.

- *MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0)* in the Simulink Verification and Validation reference documentation

Check for mismatches between Stateflow ports and associated signal names

Check for mismatches between Stateflow ports and associated signal names.

Description

The name of Stateflow input and output should be the same as the corresponding signal. This guideline is required for:

- Readability
- Workflow

See MAAB guideline db_0123: Stateflow port names.

Results and Recommended Actions

Condition	Recommended Action
Signals have names that differ from those of their corresponding Stateflow ports.	Change the names of either the signals or the Stateflow ports.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Check for proper scope of From and Goto blocks

Check the scope of From and Goto blocks.

Description

You can use global scope for controlling flow. However, From and Goto blocks must use local scope for signal flows.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline na_0011: Scope of Goto and From blocks.

Results and Recommended Actions

Condition	Recommended Action
From and Goto blocks are not configured with local scope.	<ul style="list-style-type: none">• Make sure the ports are connected correctly.• Change the scope of the specified blocks to local.

See Also

MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow (Version 2.0) in the Simulink Verification and Validation reference documentation

Requirements Consistency Checks

In this section...

“Identify requirement links with missing documents” on page 14-146

“Identify requirement links that specify invalid locations within documents” on page 14-147

“Identify selection-based links having descriptions that do not match their requirements document text” on page 14-148

“Identify requirement links with inconsistent path types and preferences” on page 14-150

Identify requirement links with missing documents

Ensure that requirements link to existing documents.

Description

You used the Requirements Management Interface (RMI) to associate a design requirements document with a part of your model design and the interface cannot find the specified document.

Results and Recommended Actions

Condition	Recommended Action
The requirements document associated with a part of your model design is not accessible at the specified location.	Open the Requirements dialog box and correct the path name of the requirements document or move the document to the specified location.

Tips

If your model has links to a DOORS requirements document, to run this check, the DOORS software must be open and you must be logged in.

See Also

Identify requirement links that specify invalid locations within documents

Ensure that requirements link to valid locations (e.g., bookmarks, line numbers, anchors) within documents.

Description

You used the Requirements Management Interface (RMI) to associate a location in a design requirements document (a bookmark, line number, or anchor) with a part of your model design and the interface cannot find the specified location in the specified document.

Results and Recommended Actions

Condition	Recommended Action
The location in the requirements document associated with a part of your model design is not accessible.	Open the Requirements dialog box and correct the location reference within the requirements document.

Tips

If your model has links to a DOORS requirements document, to run this check, the DOORS software must be open and you must be logged in.

If your model has links to a Microsoft Word or Microsoft Excel document, to run this check, those applications must be closed on your computer.

See Also

Identify selection-based links having descriptions that do not match their requirements document text

Ensure that descriptions of selection-based links use the same text found in their requirements documents.

Description

You used selection-based linking of the Requirements Management Interface (RMI) to label requirements in the model's **Requirements** menu with text that appears in the corresponding requirements document. This check helps you manage traceability by identifying requirement descriptions in the menu that are not synchronized with text in the documents.

Results and Recommended Actions

Condition	Recommended Action
Selection-based links have descriptions that differ from their corresponding selections in the requirements documents.	If the difference reflects a change in the requirements document, click the link in the Model Advisor results to replace the current description in the selection-based link with the text from the requirements document (the external description). Alternatively, you can right click the object in the model window, select Edit/Add Links from the Requirements menu, and use the Requirements dialog box that appears to synchronize the text.

Tips

If your model has links to a DOORS requirements document, to run this check, the DOORS software must be open and you must be logged in.

If your model has links to a Microsoft Word or Microsoft Excel document, to run this check, those applications must be closed on your computer.

See Also

Identify requirement links with inconsistent path types and preferences

Check that requirement paths are of the type selected in the preferences.

Description

You are using the Requirements Management Interface (RMI) and the paths specifying the location of your requirements documents differ from the file reference type set as your preference.

Results and Recommended Actions

Condition	Recommended Action
<p>The paths indicating the location of requirements documents use a file reference type that differs from the preferences specified in the Selection-based linking dialog box.</p>	<p>Change the preferred document file reference type or the specified paths by doing one of the following:</p> <ul style="list-style-type: none"> • Click Fix to change the current path to the valid path. • Update the preference in the Selection-based linking dialog box. In the model window, select Tools > Requirements > Link settings and change the value for the Document file reference option.

See Also

Examples

Use this list to find examples in the documentation.

Requirements Management Interface

- “Linking from a Simulink Object to a Selected Item in a Requirements Document” on page 2-7
- “Linking from a Simulink Object to a Specified Location in a Requirements Document” on page 2-9
- “Adding Requirement Links to Multiple Objects Simultaneously” on page 2-12
- “Linking a Signal Builder Block to a Requirement” on page 2-13
- “Viewing Simulink Objects That Have Requirements Links” on page 2-17
- “Highlighting Objects with Requirements in the Model Editor” on page 2-17
- “Highlighting Objects with Linked Requirements from Model Explorer” on page 2-18
- “Deleting a Single Link from a Simulink Object” on page 2-19
- “Deleting All Links from a Simulink Object” on page 2-19
- “Deleting Links from Multiple Simulink Objects” on page 2-19
- “Creating Requirements in Linked Libraries” on page 2-21
- “Creating the Default Requirements Report” on page 2-22
- “Customizing a Requirements Report” on page 2-26
- “Configuring the RMI to Insert Navigation Controls” on page 2-30
- “Enabling ActiveX Controls” on page 2-30
- “Creating Navigation Controls in Requirements Documents” on page 2-31
- “Troubleshooting Simulink Navigation Controls in Microsoft Office 2007” on page 2-32
- “Creating a Custom Link Requirement Type” on page 2-44
- “Using the System Requirements Block in a Model” on page 2-56
- “Adding the System Requirements Block” on page 2-56
- “Renaming the System Requirements Block” on page 2-57
- “Including Requirements Information with Generated Code” on page 2-59

Requirements Management Interface (DOORS Version)

- “Creating DOORS Requirements” on page 3-7
- “Creating One-Way Links from Simulink Objects to DOORS Requirements” on page 3-8

- “Navigating from a Simulink Object to a DOORS Requirement” on page 3-10
- “Synchronizing a Simulink Model to Create a Surrogate Module” on page 3-14
- “Updating the Surrogate Module to Reflect Model Changes” on page 3-22
- “Navigating Using the Surrogate Module” on page 3-25
- “Navigating from a Simulink Object to a Requirement” on page 3-26
- “Navigating from a Requirement to the Model” on page 3-26
- “Viewing Objects with Requirements in the Model Editor” on page 3-28
- “Viewing Objects with Requirements in the Model Explorer” on page 3-28
- “Creating a Default Requirements Report for a Model” on page 3-30
- “Customizing a Requirements Report with Links to DOORS Requirements” on page 3-31
- “Creating Two-Way Links” on page 3-35
- “Navigating Two-Way Links” on page 3-36

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” on page 5-33
- “Decisions Analyzed” on page 5-39
- “Conditions Analyzed” on page 5-41
- “MCDC Analysis” on page 5-41
- “N-Dimensional Lookup Table” on page 5-45
- “Signal Range Analysis” on page 5-54
- “Displaying Model Coverage with Model Coloring” on page 5-70

“Creating a Model with Embedded MATLAB Function Block Decisions”
on page 5-88

“Understanding Embedded MATLAB Function Block Model Coverage”
on page 5-92

Model Advisor Check

“Model Advisor Code Example: Registering Custom Checks and Process
Callbacks” on page 7-7

“Model Advisor Code Example: Check Definition Function” on page 7-15

“Model Advisor Code Example: Input Parameter Definition” on page 7-17

“Model Advisor Code Example: List View Definition” on page 7-19

“Model Advisor Code Example: Action Definition” on page 7-20

“Model Advisor Code Example: Informational Check Callback Function”
on page 7-24

“Model Advisor Code Example: Basic Check with Pass/Fail Status” on
page 7-26

“Model Advisor Code Example: Check With Subchecks and Actions” on
page 7-29

“Model Advisor Code Example: Action Callback Function” on page 7-37

“Model Advisor Code Example: Formatted Output” on page 7-41

Model Advisor Organization

“How To Organize Checks and Folders Using the Model Advisor
Configuration Editor” on page 8-10

“Model Advisor Code Example: Registering Custom Tasks and Folders”
on page 8-15

“Model Advisor Code Example: Task Definition Function” on page 8-17

“Model Advisor Code Example: Group Definition” on page 8-19

A

Assertion block appearance 4-19

C

categorical lists of functions 10-1 11-1

classes

- cv.cvdatagroup 12-15
- cv.cvtestgroup 12-17
- ModelAdvisor.Action 12-54
- ModelAdvisor.Check 12-56
- ModelAdvisor.FactoryGroup 12-60
- ModelAdvisor.FormatTemplate 12-62
- ModelAdvisor.Group 12-70
- ModelAdvisor.Image 12-72
- ModelAdvisor.InputParameter 12-74
- ModelAdvisor.LineBreak 12-77
- ModelAdvisor.List 12-79
- ModelAdvisor.ListViewParameter 12-81
- ModelAdvisor.Paragraph 12-84
- ModelAdvisor.Root 12-86
- ModelAdvisor.Table 12-88
- ModelAdvisor.Task 12-90
- ModelAdvisor.Text 12-93

closing Signal Builder Requirements pane 4-13

colored diagram model coverage display 5-69

- enabling 5-69

condition coverage

- Embedded MATLAB Function blocks 5-102
- statements in Embedded MATLAB Function block 5-88

conditioninfo function 12-11

constructors

- cv.cvdatagroup 12-16
- cv.cvtestgroup 12-18
- ModelAdvisor.Action 12-55
- ModelAdvisor.Check 12-59
- ModelAdvisor.FactoryGroup 12-61
- ModelAdvisor.FormatTemplate 12-69
- ModelAdvisor.Group 12-71

- ModelAdvisor.Image 12-73
- ModelAdvisor.InputParameter 12-75
- ModelAdvisor.LineBreak 12-78
- ModelAdvisor.List 12-80
- ModelAdvisor.ListViewParameter 12-83
- ModelAdvisor.Paragraph 12-85
- ModelAdvisor.Root 12-87
- ModelAdvisor.Table 12-89
- ModelAdvisor.Task 12-92
- ModelAdvisor.Text 12-94

cv.cvdatagroup class 12-15

cv.cvdatagroup constructor 12-16

cv.cvdatagroup.allNames method 12-9

cv.cvdatagroup.get method 12-41

cv.cvdatagroup.getAll method 12-43

cv.cvdatagroup.name property 12-182

cv.cvtestgroup class 12-17

cv.cvtestgroup constructor 12-18

cv.cvtestgroup.add method 12-2

cv.cvtestgroup.allNames method 12-10

cv.cvtestgroup.get method 12-42

cv.cvtestgroup.name property 12-183

cvexit function 12-19

cvhtml function 12-20

- model coverage 5-77

cvload function 12-23

- model coverage 5-79

cvmodelview function 12-24

cvsave function 12-26

- model coverage 5-78

cvsim function 12-28

- model coverage 5-76

cvsimref function 12-31

cvtest function 12-34

- model coverage 5-74

D

decision coverage

- Embedded MATLAB Function blocks 5-101

- statements in Embedded MATLAB Function blocks 5-87
- decisioninfo function 12-37
- defining Model Advisor checks 7-11
- defining Model Advisor folders 8-18
- defining Model Advisor tasks 8-15
- demos
 - Model Advisor customization demo 8-20
 - simcovdemo model coverage demo 5-11
- disabling Model Verification blocks across test groups 4-20
- DO-178B
 - Model Advisor checks 14-5
- DOORS
 - additional installation for 3-4
- DOORS Requirements Management Interface
 - block type descriptions 3-21
 - definition for object 3-14
 - from Simulink to DOORS 3-26
 - hierarchical numbers 3-21
 - object identifiers 3-21
 - opening the object in Simulink, Stateflow, or MATLAB 3-26
 - overview 3-2
 - saving formal modules 3-25
 - synchronizing models with DOORS 3-14
 - synchronizing objects with DOORS formal module 3-14
 - viewing model elements with requirements 3-28
 - viewing requirements 3-25

E

- Embedded MATLAB Function blocks
 - condition coverage 5-102
 - condition coverage statements 5-88
 - decision coverage 5-101
 - decision coverage statements 5-87

- MCDC coverage 5-102
- MCDC coverage statements 5-88
- model coverage 5-87
- model coverage example 5-88
- types of model coverage 5-87
- enabling Model Verification blocks across test groups 4-20

F

- functions
 - categories 10-1 11-1
 - conditioninfo 12-11
 - cvexit 12-19
 - cvhtml 12-20
 - cvload 12-23
 - cvmodelview 12-24
 - cvsave 12-26
 - cvsim 12-28
 - cvsimref 12-31
 - cvtest 12-34
 - decisioninfo 12-37
 - getCoverageInfo 12-44
 - mcdcinfo 12-50
 - Model Advisor customization API 10-5 11-3
 - Model Advisor formatting API 10-8 11-5
 - Model Advisor result template API 10-7 11-4
 - model coverage 10-3 11-2
 - rmi 12-98
 - rmidocrename 12-104
 - rminav 12-106
 - sigrangeinfo 12-151
 - start old Requirements Management Interface 10-2
 - tableinfo 12-153

G

- getCoverageInfo function 12-44

I

icons for Model Verification blocks in Verification Manager 4-16
 IEC 61508
 Model Advisor checks 14-62
 installing DOORS 3-4

L

Lookup Table block in model coverage report 5-45
 Lookup Table model coverage
 n-dimensional 5-52
 three-dimensional example 5-49
 two-dimensional example 5-45

M

MathWorks Automotive Advisory Board
 Model Advisor checks 14-79
 MCDC coverage
 Embedded MATLAB Function blocks 5-102
 statements in Embedded MATLAB Function blocks 5-88
 MCDC table
 condition cases 5-42
 mcdcinfo function 12-50
 methods
 cv.cvdatagroup.allNames 12-9
 cv.cvdatagroup.get 12-41
 cv.cvdatagroup.getAll 12-43
 cv.cvtestgroup.add 12-2
 cv.cvtestgroup.allNames 12-10
 cv.cvtestgroup.get 12-42
 ModelAdvisor.Action.setCallbackFcn 12-110
 ModelAdvisor.Check.getID 12-49
 ModelAdvisor.Check.setAction 12-107
 ModelAdvisor.Check.setCallbackFcn 12-111
 ModelAdvisor.Check.setInputParameters 12-129
 ModelAdvisor.Check.setInputParameters-LayoutGrid 12-130

ModelAdvisor.FactoryGroup.addCheck 12-3
 ModelAdvisor.FormatTemplate.addRow 12-7
 ModelAdvisor.FormatTemplate.-
 setCheckText 12-114
 ModelAdvisor.FormatTemplate.-
 setColTitles 12-119
 ModelAdvisor.FormatTemplate.-
 setInformation 12-128
 ModelAdvisor.FormatTemplate.-
 setListObj 12-132
 ModelAdvisor.FormatTemplate.-
 setRecAction 12-133
 ModelAdvisor.FormatTemplate.-
 setRefLink 12-135
 ModelAdvisor.FormatTemplate.-
 setSubBar 12-141
 ModelAdvisor.FormatTemplate.-
 setSubResultStatus 12-142
 ModelAdvisor.FormatTemplate.-
 setSubResultStatusText 12-143
 ModelAdvisor.FormatTemplate.-
 setSubTitle 12-146
 ModelAdvisor.FormatTemplate.-
 setTableInfo 12-147
 ModelAdvisor.FormatTemplate.-
 setTableTitle 12-148
 ModelAdvisor.Group.AddGroup 12-4
 ModelAdvisor.Group.AddTask 12-8
 ModelAdvisor.Image.setHyperlink 12-125
 ModelAdvisor.Image.setImageSource 12-127
 ModelAdvisor.InputParameter.setColSpan 12-118
 ModelAdvisor.InputParameter.setRowSpan 12-140
 ModelAdvisor.List.addItem 12-5
 ModelAdvisor.List.setType 12-149
 ModelAdvisor.Paragraph.addItem 12-6
 ModelAdvisor.Paragraph.setAlign 12-108
 ModelAdvisor.Root.publish 12-96
 ModelAdvisor.Root.register 12-97
 ModelAdvisor.Table.getEntry 12-48
 ModelAdvisor.Table.setColHeading 12-115

- ModelAdvisor.Table.setColHeadingAlign 12-116
- ModelAdvisor.Table.setColWidth 12-120
- ModelAdvisor.Table.setEntry 12-121
- ModelAdvisor.Table.setEntryAlign 12-122
- ModelAdvisor.Table.setHeading 12-123
- ModelAdvisor.Table.setHeadingAlign 12-124
- ModelAdvisor.Table.setRowHeading 12-138
- ModelAdvisor.Table.setRowHeadingAlign 12-139
- ModelAdvisor.Task.setCheck 12-113
- ModelAdvisor.Text.setBold 12-109
- ModelAdvisor.Text.setColor 12-117
- ModelAdvisor.Text.setHyperlink 12-126
- ModelAdvisor.Text.setItalic 12-131
- ModelAdvisor.Text.setRetainSpace-
Return 12-137
- ModelAdvisor.Text.setSubscript 12-144
- ModelAdvisor.Text.setSuperscript 12-145
- ModelAdvisor.Text.setUnderlined 12-150
- model
 - synchronizing to DOORS surrogate
 - module 3-12
- Model Advisor checks
 - DO-178B 14-5
 - IEC 61508 14-62
 - MathWorks Automotive Advisory
Board 14-79
 - requirements consistency 14-145
- Model Advisor customization API functions 10-5
- Model Advisor customization classes 11-3
- Model Advisor customizations
 - creating check callback functions 7-22
 - defining custom checks 7-11
 - defining custom folders 8-18
 - defining custom tasks 8-15
 - defining process callback functions 7-8
 - formatting Model Advisor results 7-38
 - registering custom checks 7-6
 - registering custom tasks and folders 8-13
 - slvnvdemo_mdladv demo 8-20
 - workflow overview 6-4
- Model Advisor formatting API functions 10-8
- Model Advisor formatting classes 11-5
- Model Advisor result template class 10-7 11-4
- model coverage
 - colored Simulink diagram display 5-69
 - colored Simulink diagram example 5-70
 - commands in MATLAB 5-74
 - Conditions analyzed table 5-41
 - Decisions analyzed table 5-39
 - Embedded MATLAB Function blocks 5-87
 - enabling colored diagram display 5-69
 - enabling colored Simulink diagram
display 5-69
 - HTML settings 5-24
 - introduction 5-2
 - Lookup Table block report 5-45
 - MCDC table 5-42
 - n-dimensional Lookup Table 5-52
 - settings in dialog 5-16
 - signal range analysis report 5-54
 - three-dimensional Lookup Table
example 5-49
 - two-dimensional Lookup Table 5-45
 - understanding report 5-30
 - workflow 5-11
- model coverage demo
 - simcovdemo 5-11
- model coverage functions 10-3 11-2
 - cvhtml 5-77
 - cvload 5-79
 - cvsave 5-78
 - cvsim 5-76
 - cvtest 5-74
- 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

ModelAdvisor.Action class 12-54
ModelAdvisor.Action constructor 12-55
ModelAdvisor.Action.Description
property 12-161
ModelAdvisor.Action.Name property 12-184
ModelAdvisor.Action.setCallbackFcn
method 12-110
ModelAdvisor.Check class 12-56
ModelAdvisor.Check constructor 12-59
ModelAdvisor.Check.CallbackContext
property 12-157
ModelAdvisor.Check.CallbackFunction
property 12-158
ModelAdvisor.Check.CallbackStyle
property 12-159
ModelAdvisor.Check.Enable property 12-169
ModelAdvisor.Check.getID method 12-49
ModelAdvisor.Check.ID property 12-172
ModelAdvisor.Check.LicenseName
property 12-176
ModelAdvisor.Check.ListViewVisible
property 12-178
ModelAdvisor.Check.Result property 12-187
ModelAdvisor.Check.setAction
method 12-107
ModelAdvisor.Check.setCallbackFcn
method 12-111
ModelAdvisor.Check.setInputParameters
method 12-129
ModelAdvisor.Check.setInputParameters-
LayoutGrid method 12-130
ModelAdvisor.Check.Title property 12-188
ModelAdvisor.Check.TitleTips
property 12-189
ModelAdvisor.Check.Value property 12-192
ModelAdvisor.Check.Visible property 12-195
ModelAdvisor.FactoryGroup class 12-60
ModelAdvisor.FactoryGroup constructor 12-61
ModelAdvisor.FactoryGroup.addCheck
method 12-3
ModelAdvisor.FactoryGroup.Description
property 12-162
ModelAdvisor.FactoryGroup.DisplayName
property 12-166
ModelAdvisor.FactoryGroup.ID
property 12-173
ModelAdvisor.FactoryGroup.MAObj
property 12-179
ModelAdvisor.FormatTemplate class 12-62
ModelAdvisor.FormatTemplate
constructor 12-69
ModelAdvisor.FormatTemplate.addRow
method 12-7
ModelAdvisor.FormatTemplate.setCheckText
method 12-114
ModelAdvisor.FormatTemplate.setColTitles
method 12-119
ModelAdvisor.FormatTemplate.setInformation
method 12-128
ModelAdvisor.FormatTemplate.setListObj
method 12-132
ModelAdvisor.FormatTemplate.setRecAction
method 12-133
ModelAdvisor.FormatTemplate.setRefLink
method 12-135
ModelAdvisor.FormatTemplate.setSubBar
method 12-141
ModelAdvisor.FormatTemplate.-
setSubResultStatus method 12-142
ModelAdvisor.FormatTemplate.-
setSubResultStatusText method 12-143
ModelAdvisor.FormatTemplate.setSubTitle
method 12-146
ModelAdvisor.FormatTemplate.setTableInfo
method 12-147
ModelAdvisor.FormatTemplate.setTableTitle
method 12-148
ModelAdvisor.Group class 12-70
ModelAdvisor.Group constructor 12-71
ModelAdvisor.Group.AddGroup method 12-4

- ModelAdvisor.Group.AddTask method 12-8
- ModelAdvisor.Group.Description
 - property 12-163
- ModelAdvisor.Group.DisplayName
 - property 12-167
- ModelAdvisor.Group.ID property 12-174
- ModelAdvisor.Group.MAObj property 12-180
- ModelAdvisor.Image class 12-72
- ModelAdvisor.Image constructor 12-73
- ModelAdvisor.Image.setHyperlink
 - method 12-125
- ModelAdvisor.Image.setImageSource
 - method 12-127
- ModelAdvisor.InputParameter class 12-74
- ModelAdvisor.InputParameter
 - constructor 12-75
- ModelAdvisor.InputParameter.Description
 - property 12-164
- ModelAdvisor.InputParameter.Entries
 - property 12-171
- ModelAdvisor.InputParameter.Name
 - property 12-185
- ModelAdvisor.InputParameter.setColSpan
 - method 12-118
- ModelAdvisor.InputParameter.setRowSpan
 - method 12-140
- ModelAdvisor.InputParameter.Type
 - property 12-190
- ModelAdvisor.InputParameter.Value
 - property 12-193
- ModelAdvisor.LineBreak class 12-77
- ModelAdvisor.LineBreak constructor 12-78
- ModelAdvisor.List class 12-79
- ModelAdvisor.List constructor 12-80
- ModelAdvisor.List.addItem method 12-5
- ModelAdvisor.List.setType method 12-149
- ModelAdvisor.ListViewParameter class 12-81
- ModelAdvisor.ListViewParameter
 - constructor 12-83
- ModelAdvisor.ListViewParameter.Attributes
 - property 12-156
- ModelAdvisor.ListViewParameter.Data
 - property 12-160
- ModelAdvisor.ListViewParameter.Name
 - property 12-186
- ModelAdvisor.Paragraph class 12-84
- ModelAdvisor.Paragraph constructor 12-85
- ModelAdvisor.Paragraph.addItem
 - method 12-6
- ModelAdvisor.Paragraph.setAlign
 - method 12-108
- ModelAdvisor.Root class 12-86
- ModelAdvisor.Root constructor 12-87
- ModelAdvisor.Root.publish method 12-96
- ModelAdvisor.Root.register method 12-97
- ModelAdvisor.Table class 12-88
- ModelAdvisor.Table constructor 12-89
- ModelAdvisor.Table.getEntry method 12-48
- ModelAdvisor.Table.setColHeading
 - method 12-115
- ModelAdvisor.Table.setColHeadingAlign
 - method 12-116
- ModelAdvisor.Table.setColWidth
 - method 12-120
- ModelAdvisor.Table.setEntry method 12-121
- ModelAdvisor.Table.setEntryAlign
 - method 12-122
- ModelAdvisor.Table.setHeading
 - method 12-123
- ModelAdvisor.Table.setHeadingAlign
 - method 12-124
- ModelAdvisor.Table.setRowHeading
 - method 12-138
- ModelAdvisor.Table.setRowHeadingAlign
 - method 12-139
- ModelAdvisor.Task class 12-90
- ModelAdvisor.Task constructor 12-92
- ModelAdvisor.Task.Description
 - property 12-165

- ModelAdvisor.Task.DisplayName
 - property 12-168
 - ModelAdvisor.Task.Enable property 12-170
 - ModelAdvisor.Task.ID property 12-175
 - ModelAdvisor.Task.LicenseName
 - property 12-177
 - ModelAdvisor.Task.MAObj property 12-181
 - ModelAdvisor.Task.setCheck method 12-113
 - ModelAdvisor.Task.Value property 12-194
 - ModelAdvisor.Task.Visible property 12-196
 - ModelAdvisor.Text class 12-93
 - ModelAdvisor.Text constructor 12-94
 - ModelAdvisor.Text.setBold method 12-109
 - ModelAdvisor.Text.setColor method 12-117
 - ModelAdvisor.Text.setHyperlink
 - method 12-126
 - ModelAdvisor.Text.setItalic method 12-131
 - ModelAdvisor.Text.setRetainSpaceReturn
 - method 12-137
 - ModelAdvisor.Text.setSubscript
 - method 12-144
 - ModelAdvisor.Text.setSuperscript
 - method 12-145
 - ModelAdvisor.Text.setUnderlined
 - method 12-150
 - models
 - running test cases 5-11
- O**
- objects
 - viewing objects with requirements 2-17
 - old Requirements Management Interface 10-2
 - opening a Signal Builder block 4-9
 - operating system requirements 1-3
- P**
- parameters for Model Verification blocks 4-3
 - properties
 - cv.cvdatagroup.name 12-182
 - cv.cvtestgroup.name 12-183
 - ModelAdvisor.Action.Description 12-161
 - ModelAdvisor.Action.Name 12-184
 - ModelAdvisor.Check.CallbackContext 12-157
 - ModelAdvisor.Check.CallbackFunction 12-158
 - ModelAdvisor.Check.CallbackStyle 12-159
 - ModelAdvisor.Check.Enable 12-169
 - ModelAdvisor.Check.ID 12-172
 - ModelAdvisor.Check.LicenseName 12-176
 - ModelAdvisor.Check.ListViewVisible 12-178
 - ModelAdvisor.Check.Result 12-187
 - ModelAdvisor.Check.Title 12-188
 - ModelAdvisor.Check.TitleTips 12-189
 - ModelAdvisor.Check.Value 12-192
 - ModelAdvisor.Check.Visible 12-195
 - ModelAdvisor.FactoryGroup.Description 12-162
 - ModelAdvisor.FactoryGroup.DisplayName 12-166
 - ModelAdvisor.FactoryGroup.ID 12-173
 - ModelAdvisor.FactoryGroup.MAObj 12-179
 - ModelAdvisor.Group.Description 12-163
 - ModelAdvisor.Group.DisplayName 12-167
 - ModelAdvisor.Group.ID 12-174
 - ModelAdvisor.Group.MAObj 12-180
 - ModelAdvisor.InputParameter.-
 - Description 12-164
 - ModelAdvisor.InputParameter.Entries 12-171
 - ModelAdvisor.InputParameter.Name 12-185
 - ModelAdvisor.InputParameter.Type 12-190
 - ModelAdvisor.InputParameter.Value 12-193
 - ModelAdvisor.ListViewParameter.-
 - Attributes 12-156
 - ModelAdvisor.ListViewParameter.Data 12-160
 - ModelAdvisor.ListViewParameter.Name 12-186
 - ModelAdvisor.Task.Description 12-165
 - ModelAdvisor.Task.DisplayName 12-168
 - ModelAdvisor.Task.Enable 12-170
 - ModelAdvisor.Task.ID 12-175
 - ModelAdvisor.Task.LicenseName 12-177
 - ModelAdvisor.Task.MAObj 12-181

ModelAdvisor.Task.Value 12-194
ModelAdvisor.Task.Visible 12-196

R

report

model coverage HTML options 5-24
understanding model coverage report 5-30

requirements

adding to test groups 4-25
for Model Verification block settings 4-24
for Requirements Management Interface for
DOORS 3-2
in generated code 2-59
viewing for test groups 4-27
viewing objects with 2-17

requirements consistency

Model Advisor checks 14-145

requirements links

two-way
creating 3-35
navigating 3-36

Requirements Management Interface

overview 2-2

Requirements Management Interface for DOORS

block type descriptions 3-21
definition of object in DOORS 3-14
from Simulink to DOORS 3-26
hierarchical numbers 3-21
object identifiers 3-21
opening the object in Simulink or
Stateflow 3-26
overview 3-2
saving formal modules 3-25
synchronizing models with DOORS 3-14
synchronizing objects with DOORS formal
module 3-14
viewing model elements with
requirements 3-28

viewing requirements 3-25

Requirements pane for Verification
Manager 4-24

requirements reports

content 3-30
creating default 3-30
customizing 2-26 3-31

rmi function 12-98

rmidocrename function 12-104

rminav function 12-106

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 5-54

sigrangeinfo function 12-151

simcovdemo

model coverage demo 5-11

slvndemo_mdladv

Model Advisor customization demo 8-20

surrogate modules 3-12

synchronization

customizing level of detail 3-19
Simulink model to DOORS surrogate
module 3-12

synchronizing models with DOORS 3-14

system requirements 1-3

IBM Rational DOORS 1-3

MATLAB 1-3

Microsoft Excel 1-3

Microsoft Word 1-3

operating system 1-3

Simulink 1-3

Stateflow 1-3

T

- tableinfo function 12-153
- test case commands 5-11
- 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 2-17