

# Simulink® Verification and Validation™ 2 User's Guide



MATLAB®  
& SIMULINK®

## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto: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 2004–2008 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](http://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](http://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)



## Getting Started

### 1

<b>Product Overview</b> .....	1-2
<b>System Requirements</b> .....	1-3
Operating System Requirements .....	1-3
Product Requirements .....	1-3
<b>Organization of This User's Guide</b> .....	1-4

## Managing Model Requirements

### 2

<b>What Is the Requirements Management Interface?</b> ...	2-2
<b>Configuring the Requirements Management Interface</b> .....	2-3
<b>Adding and Viewing Requirement Links</b> .....	2-4
Object and Document Types .....	2-4
Adding Requirement Links to an Object .....	2-7
Viewing Requirements Documents .....	2-14
Resolving the Document Path .....	2-17
Adding Requirement Links to Multiple Objects	
Simultaneously .....	2-18
Selection-Based Linking .....	2-22
<b>Linking to Custom Types of Requirements</b>	
<b>Documents</b> .....	2-28
Why Create a Custom Link Type? .....	2-28
Custom Link Type Registration .....	2-29
Built-In Link Types .....	2-29

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

## Managing Model Requirements with DOORS Software

### 3

<b>What Is the Requirements Management Interface for DOORS Software? .....</b>	<b>3-2</b>
<b>Configuring the Requirements Management Interface for DOORS Software .....</b>	<b>3-3</b>
Before You Begin .....	3-3
Installing DOORS Software Before RMI .....	3-3
Installing DOORS Software After RMI .....	3-3
Upgrading DOORS Software .....	3-4
Manual Installation for DOORS Software .....	3-4
<b>Starting the Requirements Management Interface for DOORS Software .....</b>	<b>3-6</b>
<b>Linking Objects to DOORS Requirements .....</b>	<b>3-9</b>

About Linkages Between a Simulink Model and DOORS	
Software .....	3-9
Creating a DOORS Requirement Object .....	3-9
Linking a Simulink or Stateflow Object to a DOORS	
Requirement .....	3-11

### **Synchronizing a DOORS Module with the Simulink**

<b>Model</b> .....	3-14
About Module Synchronization .....	3-14
Synchronizing a Model with the DOORS Software .....	3-16
Customizing the Level of Synchronization Detail .....	3-17
Customizing the DOORS Synchronization Settings .....	3-22
Linking Requirements to the DOORS Synchronized	
Module .....	3-24

### **Navigating Between Model Objects and DOORS**

<b>Requirements</b> .....	3-26
Viewing Model Elements with Requirements .....	3-26
Navigating from a Simulink Model to DOORS	
Requirements .....	3-28
Navigating from a DOORS Requirements to the Simulink	
Model .....	3-30

## **Managing Model Verification Blocks**

# **4**

<b>Using Model Verification Blocks</b> .....	4-2
<b>Using the Verification Manager</b> .....	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
<b>Managing Verification Requirements</b> .....	4-24

<b>Introduction to Model Coverage</b> .....	5-2
What Is Model Coverage? .....	5-2
How Model Coverage Works .....	5-2
Types of Model Coverage .....	5-2
Blocks That Receive Model Coverage .....	5-4
<b>Using Model Coverage</b> .....	5-7
Basic Workflow for Using Model Coverage .....	5-7
Creating and Running Test Cases .....	5-7
<b>Specifying Model Coverage Reporting Options</b> .....	5-11
Coverage Settings Dialog Box .....	5-11
Coverage Tab .....	5-12
Results Tab .....	5-17
Report Tab .....	5-18
Options Tab .....	5-22
<b>Understanding Model Coverage Reports</b> .....	5-25
About Model Coverage Reports .....	5-25
Summary Report Section .....	5-27
Details Report Section .....	5-28
Decisions Analyzed Table .....	5-30
Conditions Analyzed Table .....	5-31
MC/DC Analysis Table .....	5-31
Cumulative Coverage Reports .....	5-33
<b>N-Dimensional Lookup Table Report</b> .....	5-36
<b>Signal Range Analysis Report</b> .....	5-43
<b>Colored Simulink Diagram Coverage Display</b> .....	5-47
How Model Coverage Highlighting Works .....	5-47
Enabling the Colored Diagram Display .....	5-47
Displaying Model Coverage with Model Coloring .....	5-48
Accessing Coverage Information for Colored Blocks .....	5-50
<b>Using Model Coverage Commands</b> .....	5-52



About Model Coverage Commands .....	5-52
Creating Tests with cvtest .....	5-52
Running Tests with cvsim .....	5-54
Producing HTML Reports with cvhtml .....	5-55
Saving Test Runs to a File with cvsave .....	5-56
Loading Stored Coverage Test Results with cvload .....	5-56
Coverage Script Example .....	5-57

### Using Model Coverage Commands for Referenced

<b>Models</b> .....	5-59
Introduction .....	5-59
Creating a Test Group with cv.cvtestgroup .....	5-62
Running Tests with cvsimref .....	5-62
Extracting Results from cv.cvdatagroup .....	5-63

### Model Coverage for Embedded MATLAB Function

<b>Blocks</b> .....	5-64
Types of Model Coverage in Embedded MATLAB Function Blocks .....	5-64
Creating a Model with Embedded MATLAB Function Block Decisions .....	5-65
Understanding Embedded MATLAB Function Block Model Coverage .....	5-69

## Customizing the Model Advisor

# 6

<b>Customization Process and Guidelines</b> .....	6-3
<b>Demo and Code Example</b> .....	6-6
<b>Registering Custom Checks, Tasks, and Groups</b> .....	6-7
About Registering Custom Checks, Tasks, and Groups ...	6-7
Methods for Registering Custom Checks and Groups ....	6-8
Code Example: Methods for Registering Custom Checks and Tasks .....	6-8
<b>Creating Callback Functions for Checks</b> .....	6-10

About Check Callback Functions .....	6-10
Simple Check Callback Function .....	6-10
Detailed Check Callback Function .....	6-11
Check Callback Function with Hyperlinked Results .....	6-13
<b>Defining Custom Checks .....</b>	<b>6-17</b>
About Custom Checks .....	6-17
Properties of Custom Checks .....	6-17
Defining Where Custom Checks Appear .....	6-22
Code Example: Check Definition Function .....	6-23
<b>Defining Check Input Parameters .....</b>	<b>6-26</b>
About Input Parameters .....	6-26
Properties of Input Parameters .....	6-26
Specifying Input Parameter Layout .....	6-28
Code Example: Input Parameter Definition .....	6-29
<b>Defining Check List Views .....</b>	<b>6-31</b>
About List Views .....	6-31
Properties of List Views .....	6-31
Code Example: List View Definition .....	6-32
<b>Defining Check Actions .....</b>	<b>6-33</b>
About Actions .....	6-33
Properties of Actions .....	6-33
Action Callback Function .....	6-34
Code Example: Action Definition .....	6-34
Code Example: Action Callback Function .....	6-35
<b>Defining Custom Tasks .....</b>	<b>6-37</b>
About Custom Tasks .....	6-37
Properties of Custom Tasks .....	6-37
Defining Where Tasks Appear .....	6-40
Code Example: Task Definition Function .....	6-40
<b>Defining Custom Groups .....</b>	<b>6-41</b>
About Custom Groups .....	6-41
Defining Where Custom Groups Appear .....	6-41
Properties of Model Advisor Groups .....	6-42
Code Example: Group Definition .....	6-43

<b>Defining a Process Callback Function</b> .....	<b>6-45</b>
About Process Callback Functions .....	<b>6-45</b>
Process Callback Function Arguments .....	<b>6-45</b>
Code Example: Process Callback Function .....	<b>6-46</b>
<b>Formatting Model Advisor Outputs</b> .....	<b>6-48</b>
What Is the Model Advisor Formatting API? .....	<b>6-48</b>
Formatting Text .....	<b>6-48</b>
Formatting Lists .....	<b>6-49</b>
Formatting Tables .....	<b>6-50</b>
Formatting Paragraphs .....	<b>6-50</b>
Code Example: Model Advisor Formatted Output .....	<b>6-51</b>

## Function Reference

---

# 7

<b>Requirements Management Interface</b> .....	<b>7-2</b>
<b>Model Coverage</b> .....	<b>7-3</b>
<b>Model Advisor Customization API</b> .....	<b>7-4</b>
<b>Model Advisor Formatting API</b> .....	<b>7-5</b>

8

Block Reference

9

Model Advisor Checks

10

Simulink® Verification and Validation Checks .....	10-2
<b>DO-178B Checks .....</b>	<b>10-3</b>
Check safety-related optimization settings .....	10-4
Check safety-related diagnostic settings for solvers .....	10-8
Check safety-related diagnostic settings for sample time ..	10-11
Check safety-related diagnostic settings for signal data ..	10-14
Check safety-related diagnostic settings for parameters ..	10-17
Check safety-related diagnostic settings for data used for debugging .....	10-20
Check safety-related diagnostic settings for data store memory .....	10-22
Check safety-related diagnostic settings for type conversions .....	10-24
Check safety-related diagnostic settings for signal connectivity .....	10-26
Check safety-related diagnostic settings for bus connectivity .....	10-28
Check safety-related diagnostic settings that apply to function-call connectivity .....	10-30
Check safety-related diagnostic settings for compatibility .....	10-32
Check safety-related diagnostic settings for model referencing .....	10-34
Check safety-related model referencing settings .....	10-38
Check safety-related code generation settings .....	10-40
Check safety-related diagnostic settings for saving .....	10-47
Check for proper usage of For Iterator blocks .....	10-49

Check for proper usage of While Iterator blocks . . . . .	10-50
Display model version information . . . . .	10-52
Check for proper usage of blocks that compute absolute values . . . . .	10-53
Check for proper usage of Relational Operator blocks . . . .	10-55
<b>IEC 61508 Checks . . . . .</b>	<b>10-57</b>
Display model metrics and complexity . . . . .	10-58
Check for unconnected objects . . . . .	10-59
Check for fully defined interface . . . . .	10-60
Check for questionable blocks . . . . .	10-62
Check usage of Stateflow . . . . .	10-64
Display configuration management data . . . . .	10-67
Check usage of Simulink . . . . .	10-68
<b>MathWorks Automotive Advisory Board Checks . . . . .</b>	<b>10-73</b>
Check for difference in font and font sizes . . . . .	10-76
Check transition orientations in flow charts . . . . .	10-78
Check for display of nondefault block attributes . . . . .	10-79
Check for proper labeling on signal lines . . . . .	10-80
Check for propagated labels on signal lines . . . . .	10-82
Check default transition placement in Stateflow charts . .	10-84
Check setting Stateflow graphical function return value . .	10-85
Check for blocks not using one-based indexing . . . . .	10-86
Check for invalid file names . . . . .	10-88
Check for invalid model directory names . . . . .	10-90
Check for blocks that are not discrete . . . . .	10-91
Check for prohibited sink blocks . . . . .	10-92
Check for invalid port positioning and configuration . . . . .	10-93
Check for mismatches between names of ports and corresponding signals . . . . .	10-95
Check whether block names do not appear below blocks . .	10-96
Check for systems that mix primitive blocks and subsystems . . . . .	10-97
Check whether model has unconnected block input ports, output ports, or signal lines . . . . .	10-99
Check for improperly positioned Trigger and Enable blocks . . . . .	10-100
Check whether annotations have drop shadows . . . . .	10-101
Check whether tunable parameters specify expressions, data type conversions, or indexing operations . . . . .	10-102
Check whether Stateflow events are defined at the chart level or below . . . . .	10-104

Check whether Stateflow data objects with local scope are defined at the chart level or below .....	10-105
Check interface signals and parameters .....	10-106
Check for exclusive states, default states, and substate validity .....	10-107
Check optimization parameters for Boolean data types ...	10-109
Check model diagnostic settings .....	10-110
Check the display attributes of block names .....	10-114
Check icon display attributes for port blocks .....	10-115
Check whether subsystem block names include invalid characters .....	10-116
Check whether Inport and Outport block names include invalid characters .....	10-118
Check whether signal line names include invalid characters .....	10-120
Check whether block names include invalid characters ...	10-122
Check Trigger and Enable block port names .....	10-124
Check for Simulink diagrams that have nonstandard appearance attributes .....	10-125
Check visibility of port block names .....	10-128
Check for direction of subsystem blocks .....	10-130
Check for proper position of constants used in Relational Operator blocks .....	10-131
Check for entry format in state blocks .....	10-132
Check for use of tunable parameters in Stateflow .....	10-134
Check for proper use of Switch blocks .....	10-135
Check for proper use of signal buses and Mux block usage .....	10-136
Check for mismatches between Stateflow ports and associated signal names .....	10-138
Check for proper scope of From and Goto blocks .....	10-139
<b>Requirements Consistency Checks .....</b>	<b>10-140</b>
Identify requirement links with missing documents .....	10-141
Identify requirement links that specify invalid locations within documents .....	10-142
Identify selection-based links having descriptions that do not match their requirements document text .....	10-143
Identify requirement links with inconsistent path types and preferences .....	10-144

**A**

<b>Requirements Management Interface</b> .....	<b>A-2</b>
<b>Requirements Management Interface (DOORS Version)</b> .....	<b>A-2</b>
<b>Verification Manager</b> .....	<b>A-2</b>
<b>Model Coverage</b> .....	<b>A-2</b>





# 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 1-4

## Product Overview

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

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

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

# System Requirements

In this section...
“Operating System Requirements” 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 where the MATLAB® software supports the Java™ programming language (for HTML-based requirements documents only)

## Product Requirements

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

- MATLAB
- Simulink

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

- Stateflow

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

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

## Organization of This User's Guide

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

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

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

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

- Chapter 7, “Function Reference” — Provides a categorical list of functions used in executing and managing model coverage tests and reports from the MATLAB prompt. Automate your model coverage tests with scripts of MATLAB commands calling these functions.
- Chapter 8, “Functions — Alphabetical List” — Provides an alphabetical reference of functions used in executing and managing model coverage tests and reports from the MATLAB prompt.
- Chapter 9, “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

---

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

- “What Is the Requirements Management Interface?” on page 2-2
- “Configuring the Requirements Management Interface” on page 2-3
- “Adding and Viewing Requirement Links” on page 2-4
- “Linking to Custom Types of Requirements Documents” on page 2-28
- “Viewing Objects with Requirement Links” on page 2-45
- “Generating a Requirements Report” on page 2-48
- “Displaying the System Requirements in a Diagram” on page 2-50
- “Including Requirements with Generated Code” on page 2-56

# What Is the Requirements Management Interface?

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

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

Use the Requirements Management Interface to

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



## Configuring the Requirements Management Interface

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

```
rmi setup
```

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

# Adding and Viewing Requirement Links

In this section...
“Object and Document Types” on page 2-4
“Adding Requirement Links to an Object” on page 2-7
“Viewing Requirements Documents” on page 2-14
“Resolving the Document Path” on page 2-17
“Adding Requirement Links to Multiple Objects Simultaneously” on page 2-18
“Selection-Based Linking” on page 2-22

## Object and Document Types

You can add requirements to the following types of objects:

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

---

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

---

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

- Text
- HTML
- Microsoft Word
- Microsoft Excel

- PDF

You can also link to an item in the Telelogic DOORS software (see “Linking Objects to DOORS Requirements” on page 3-9), or register your own custom type of documents to link to (see “Linking to Custom Types of Requirements Documents” on page 2-28).

## Location Types

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

Document Type	Location Options
Text	<ul style="list-style-type: none"> <li>• <b>Search text</b> — Type a string in the <b>Location</b> text field. The Requirements Management Interface searches for the first occurrence of the text string within the document. This search is not case sensitive.</li> <li>• <b>Line number</b> — Type a line number in the <b>Location</b> text field. The Requirements Management Interface makes a link to the specified line.</li> </ul>
HTML	<p>You can link only to a named anchor.</p> <p>For example, if you define the anchor</p> <pre data-bbox="620 1065 1222 1095">&lt;A NAME=valve_timing&gt; ...contents... &lt;/A&gt;</pre> <p>in your HTML requirements document, you can enter <code>valve_timing</code> in the <b>Location</b> text field or click the <b>Document Index</b> tab to select <code>valve_timing</code> from an automatically generated list of anchors in the document.</p>

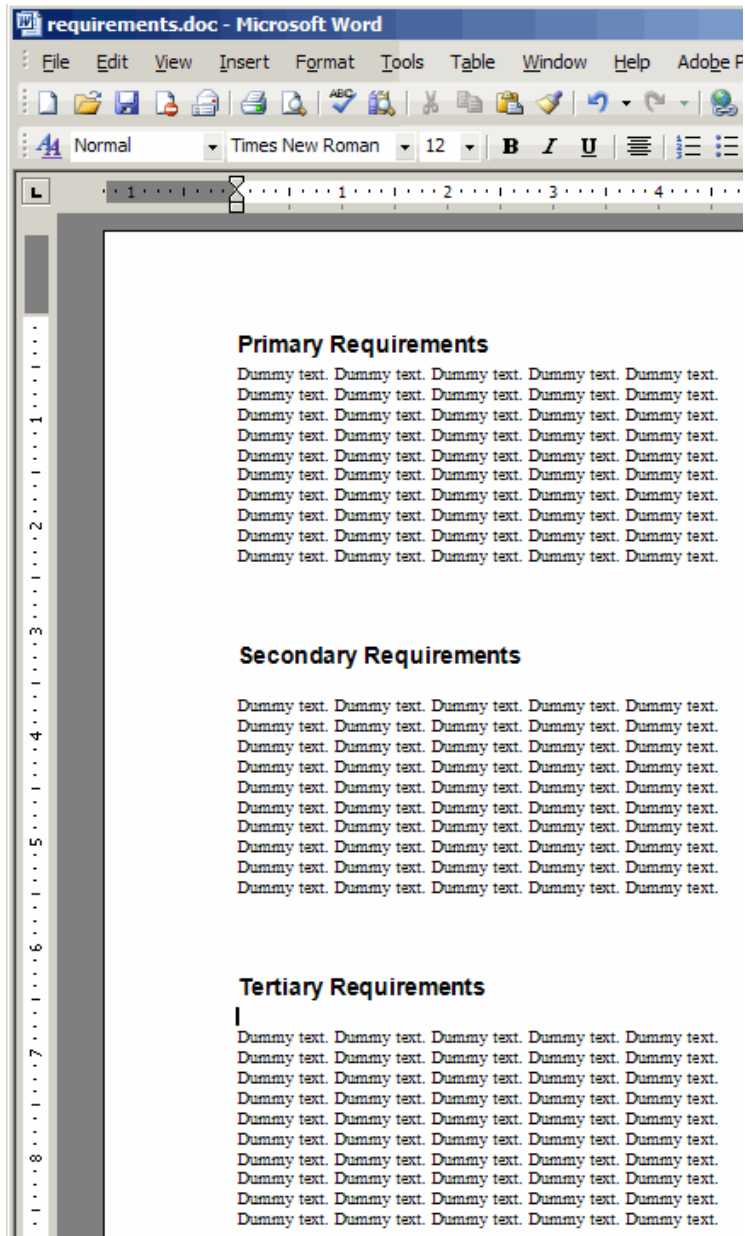
<b>Document Type</b>	<b>Location Options</b>
Microsoft Word	<ul style="list-style-type: none"><li>• <b>Search text</b> — Type a string in the <b>Location</b> text field. The Requirements Management Interface searches for the first occurrence of the text string within the document. This search is not case sensitive.</li><li>• <b>Named item</b> — Link to a bookmark within the document. The Requirements Management Interface automatically generates a document index based on its headings and bookmarks, or you can type the name in the <b>Location</b> text field.</li><li>• <b>Page/item number</b> — Type a page number in the <b>Location</b> text field. The Requirements Management Interface makes a link to the top of the page.</li></ul>
Microsoft Excel	<ul style="list-style-type: none"><li>• <b>Search text</b> — Type a string in the <b>Location</b> text field. The Requirements Management Interface searches for the first occurrence of the text string within the document. This search is not case sensitive.</li><li>• <b>Named item</b> — Link to a named item within the document (defined in the Excel® software using <b>Insert &gt; Name</b>). Type the name in the <b>Location</b> text field.</li><li>• <b>Sheet range</b> — Type a cell number or a range of cells (such as C5:D7) in the <b>Location</b> text field. The Requirements Management Interface makes a link to the specified cell or cells.</li></ul>

Document Type	Location Options
PDF	<ul style="list-style-type: none"> <li>• <b>Named item</b> — Link to a bookmark within the document. The Requirements Management Interface automatically generates a document index based on its headings and bookmarks, or you can type the bookmark name in the <b>Location</b> text field.</li> <li>• <b>Page/item number</b> — Type a page number in the <b>Location</b> text field. The Requirements Management Interface makes a link to the top of the page.</li> </ul>
Web Browser URL	You can link to a URL location only. Type the URL location string in the <b>Document</b> text field. When you follow the link, the document opens in a Web browser.

## Adding Requirement Links to an Object

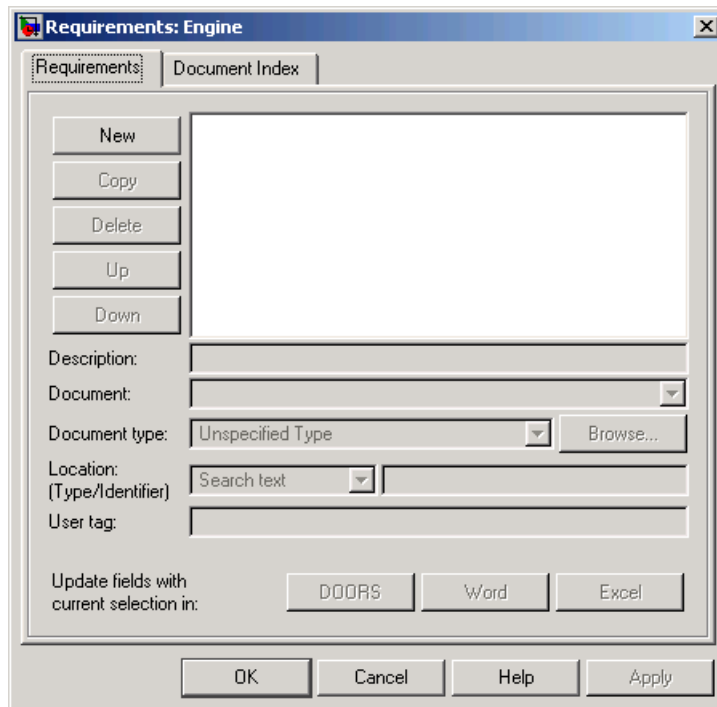
You use the Requirements dialog box to associate a requirements document with a requirements object. You can link a particular location in an existing Microsoft Word or HTML document to a block in a Simulink diagram or a Stateflow object in a Stateflow chart. In this procedure, you add three requirement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

- 1 Create and save the Microsoft Word document `requirements.doc` with the following format. Style the header lines (Primary Requirements, Second Requirements, Tertiary Requirements) as **Heading 1** in Microsoft Word.



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

The Requirements dialog box for the Engine block appears.



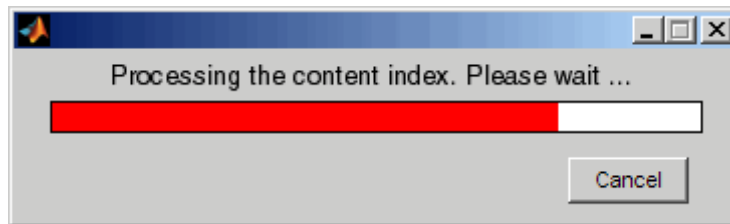
- 4 In the Requirements dialog box, click **New** to add a new requirement.

Fields and tools of the Requirements dialog box are now enabled for the new requirement.

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

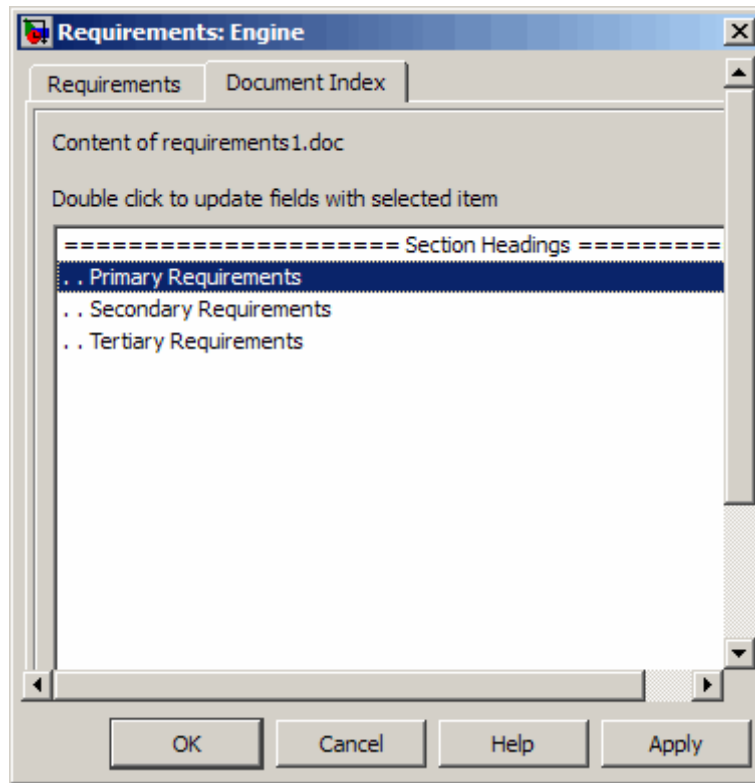
The **Document type** field is now set to Microsoft Word. If you specify the document type in the **Document type** field prior to browsing for the requirements document, only the files of the appropriate type appear. If you set **Document type** to Unspecified Type, all files are listed.

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



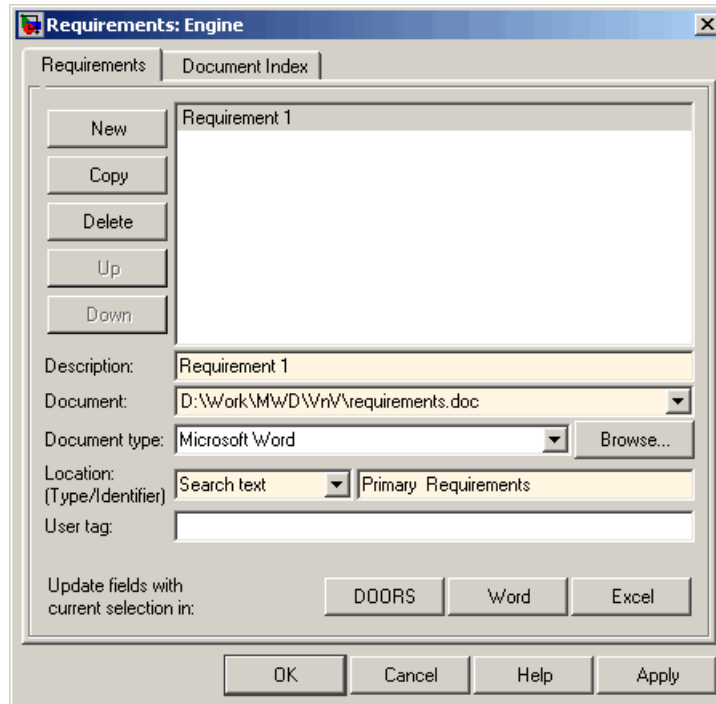
- 8 Select **Primary Requirements** from the automatically generated list of headings and bookmarks in the document.





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

Select **Search** text from the **Location** drop-down list and enter **Primary Requirements** in the text field. The search text feature is not case-sensitive; **primary requirements** and **PRIMARY REQUIREMENTS** work the same in this example.



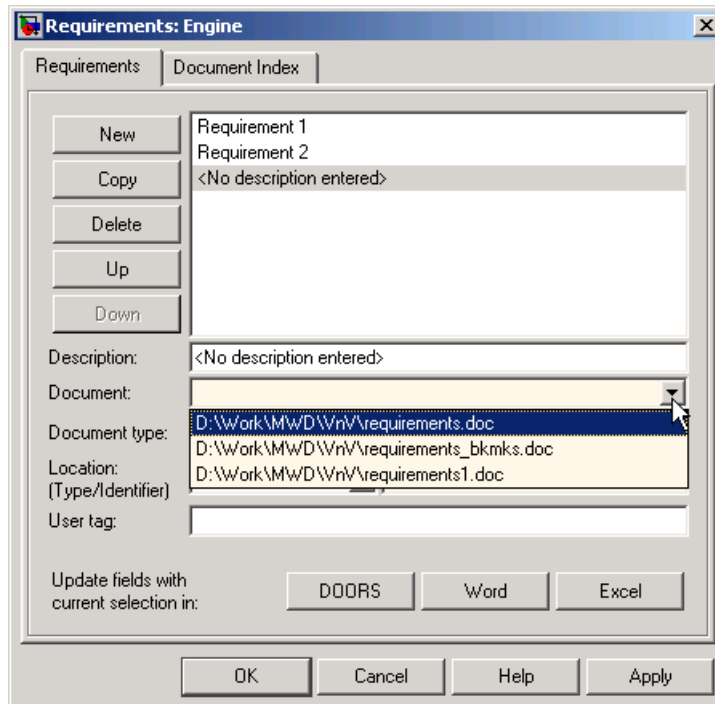
- 9 To provide additional details about the current requirement, enter text in the **User tag** field. Entering text in the **User tag** field is optional.
- 10 Click **Apply**.
- 11 Select Requirement 1 and click **Copy** to create a copy of Requirement 1 as a new requirement.
- 12 In the **Description** field, replace Copy of Requirement 1 with Requirement 2, and link Requirement 2 to the text “Secondary Requirements” in requirements.doc.

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

Tool Button	Description
Delete	Deletes the requirement.

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

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



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

---

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

---

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

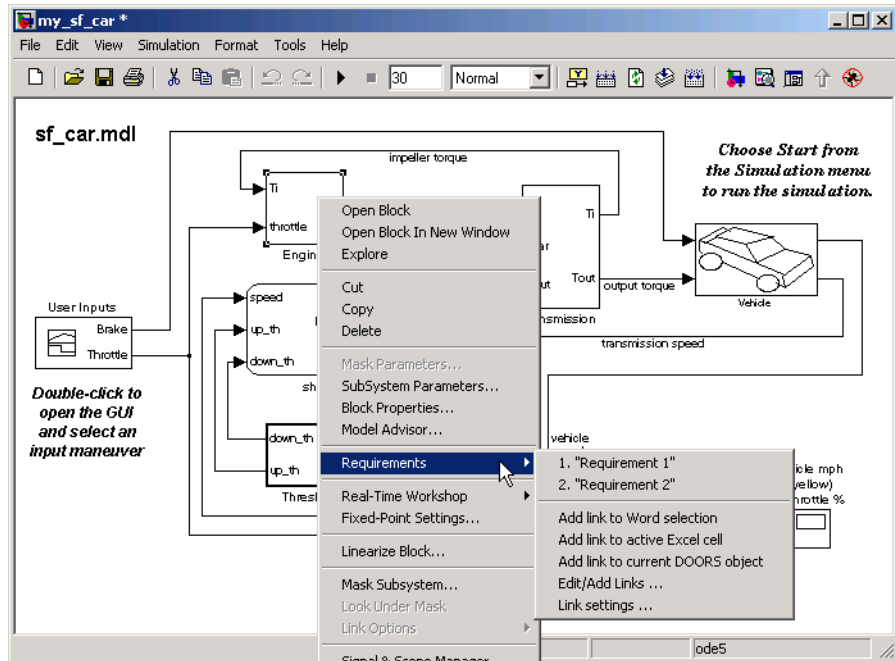
You use this model in “Viewing Requirements Documents” on page 2-14.

### Viewing Requirements Documents

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

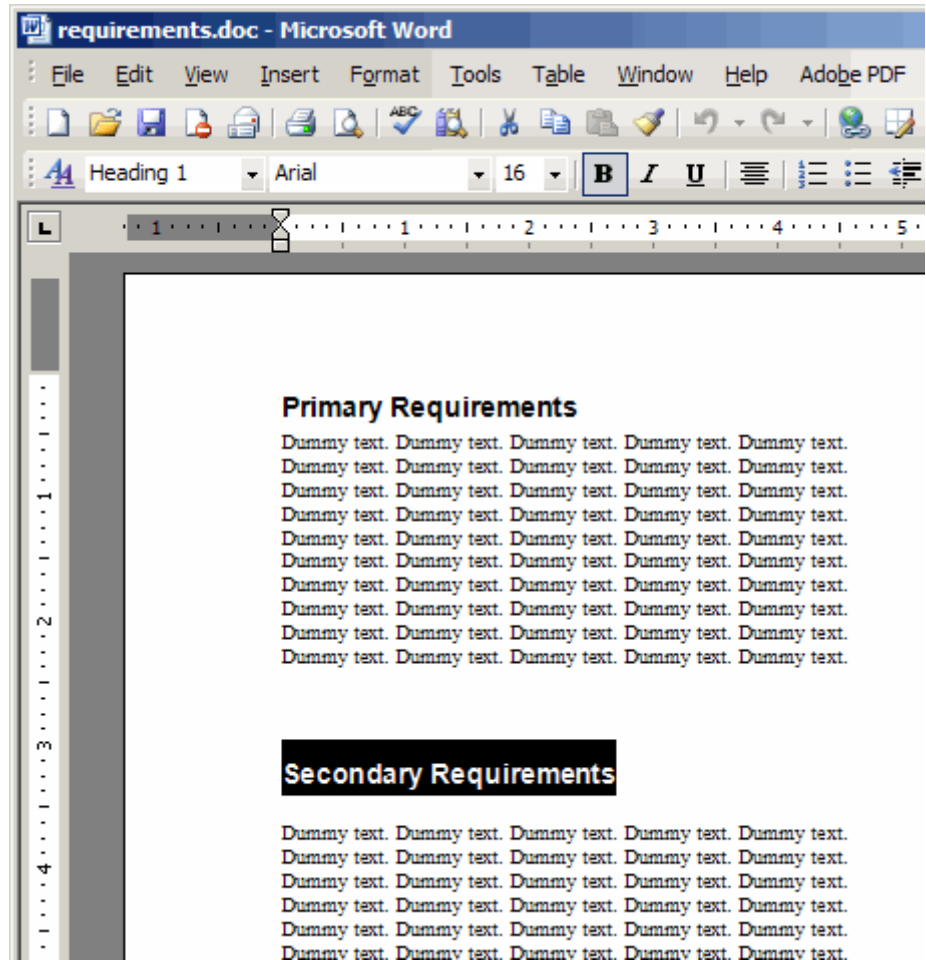
- 1 In the `my_sf_car` model, right-click the Engine block and select **Requirements** from the context menu.

The requirements you added now appear as submenu selections.



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

requirements.doc opens in its editor, the Microsoft Word software, and the first occurrence of Secondary Requirements is highlighted.



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

- 3 Keep requirements.doc and the my\_sf\_car model open; you will use them in “Adding Requirement Links to Multiple Objects Simultaneously” on page 2-18.

## Resolving the Document Path

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

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

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

### Relative Path Specified Example

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

### No Path Specified Example

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

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

### Absolute Path Specified Example

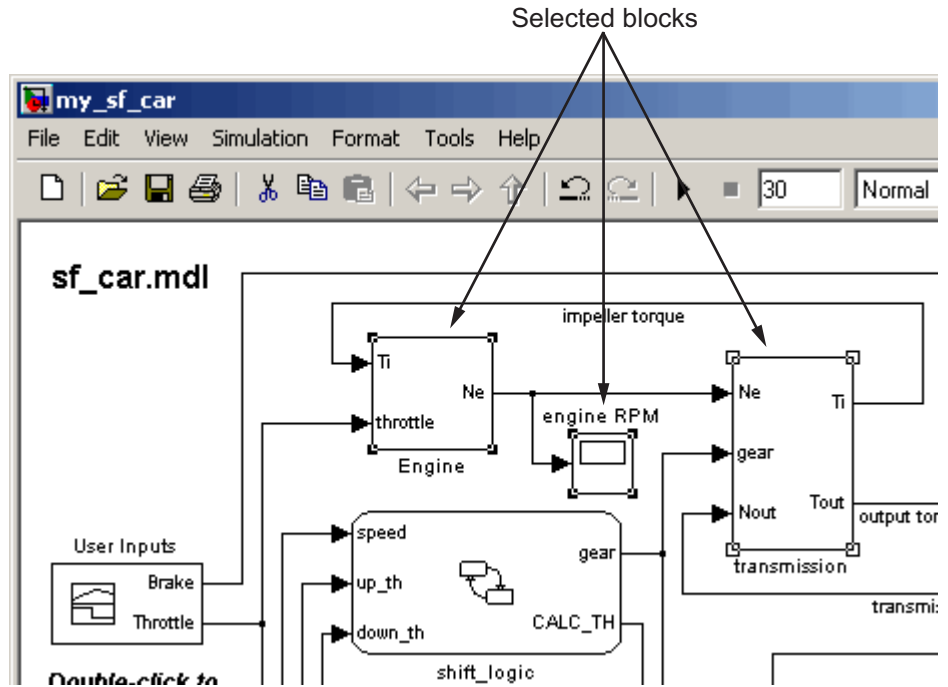
Current MATLAB directory:	C:\work\scratch
Model file:	C:\work\models\controllers\pid.mdl
Requirements document:	C:\work\reqs\pid.html
Documents searched for:	C:\work\reqs\pid.html

### Adding Requirement Links to Multiple Objects Simultaneously

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

- 1 In the my\_sf\_car model, select the Engine, engine RPM, and transmission blocks.



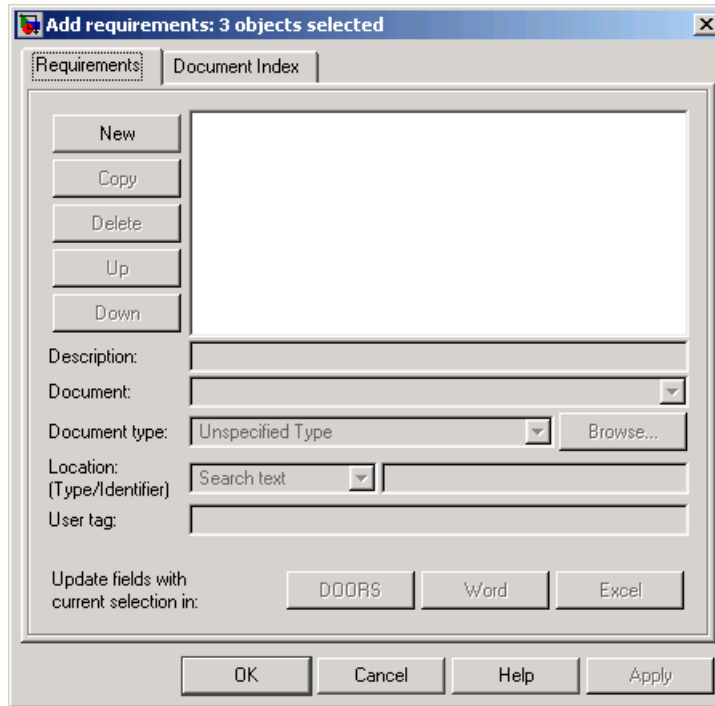


You can select multiple Simulink blocks or Stateflow objects in one of the following ways:

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

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

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



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

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

---

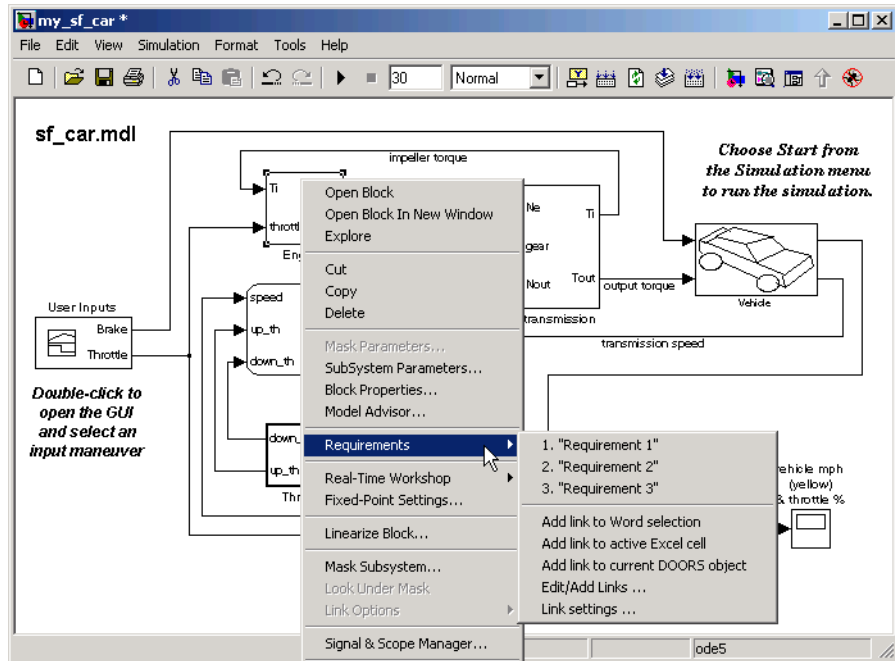
**Note** When you add a requirement link to a block such as a subsystem, the requirement is not added to children of the block.

---

- 4 Click **Apply** to apply the requirement links.
- 5 Click **OK** to close the Requirements dialog box.
- 6 In the my\_sf\_car model, click outside the three objects to deselect them.

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

The Engine block now has three requirements, as shown.



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

**9** Save the my\_sf\_car model.

## Deleting All Requirement Links for Multiple Objects Simultaneously

To delete the existing requirements for a group of selected blocks, right-click any of a group of selected blocks and, from the resulting context menu, select **Requirements > Delete All**. This deletes all the requirement links for all the selected blocks, whether they were added individually or as a group.

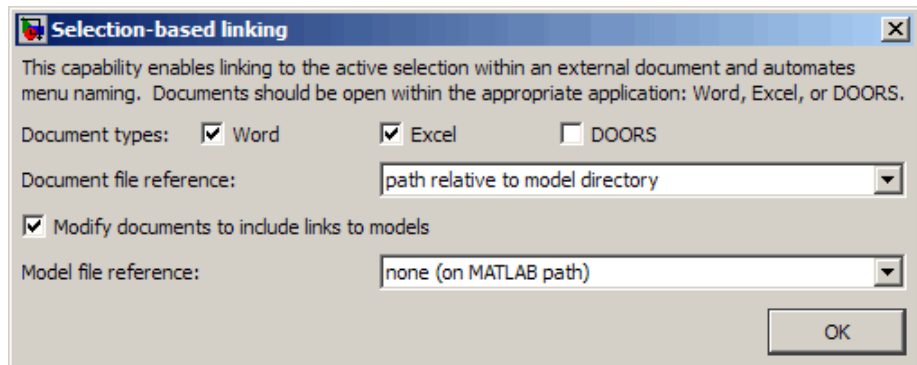
### Selection-Based Linking

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

### Configuring Selection-Based Linking

Use the following procedure to configure selection-based linking:

- 1 Open the my\_sf\_car model, if necessary.
- 2 In the model window, select **Tools > Requirements > Link settings**. The Selection-based linking dialog box opens.



- 3 Select the check box next to **Word** if necessary.
- 4 Specify the following preferences on the Selection-based linking dialog box:
  - In the **Document file reference** drop-down list, specify how to store the document location. For more information, see “Resolving the Document Path” on page 2-17.
  - To create two-way links, select **Modify documents to include links to models**. Two-way linking embeds an ActiveX® control in your requirements document, which lets you navigate from the requirements document to the Simulink model and vice versa.

- If you create two-way links the **Model file reference** drop-down list, specify how to locate the model when you navigate from a requirements document to the model using one of these two options:
  - **none** (on MATLAB path) — The model is located on the MATLAB path.
  - **Absolute** — Use the absolute path to locate the model.

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

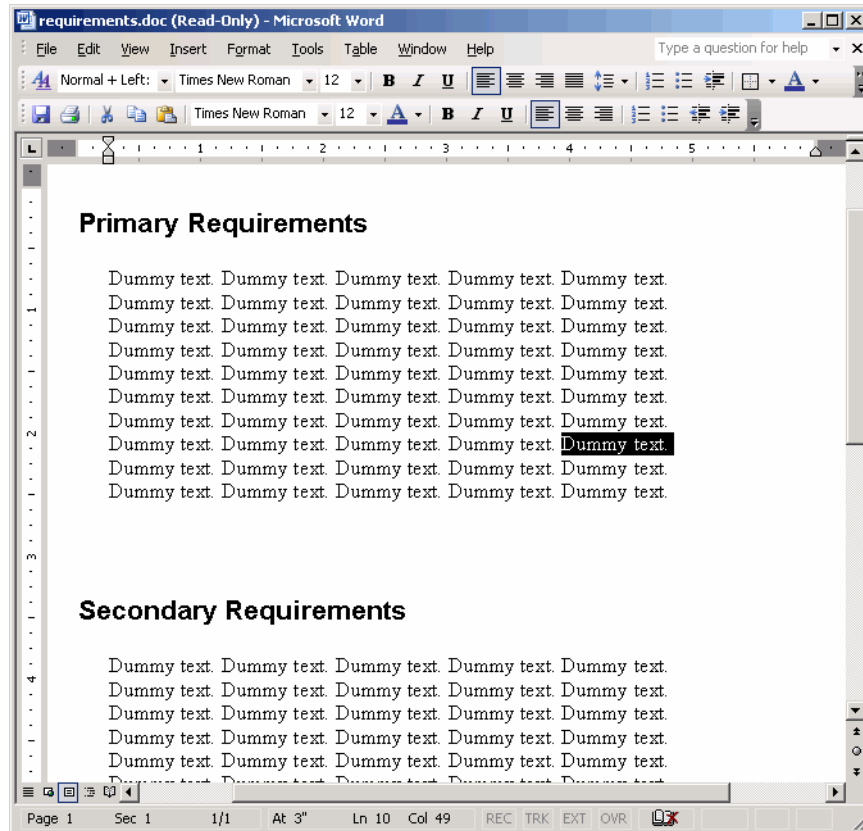
### **Making Selection-Based Links**

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

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

requirements.doc opens in a Microsoft Word window.

- 2** Select a portion of the text that documents the desired requirement. For this example, select a “Dummy text.” string.



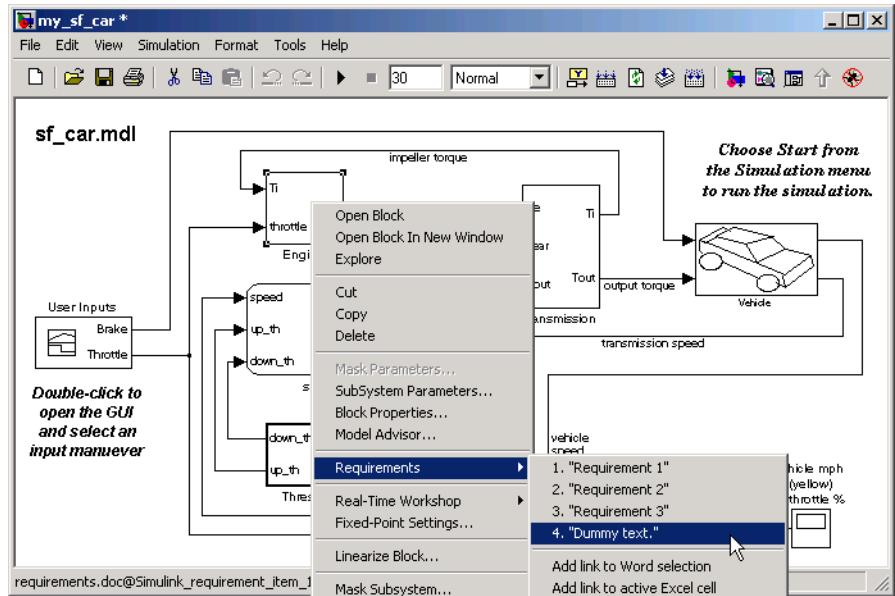
- 3 In the Simulink model, right-click the Engine block and select **Requirements > Add link to Word selection**.

---


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

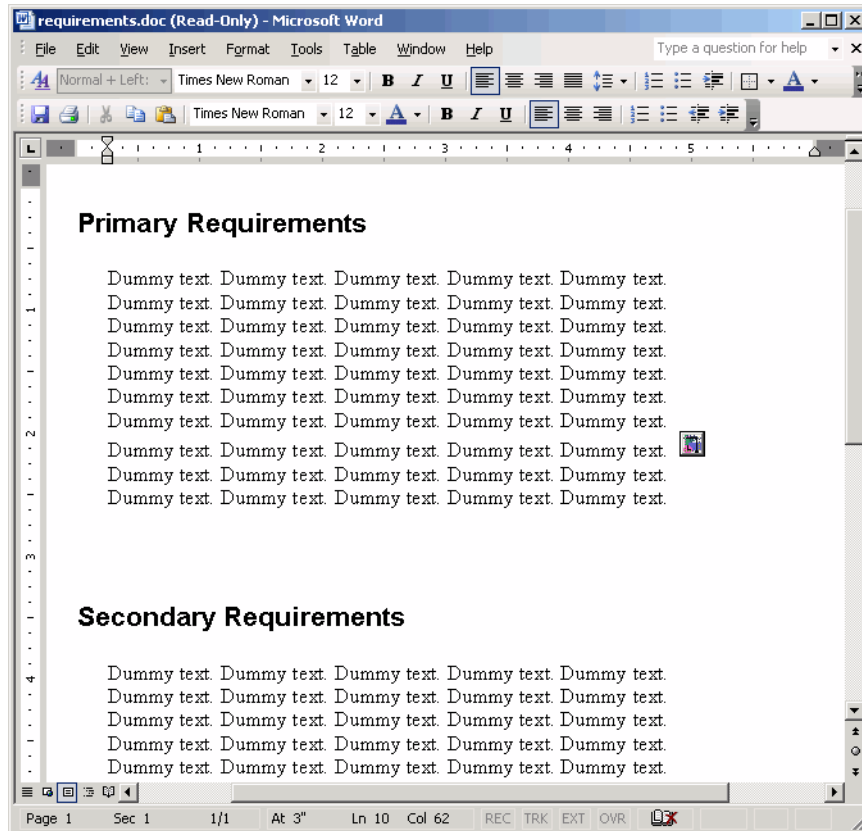
---

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



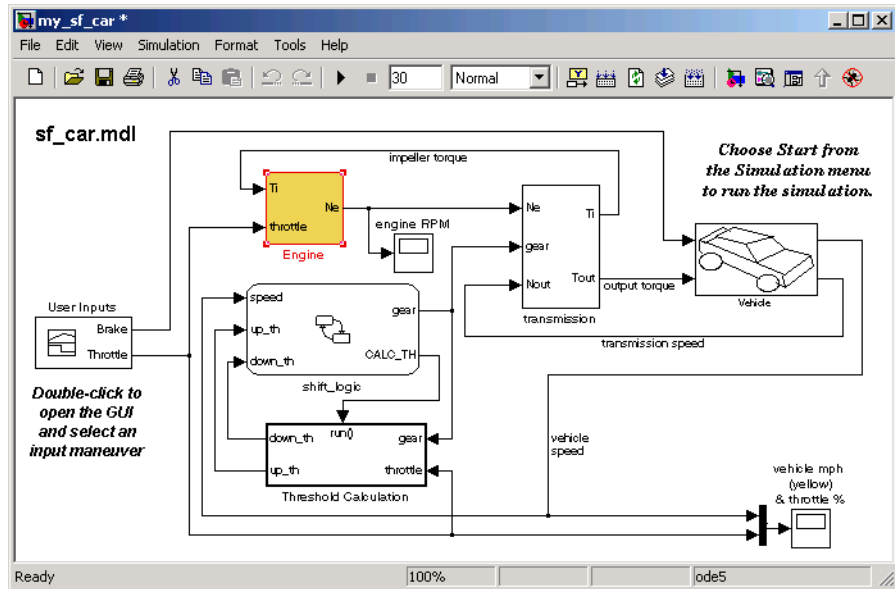
If you select **Requirements > Dummy text.**, a Microsoft Word window opens with the requirements document scrolled to the appropriate text.

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



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





## Linking to Custom Types of Requirements Documents

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

### Why Create a Custom Link Type?

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

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

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

Custom link types enable you to

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

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

## Custom Link Type Registration

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

```
rmi register mytargetfilename
```

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

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

```
rmi unregister mytargetfilename
```

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

## Built-In Link Types

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

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

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

### Link Properties

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

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

### Link Type Properties

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

Property	Description
Registration	The name of the M-file that creates the link type. This is stored in the Simulink model.
Label	A string to identify this link type. It is displayed on the <b>Document type</b> drop-down list in the Requirements dialog box for a Simulink or Stateflow object.
IsFile	A Boolean property that indicates if the linked documents are files within the computer file system. If a document is a file, then the standard method for resolving the path is used and the standard file selection dialog is invoked when the user clicks the <b>Browse</b> button in the Requirements dialog box.

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

## Creating a Custom Link Requirement Type

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

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

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

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

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

```
% >> which demo_req_1.abc

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

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

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

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

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

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

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

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

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

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

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

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



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

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

depths = [];
```

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

```
rmi register rmicustabcinterface
```

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

- 3** Create a text file with the .abc extension, containing several requirement items marked by the Requirement:: string, as described above. For your convenience, an example of such a file is included in the installation. It is named demo\_req\_1.abc, and is located in *matlabroot*\toolbox\slvnx\rmidemos. 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.

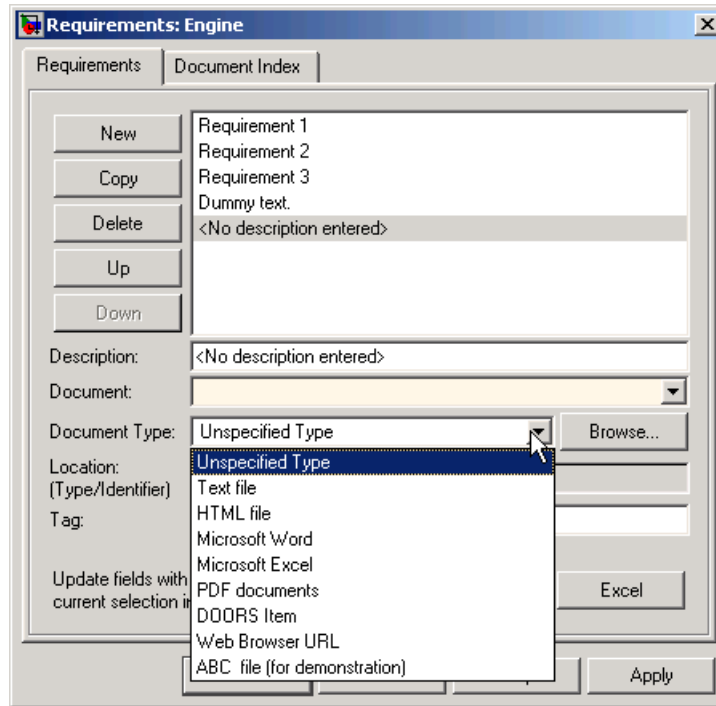
The user-selectable climb rate is always a positive number if the current altitude is above the target altitude the actual target climb rate is the negative of the user setting.

<END "Glide Slope Coupled">

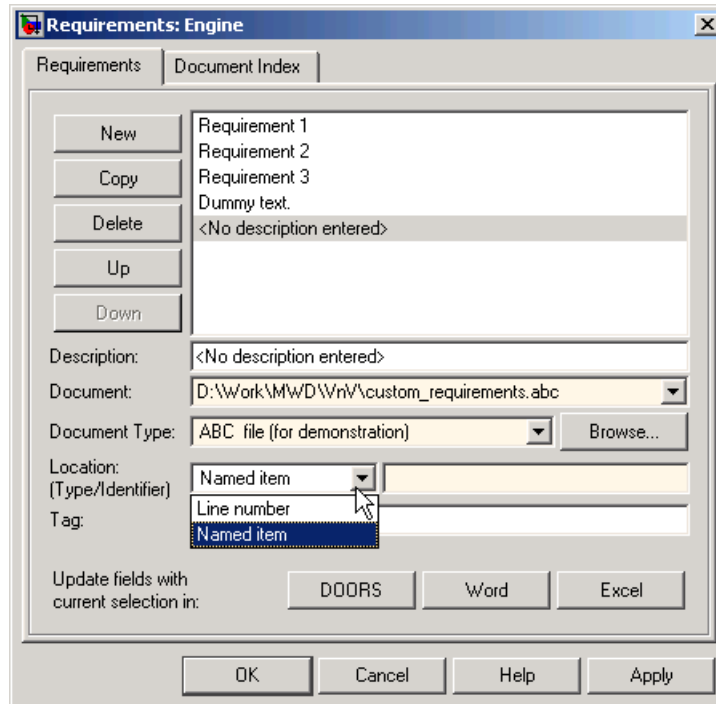
- 4 Open the model my\_sf\_car.
- 5 Right-click the Engine block and, from the resulting pop-up menu, select **Requirements > Edit/Add Links**.

The Requirements dialog box appears.

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



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



### Creating a Document Index

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

The following code generates an index for the ABC files.

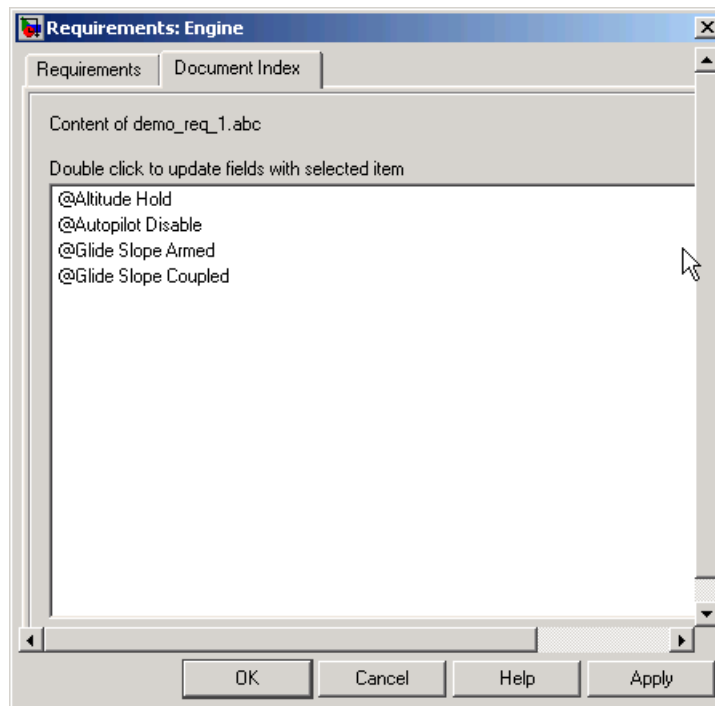
```
function [labels, depths, locations] = ContentsFcn(filePath)
% Read the entire M-file into a variable
fid = fopen(filePath,'r');
contents = char(fread(fid));
fclose(fid);
```

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

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

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

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



### **Navigating to Simulink Models from External Documents**

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

#### **Providing Unique Object Identifiers**

Whenever you create a requirement link for a Simulink or Stateflow object, a globally unique identifier is created for that object. This identifier is used to identify the object and does not change if the object is renamed or moved or when requirement links are added or deleted. Although the unique identifier is only used to resolve an object within a model, the identifier is globally unique and should not collide with identifiers in other models unless the two models derive from the same original model. Unique object identifiers have formats like GIDa\_cd14afcd\_7640\_4ff8\_9ca6\_14904bdf2f0f.

#### **Using the `rmiobjnavigate` Utility**

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

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

#### **Determining the Navigation Command**

Once you have created a requirement link for a Simulink or Stateflow object, you can find the appropriate navigation command string by using the `rmi` function at the MATLAB prompt. 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);
```



You need to 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**

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

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

### **Typical Code Sequence for Establishing Two-Way Links**

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

In generic terms, use the following process:

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

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

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

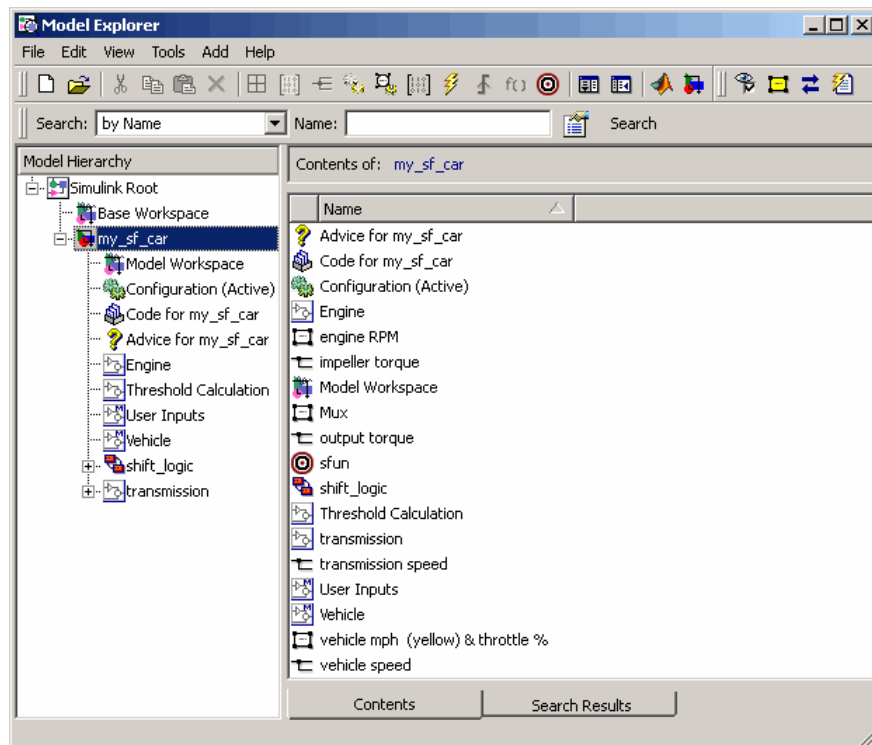
You can use the code for selection-based linking to the Word, Excel, and Telelogic DOORS software as an example of this type of automation. The files are contained in `matlabroot\toolbox\slvnv\reqmgt\private`:


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

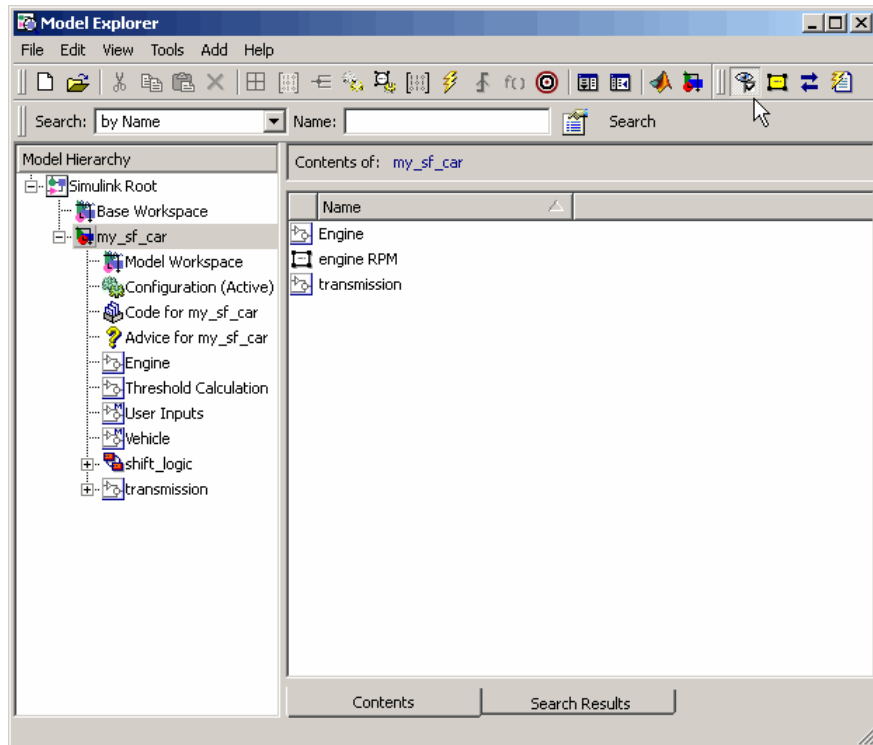
## Viewing Objects with Requirement Links

After you have added requirements to blocks in a model, you can change the view in the Model Explorer window to show only objects that have requirements associated with them. In “Adding Requirement Links to an Object” on page 2-7 and “Adding Requirement Links to Multiple Objects Simultaneously” on page 2-18, you add requirements to the Engine, Engine RPM, and transmission blocks in the model you save as `my_sf_car`. Use the following procedure to highlight these objects in the Model Explorer and Simulink model window:

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




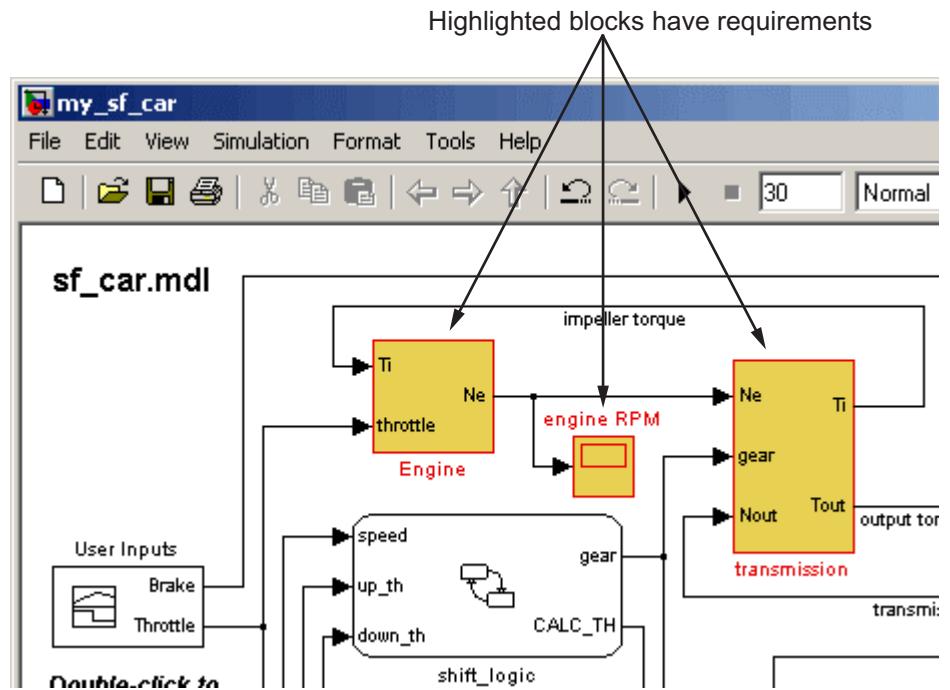
- 4 In the **Model Explorer** toolbar, select the Display Objects with Linked Requirements tool .



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

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

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



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

# Generating a Requirements Report

After you have added requirements to a model, you can generate a report on all the requirements associated with the model and its blocks. In “Adding Requirement Links to an Object” on page 2-7 and “Adding Requirement Links to Multiple Objects Simultaneously” on page 2-18, you add requirements to the Engine, engine RPM, and transmission blocks in the model you save as `my_sf_car`. Use the following procedure to generate a requirements report:

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

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

**Model Requirements**

scowan

21-Jan-2008 20:20:36

**Table of Contents**

[1. System - Engine](#)  
[2. System - my\\_sf\\_car](#)  
[3. System - transmission](#)

**List of Tables**

1.1. [System Requirements](#)  
2.1. [Blocks that have requirements](#)  
3.1. [System Requirements](#)

**Chapter 1. System - Engine**

**Table 1.1. System Requirements**

Description	Document	ID
Requirement 1	<a href="#">requirements.doc</a>	Primary Requirements
Requirement 2	<a href="#">requirements.doc</a>	Secondary Requirements
Requirement 3	<a href="#">requirements.doc</a>	Tertiary Requirements

**Chapter 2. System - my\_sf\_car**

*Choose to run*

Done

- 3 Save the report with a meaningful name. Select **File > Save As**, enter the file name, and click **Save**.

## Displaying the System Requirements in a Diagram

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

### About the System Requirements Block

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

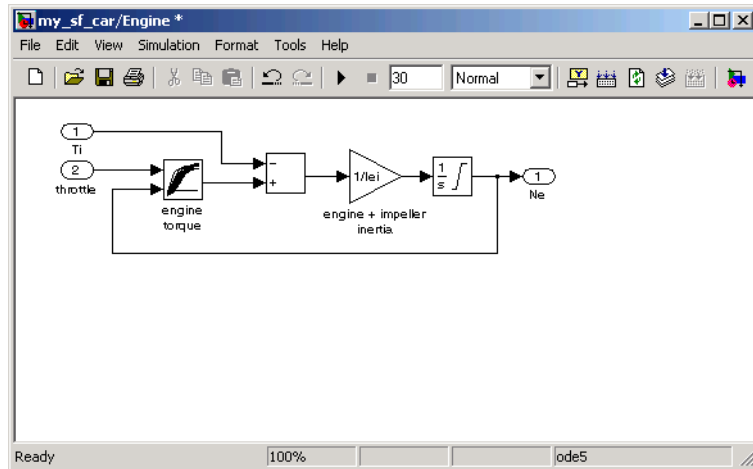
Once you place the System Requirements block in a Simulink diagram, it automatically lists the requirements associated with the model or subsystem depicted in the current diagram. It does not list requirements associated with individual blocks in the diagram.


### Adding the System Requirements Block

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

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





- 3 Click the Library Browser tool .

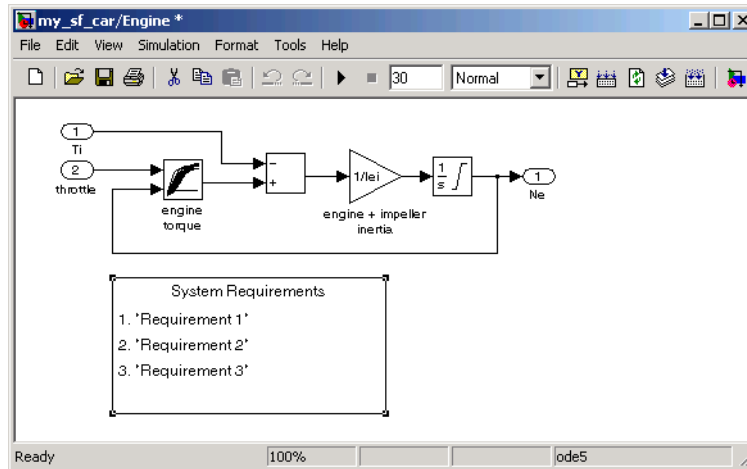
The Simulink Library Browser opens.

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

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

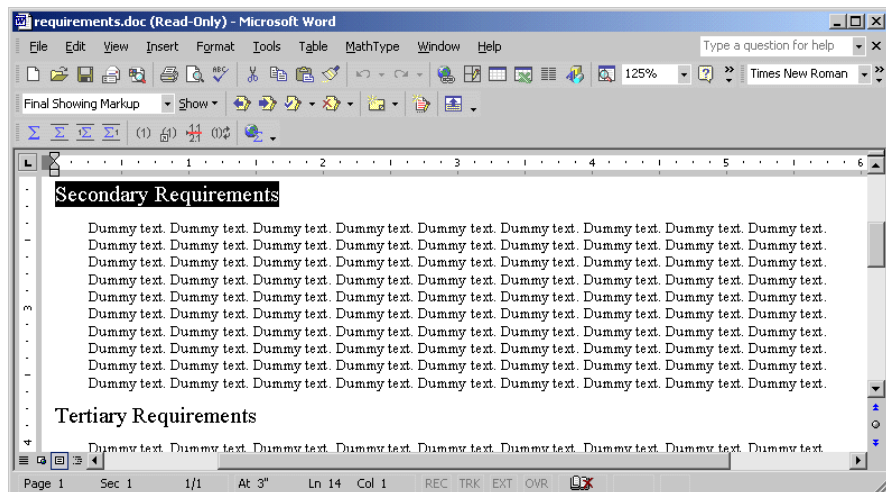
- 5 Select the System Requirements block in the right pane of the Simulink Library Browser and drag it to an empty space in the Engine diagram.

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



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

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



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

---

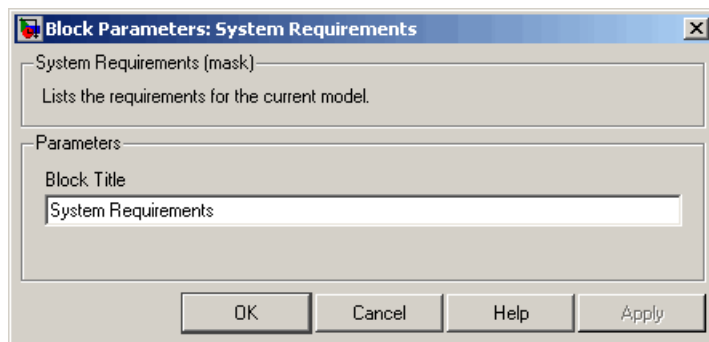
**Note** The System Requirements block automatically lists all the system requirements for the current model or subsystem. You cannot have more than one System Requirements block in a diagram.

---

## Renaming the System Requirements Block

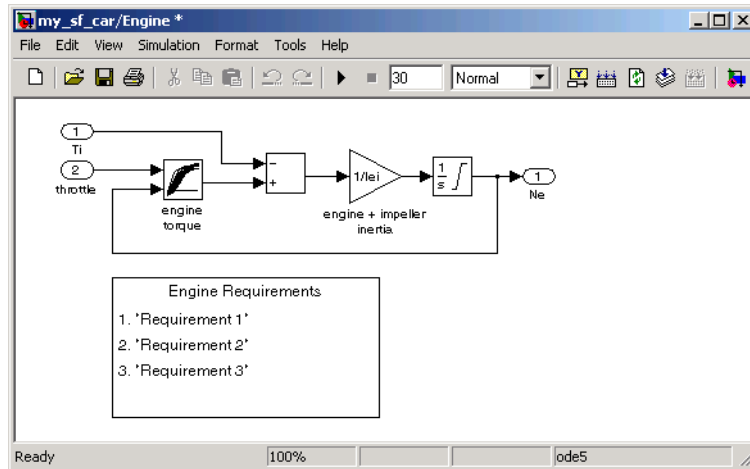
By default, the list of the system requirements in a diagram appears under a heading **System Requirements**. You can change the heading by renaming the System Requirements block in the diagram, as follows:

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



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

The requirements heading in the diagram is updated as shown.

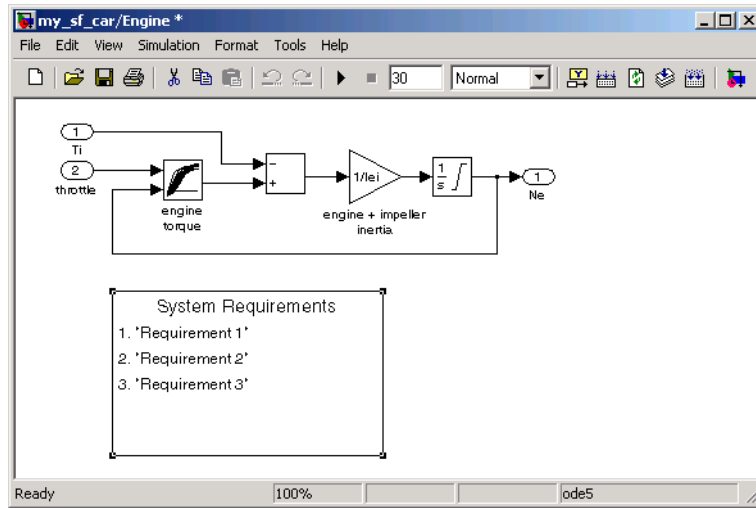


### Changing Fonts for the System Requirements Block

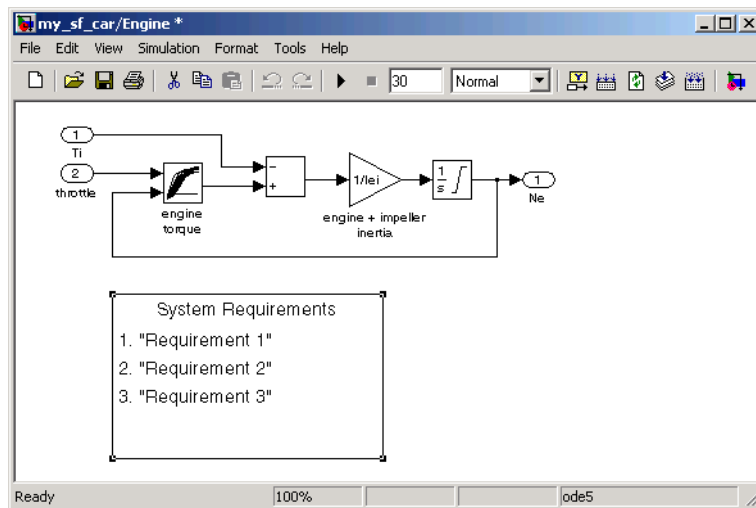
The System Requirements block is implemented using a set of empty subsystems. Because of this, occasionally the appearance is not refreshed correctly, for example, when you make a change to the font style or size. You can easily fix this problem by double-clicking the top label for the block, which causes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

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

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



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



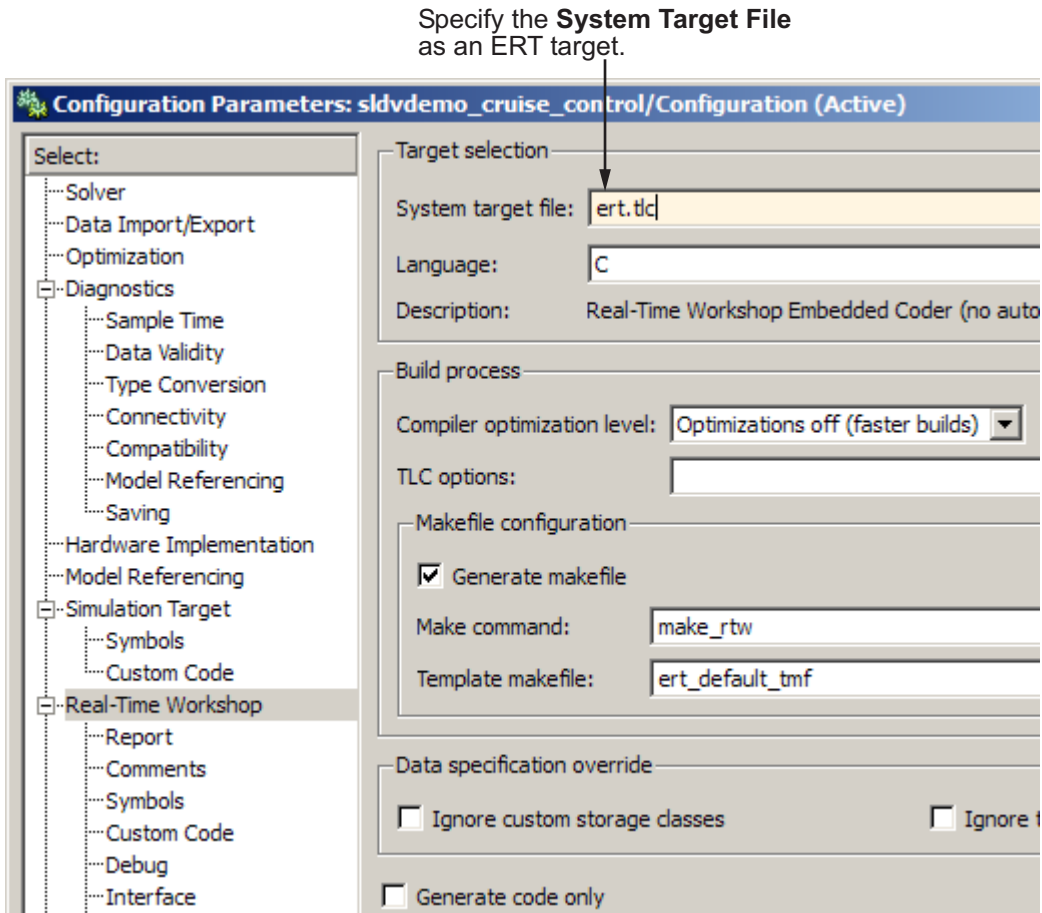
### Including Requirements with Generated Code

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

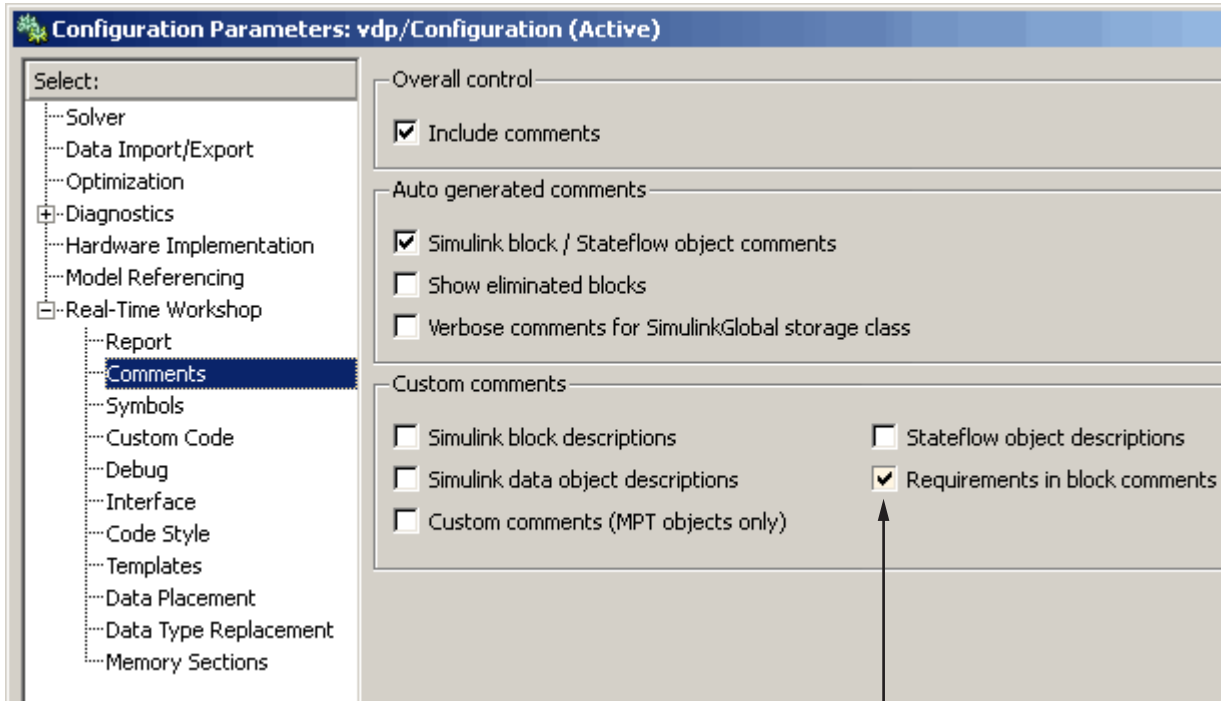
To specify that requirements be included in the generated code of an ERT target, do the following:

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

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



- 4 In the **Select** pane, under Real-Time Workshop, select **Comments**.
- 5 In the **Custom comments** section on the right, select the **Requirements in block comments** check box, as shown.



Include requirements descriptions  
in generated code comments

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

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



# Managing Model Requirements with DOORS Software

---

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

- “What Is the Requirements Management Interface for DOORS Software?” on page 3-2
- “Configuring the Requirements Management Interface for DOORS Software” on page 3-3
- “Starting the Requirements Management Interface for DOORS Software” on page 3-6
- “Linking Objects to DOORS Requirements” on page 3-9
- “Synchronizing a DOORS Module with the Simulink Model” on page 3-14
- “Navigating Between Model Objects and DOORS Requirements” on page 3-26

### **What Is the Requirements Management Interface for DOORS Software?**

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

# Configuring the Requirements Management Interface for DOORS Software

## In this section...

“Before You Begin” on page 3-3

“Installing DOORS Software Before RMI” on page 3-3

“Installing DOORS Software After RMI” on page 3-3

“Upgrading DOORS Software” on page 3-4

“Manual Installation for DOORS Software” on page 3-4

## Before You Begin

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

## Installing DOORS Software Before RMI

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

## Installing DOORS Software After RMI

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

## Upgrading DOORS Software

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

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

**1** Navigate to the directory `Telelogic\DOORS_8.0\lib\dx1\startupFiles`.

**2** Open the file `copiedFromDoors7.dxl` with a text editor.

**3** Comment out the line:

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

It should now look like this:

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

**4** Save and close the file.

**5** Start the DOORS and MATLAB software.

**6** Run the setup script.

## Manual Installation for DOORS Software

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

**1** Close the DOORS software if it is running.

**2** Copy the following files from `matlabroot\toolbox\slvnx\reqmgt` to the `<doors>\lib\dx1\addins` directory:

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

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

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

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

Replace any existing versions of these files.

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

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

## Starting the Requirements Management Interface for DOORS Software

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

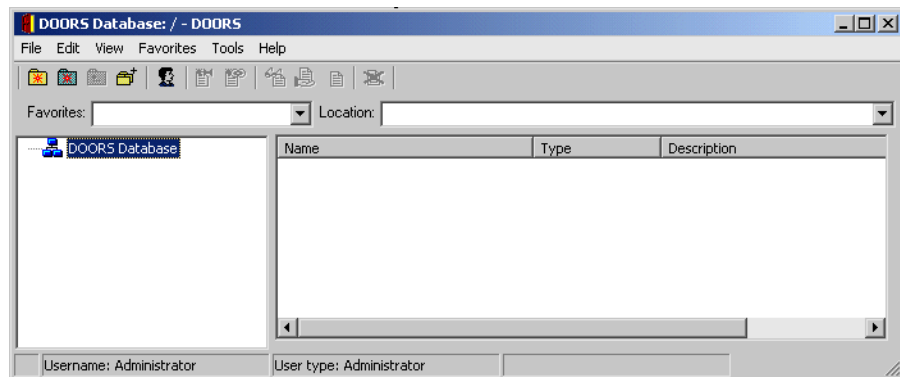
- 1 Start the MATLAB software on your DOS or UNIX system with the following command:

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

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

- 2 Start your DOORS software.

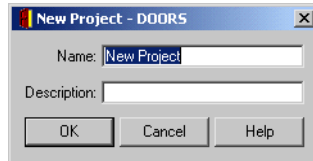
The DOORS Database window appears.



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

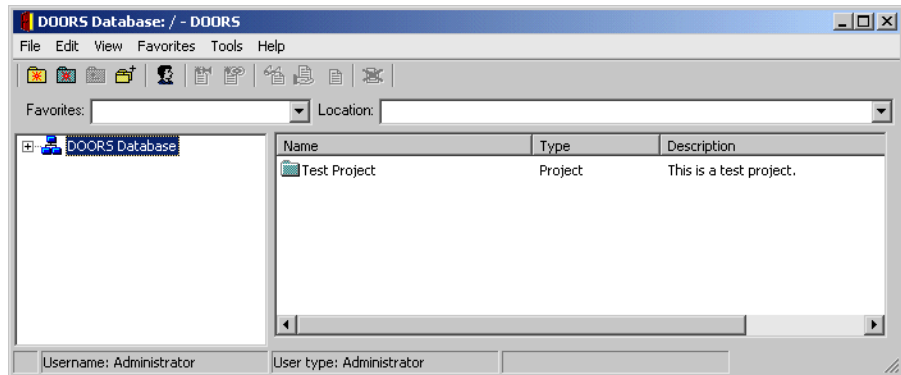
- 1 Right-click the DOORS Database node in the left pane and, from the resulting menu, select **New > Project**.

The New Project dialog box appears.



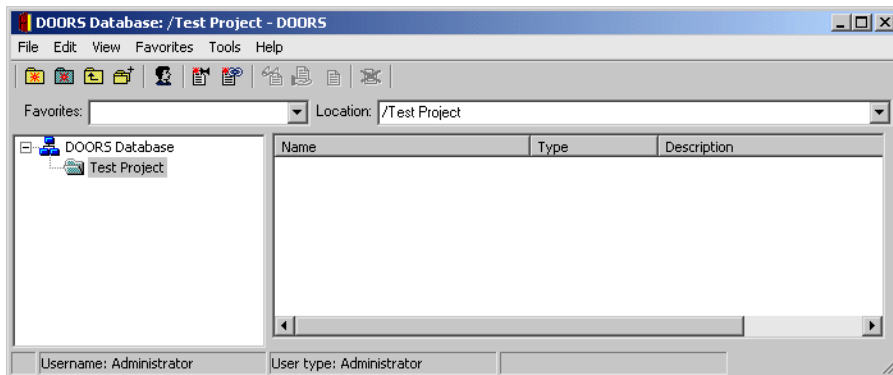
- 2 Enter the name Test Project and the description This is a test project. and click **OK**.

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



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

### 3 Managing Model Requirements with DOORS® Software





## Linking Objects to DOORS Requirements

### In this section...

“About Linkages Between a Simulink Model and DOORS Software” on page 3-9

“Creating a DOORS Requirement Object” on page 3-9

“Linking a Simulink or Stateflow Object to a DOORS Requirement” on page 3-11

### About Linkages Between a Simulink Model and DOORS Software

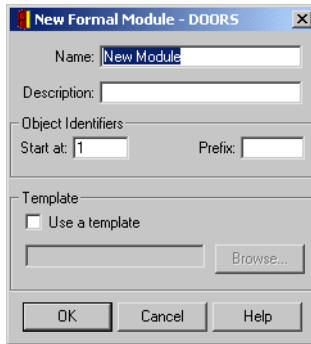
The Requirements Management Interface for Telelogic DOORS software lets you link from Simulink or Stateflow objects directly to DOORS requirements. This linking mechanism is similar to selection-based linking for Microsoft Word and Excel documents, described in “Selection-Based Linking” on page 2-22. That is, it provides two-way links by creating a special navigation object in the DOORS software, which allows you to navigate from the DOORS requirement to the associated object in the Simulink or Stateflow diagram. The sections that follow describe how to link DOORS requirements to objects in your Simulink or Stateflow diagram.

### Creating a DOORS Requirement Object

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

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

The New Formal Module dialog box appears.



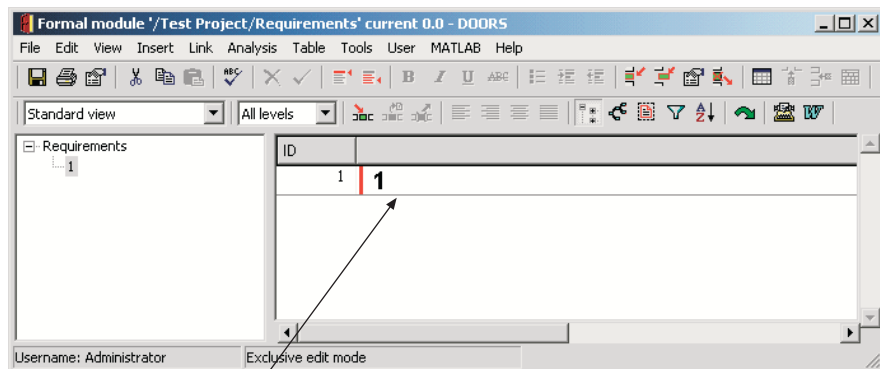
- 2 In the **Name** text field, enter the name **Requirements** and click **OK**.

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

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

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

A new object appears in the formal module.

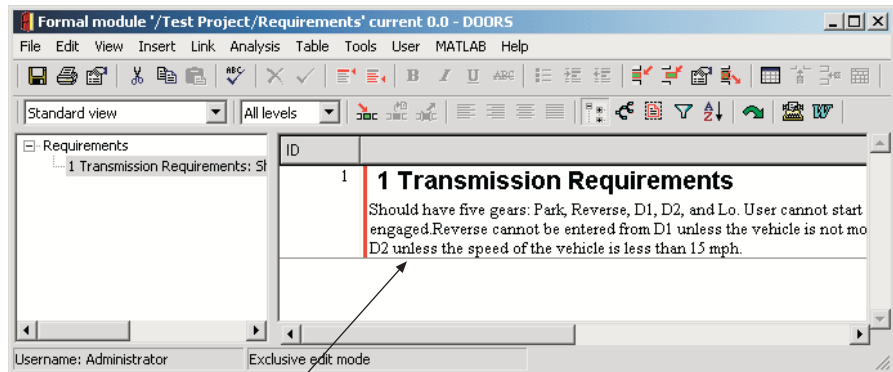


New object in Requirements module

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

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

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

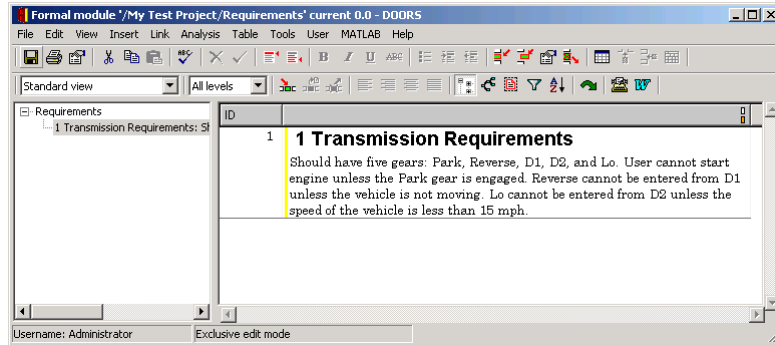


Specified object in Requirements module

## Linking a Simulink or Stateflow Object to a DOORS Requirement

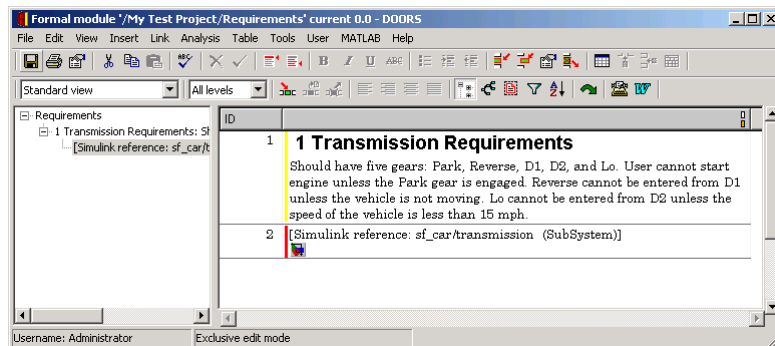
In “Creating a DOORS Requirement Object” on page 3-9, you created a Transmission Requirements object in the Requirements formal module in the DOORS software. Now use the following procedure to link the transmission block in the sf\_car model to this DOORS requirement.

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



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

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



- 4 Save the DOORS module.
- 5 Save the Simulink model as sf\_car\_doors.mdl.

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

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

## Synchronizing a DOORS Module with the Simulink Model

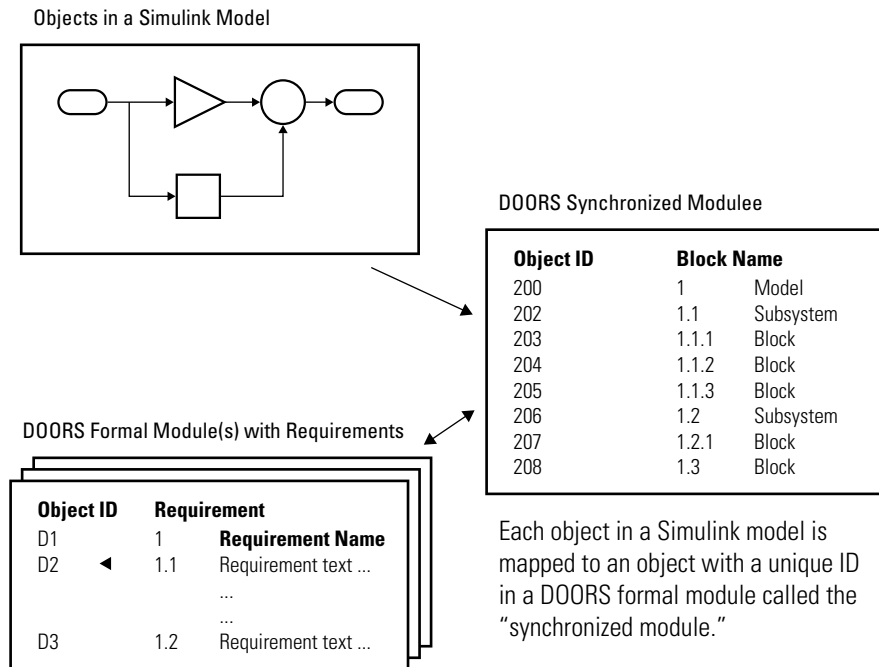
In this section...
“About Module Synchronization” on page 3-14
“Synchronizing a Model with the DOORS Software” on page 3-16
“Customizing the Level of Synchronization Detail” on page 3-17
“Customizing the DOORS Synchronization Settings” on page 3-22
“Linking Requirements to the DOORS Synchronized Module” on page 3-24

### About Module Synchronization

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

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

The following diagram illustrates the synchronization process.



Enter a requirement in a DOORS format module and link it to a uniquely mapped object in the synchronized module. Now you can navigate between the requirement and its Simulink object.

---

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

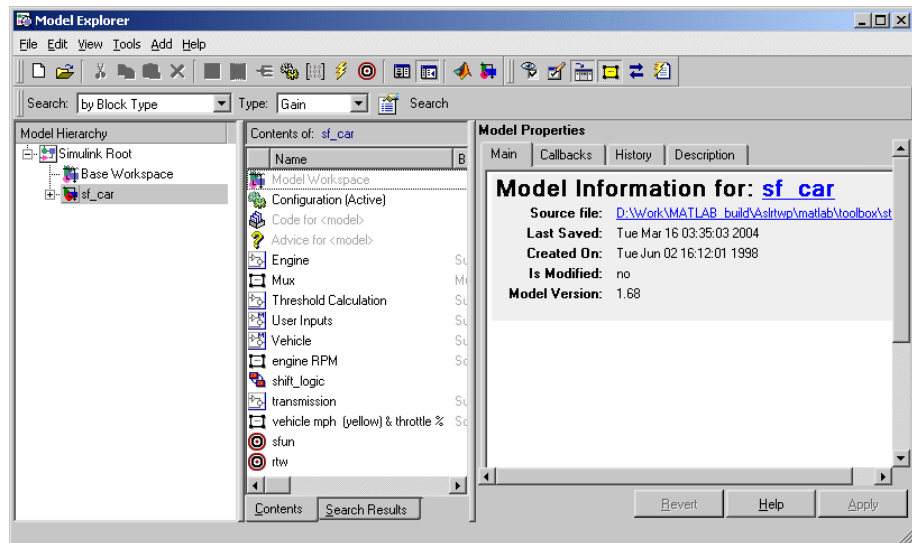
---


## Synchronizing a Model with the DOORS Software

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

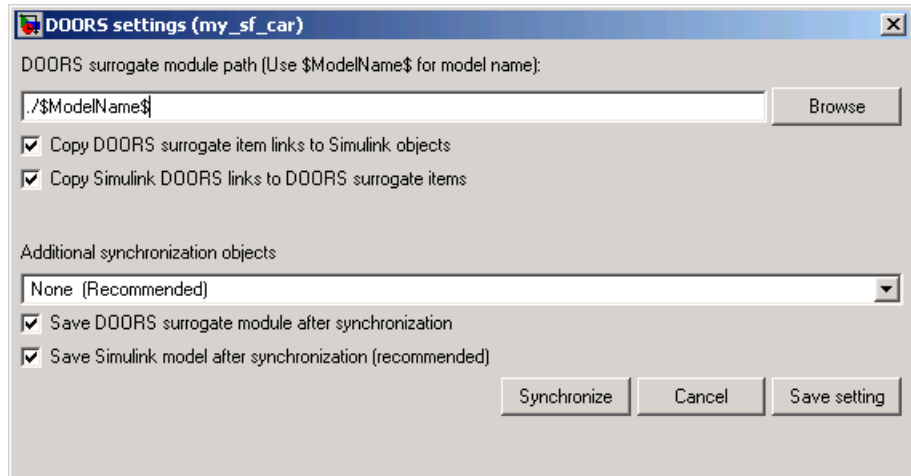
Use the following procedure to synchronize a Simulink model with the DOORS software:

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



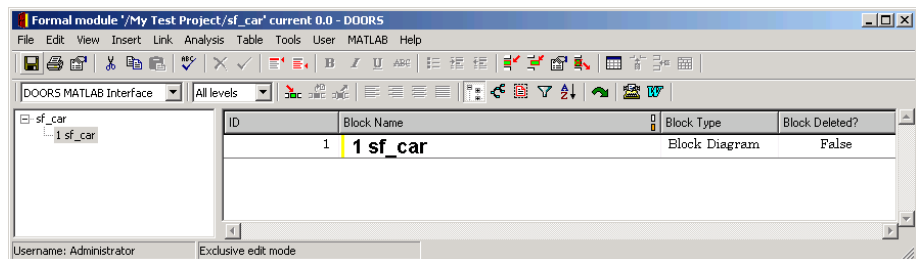
- 3 Select the Synchronize Requirements with DOORS tool  in the Model Explorer window. The DOORS settings dialog box opens.





#### 4 Click **Synchronize**.

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



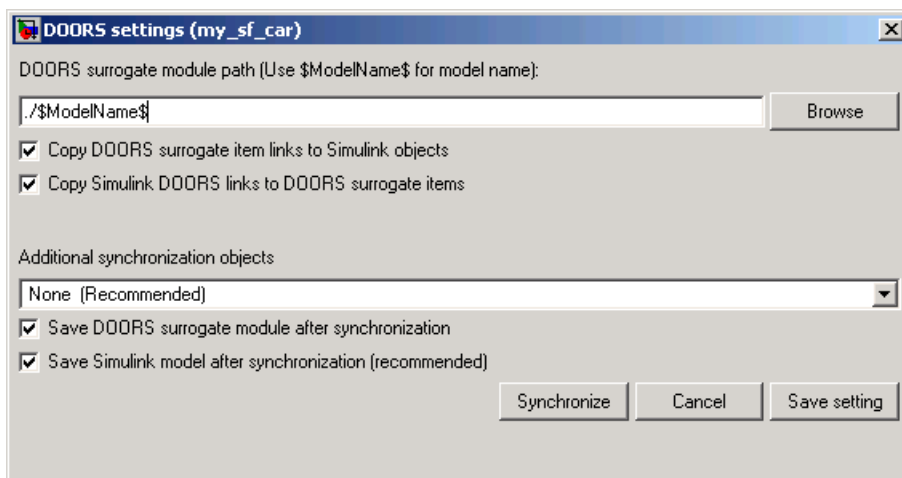
Notice that by default the DOORS formal module contains only one synchronized object, which corresponds to the top-level diagram. To include all the model blocks in the DOORS formal module, use the following procedure, “Customizing the Level of Synchronization Detail” on page 3-17.


## Customizing the Level of Synchronization Detail

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

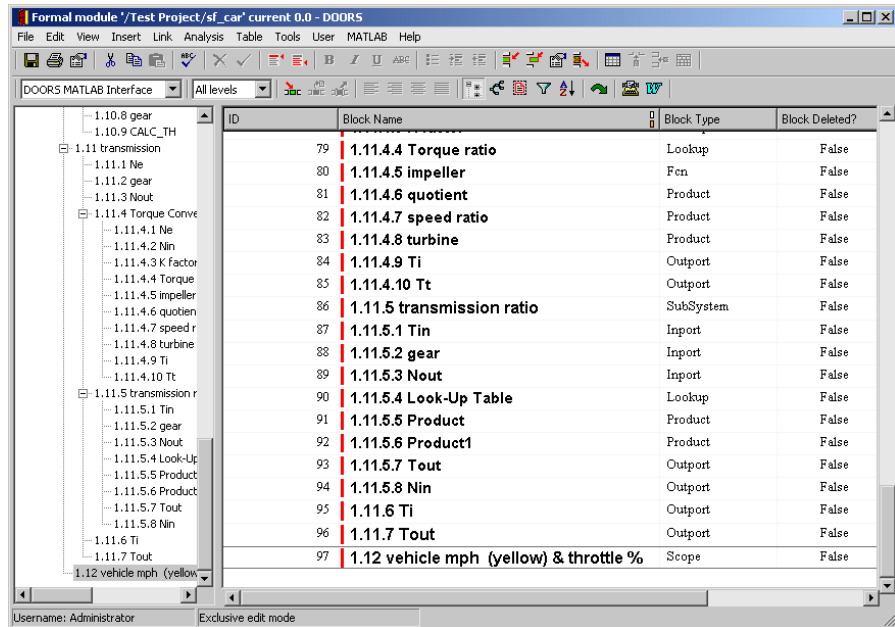
To include all the model blocks in the DOORS formal module, use the following procedure.

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




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

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

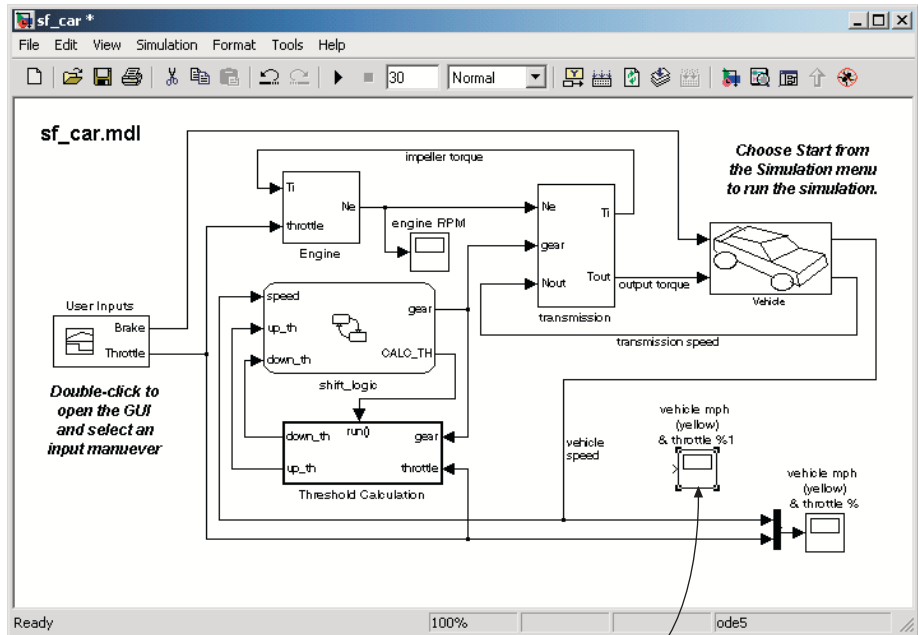



Notice the following:

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

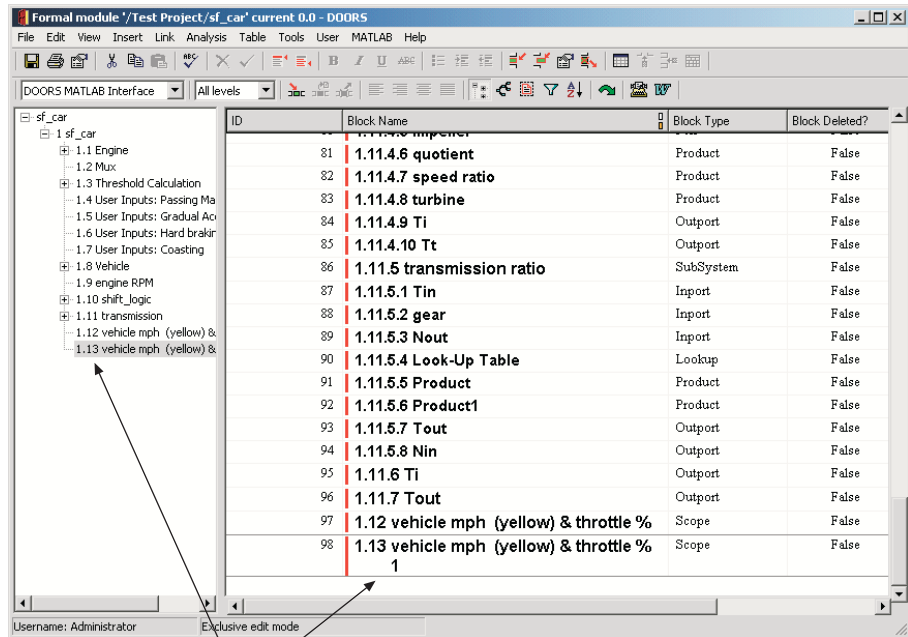
- You can add additional information columns to the right pane with the Insert Column tool  in the DOORS toolbar.

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



6 Select the Synchronize Requirements with DOORS tool  again.

The synchronized module is updated with the new block.



New block and link

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

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

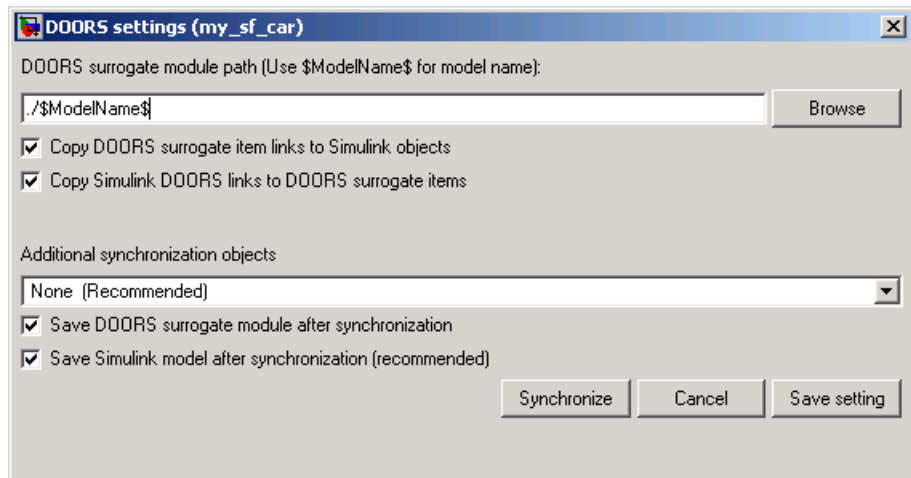
The deleted block appears at the bottom of the list of objects in the formal module and its entry in the **Block Deleted** column is True. If you want, you can delete this entry by right-clicking the line and selecting **Delete**. Otherwise, the module records the former presence of the deleted block.


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

## Customizing the DOORS Synchronization Settings

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

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

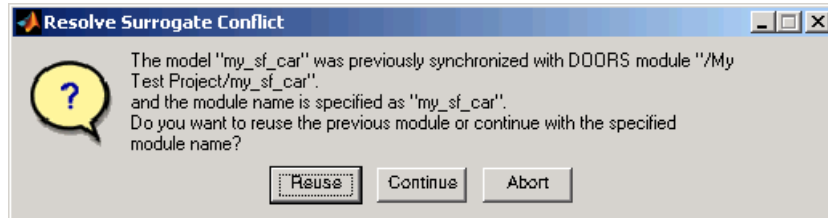


Another way to access this dialog box is to select the Synchronize Requirements with DOORS tool  in the Model Explorer window.

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

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

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



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

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

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

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

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

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

### **Linking Requirements to the DOORS Synchronized Module**

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

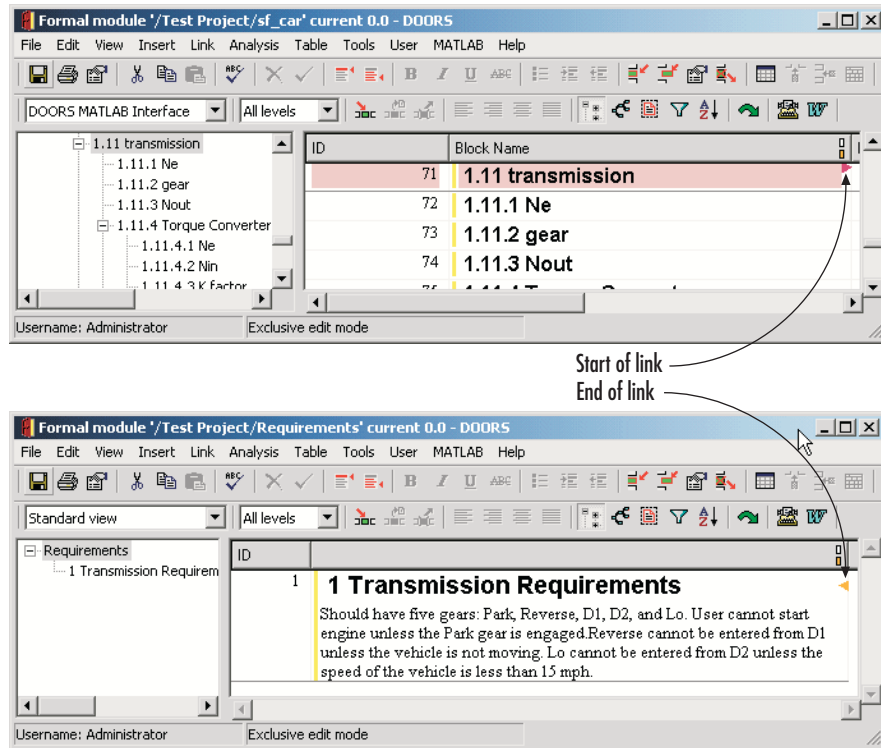
In “Creating a DOORS Requirement Object” on page 3-9, you created a Transmission Requirements object in the Requirements formal module in the DOORS software.

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

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

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





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

## Navigating Between Model Objects and DOORS Requirements

In this section...
“Viewing Model Elements with Requirements” on page 3-26
“Navigating from a Simulink Model to DOORS Requirements” on page 3-28
“Navigating from a DOORS Requirements to the Simulink Model” on page 3-30

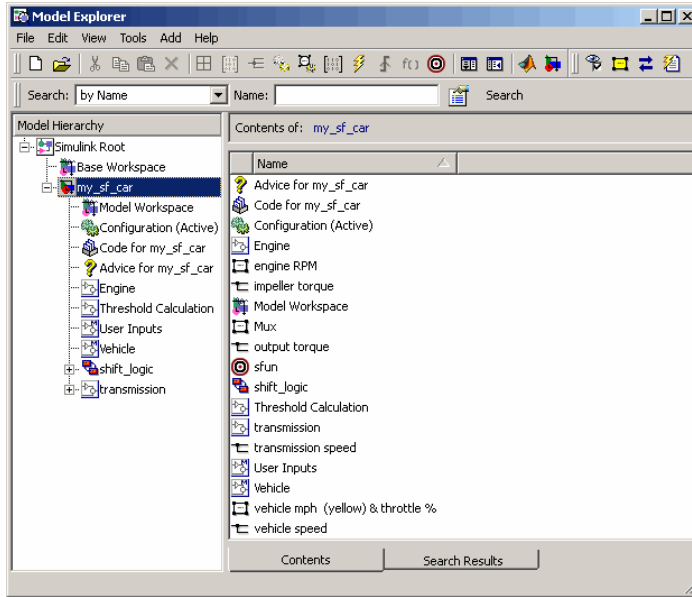
### Viewing Model Elements with Requirements

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

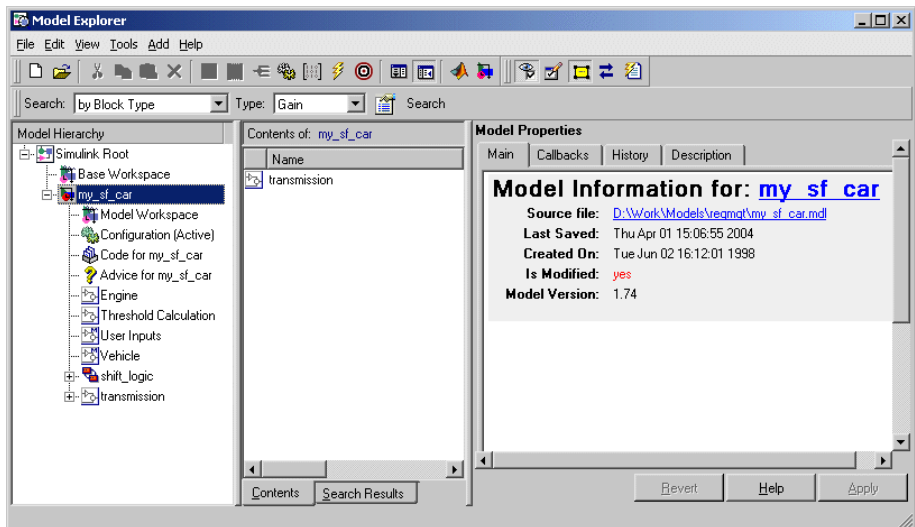
Use the following procedure to display only those model elements with requirements:

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

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



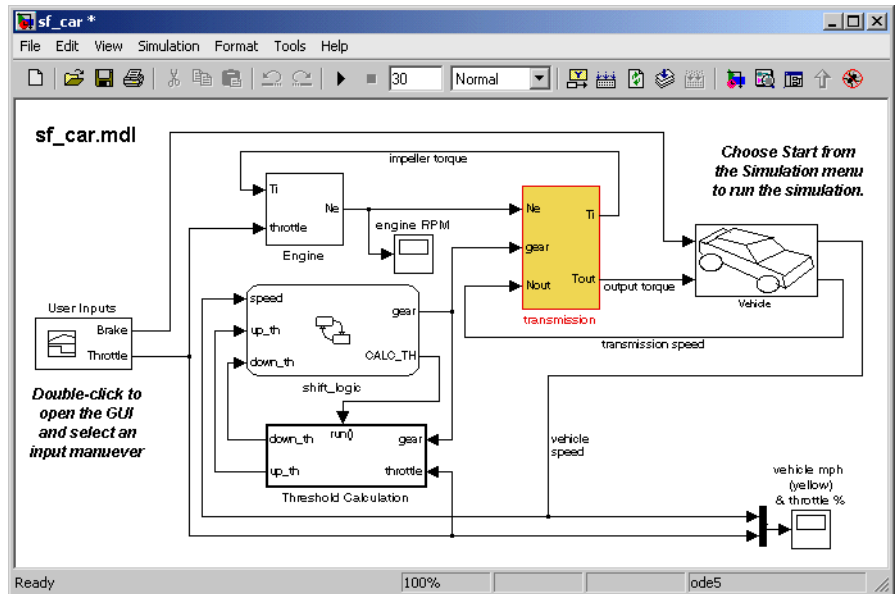
2 Select the Display Objects with Linked Requirements tool  in the Model Explorer toolbar.



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

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

The transmission block in the Simulink model is highlighted.

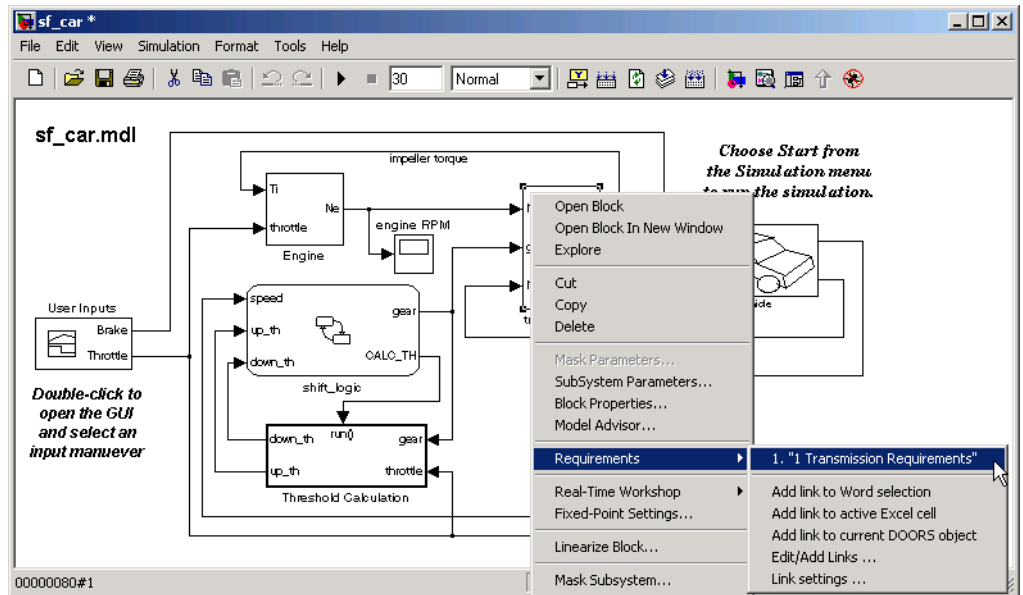


## Navigating from a Simulink Model to DOORS Requirements

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

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

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



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

## Navigating Through the Synchronized Module

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

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

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

A pop-up menu appears.

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

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

- 3 Access individual requirements in the synchronized module.

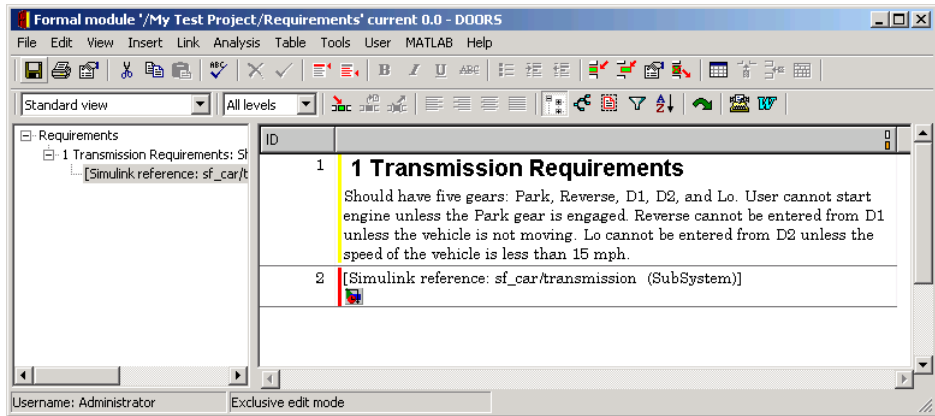
You can access individual requirements by right-clicking the arrows that appear in the **Block Name** column for each mapped object with requirements and making a requirement selection.

### **Navigating from a DOORS Requirements to the Simulink Model**

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

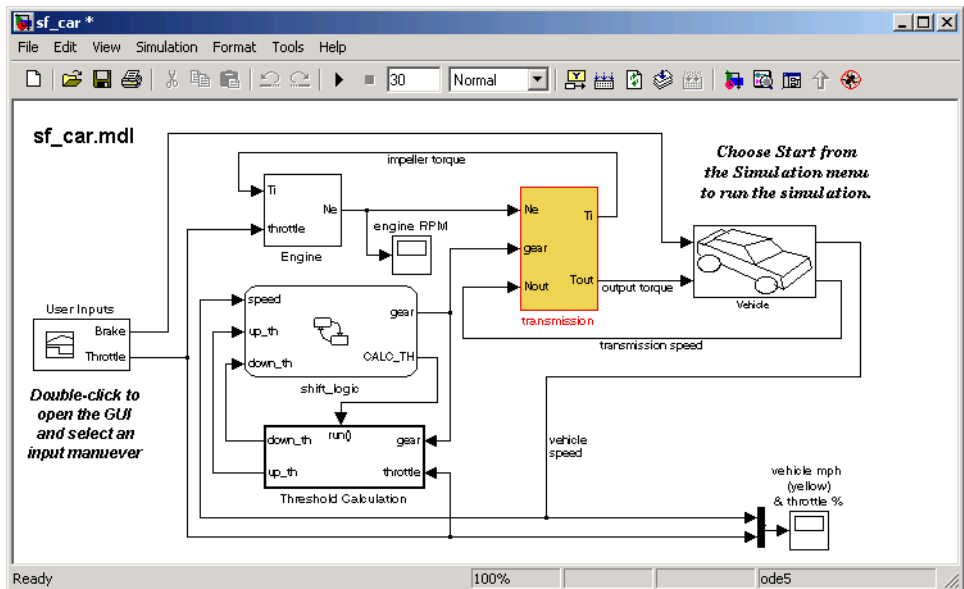
Use the following procedure to navigate from the Transmission Requirements DOORS object to the transmission block in the Simulink model:

- 1 In the DOORS software, open the formal module window **Requirements**.
- 2 Select the Simulink Reference sub-node of the Transmission Requirements node in the left pane.



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

The transmission block in the Simulink diagram is highlighted.



## Navigating Through the Synchronized Module

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

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

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

The object opens in its native diagram as follows:

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

---

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

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

---



# 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

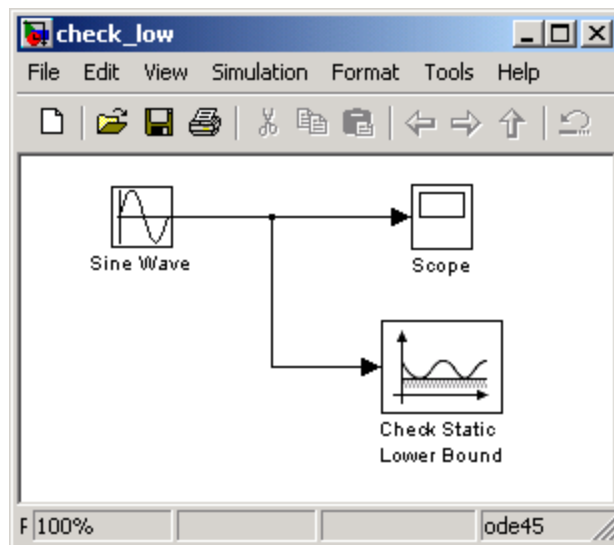
You use Model Verification blocks throughout your model to monitor its signals. You can set a verification block to assert when its signal leaves the specified limit or range. During simulation, when the signal crosses the limit, the verification block can

- 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

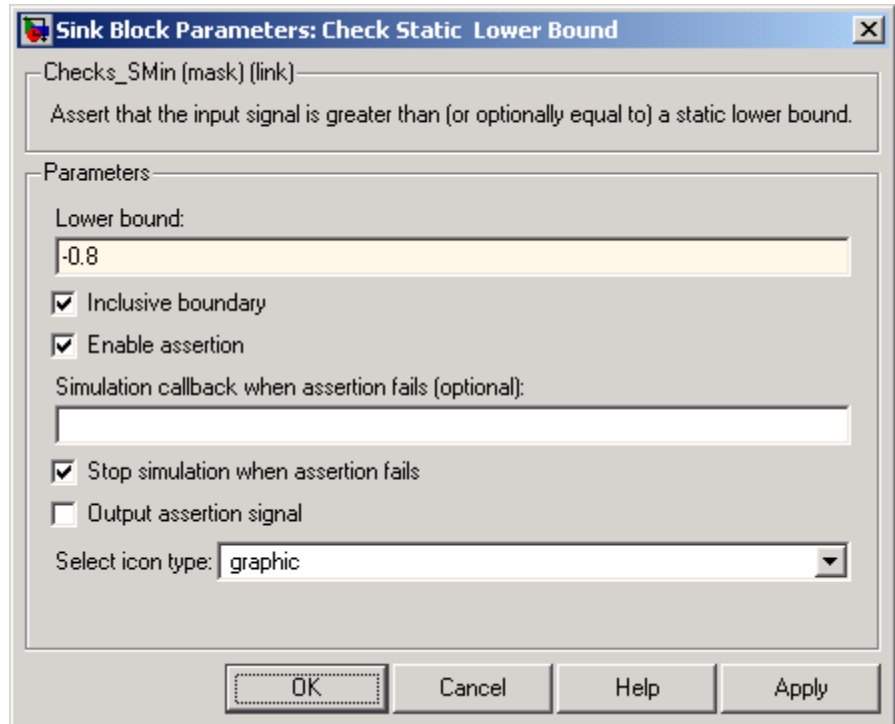
To see a complete list of all Model Verification blocks and references for each, see the “Model Verification” category in the Simulink Block Reference documentation.

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.



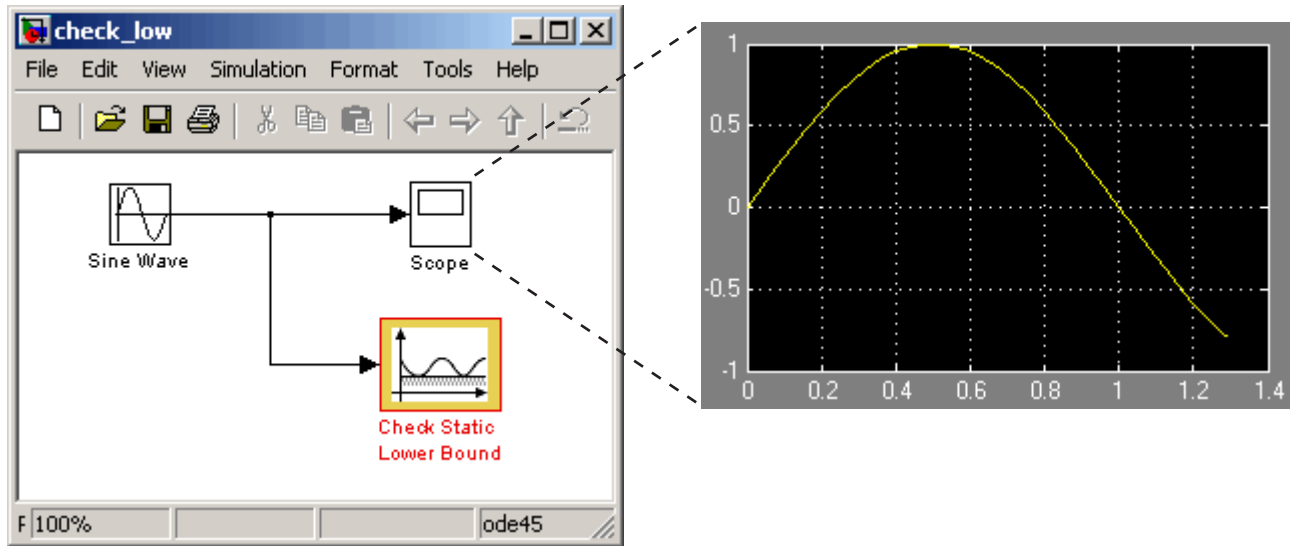
- 2 Set the model to run for 2 seconds while the Sine Wave block outputs a signal with an amplitude of 1 and a frequency of  $\pi$  radians per second.
- 3 Open the Check Static Lower Bound block and set the parameters as follows:



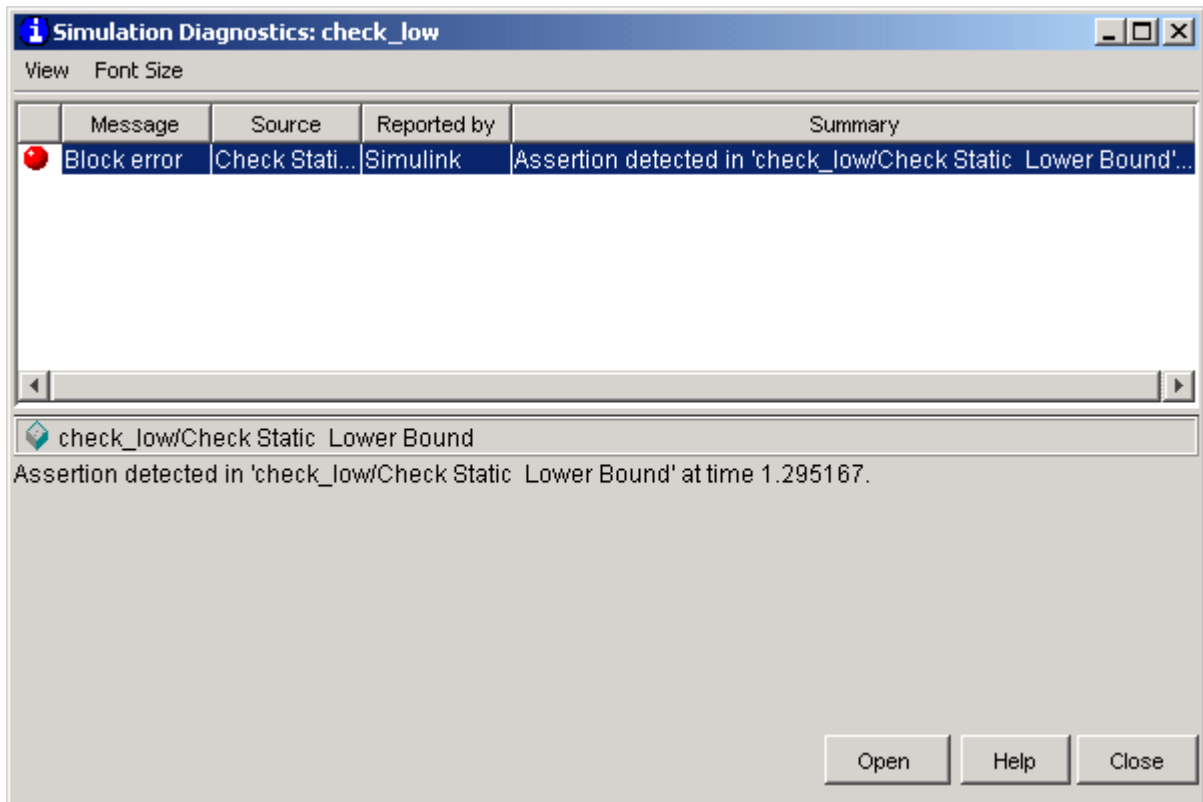
A verification block is enabled for an assertion when the **Enable assertion** check box is selected (this is the default setting). According to the preceding property settings, the Check Static Lower Bound block is set to detect a signal value of -0.8 or lower. If this signal is detected, simulation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

- 4 Run the simulation.

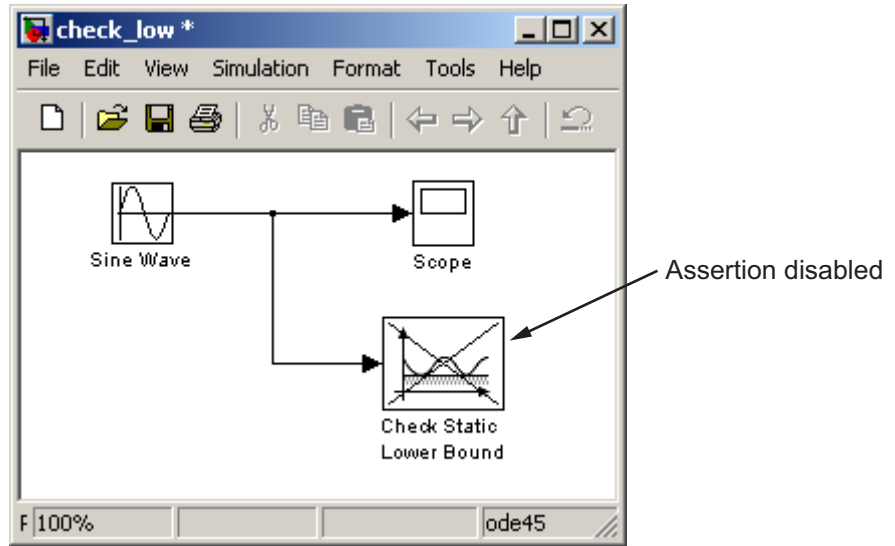
The model stops simulating after 1.295 seconds, when the output is -0.8, as shown. This brings focus to the asserting verification block, which is highlighted.



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



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



## 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

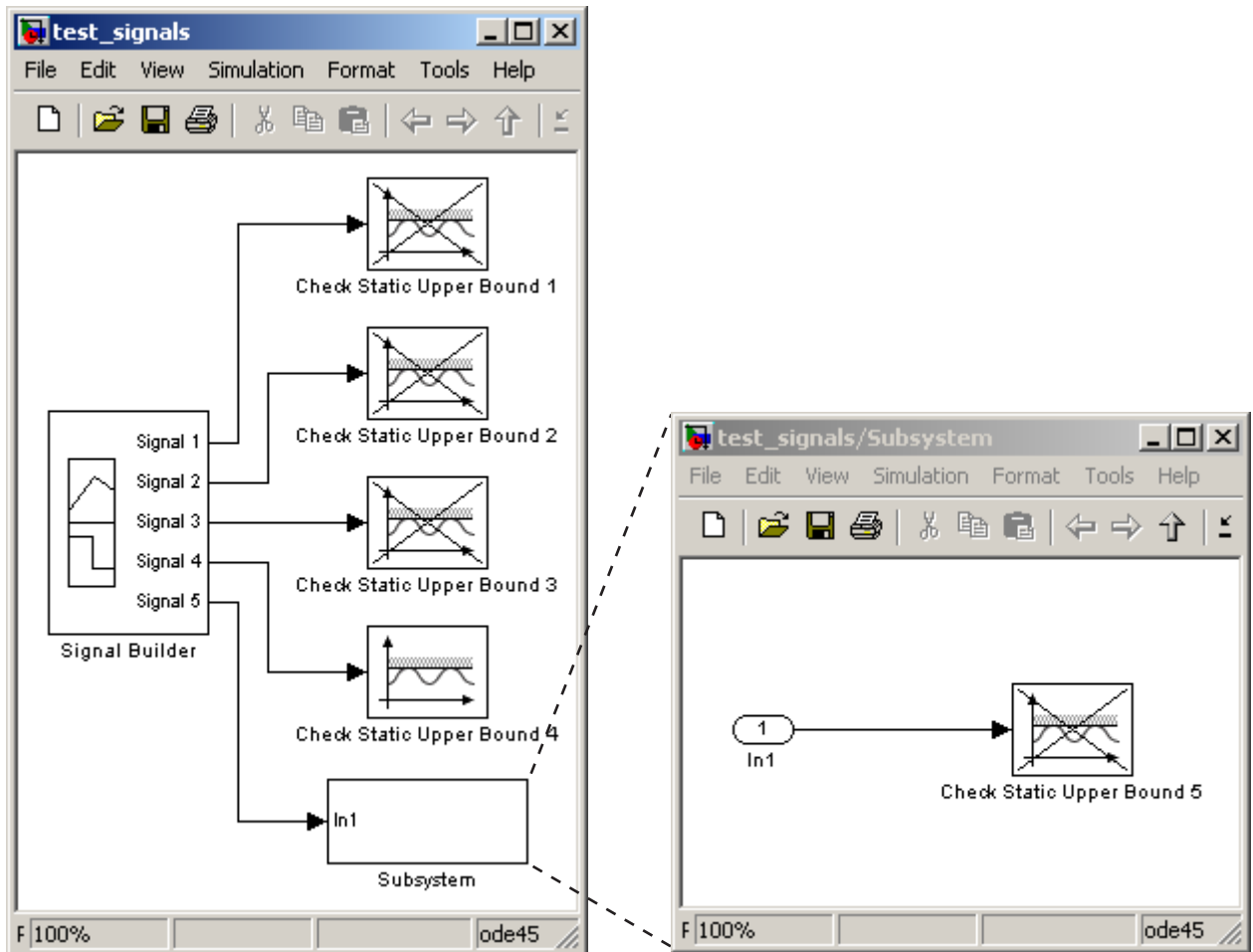
### What Is the Verification Manager?

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.



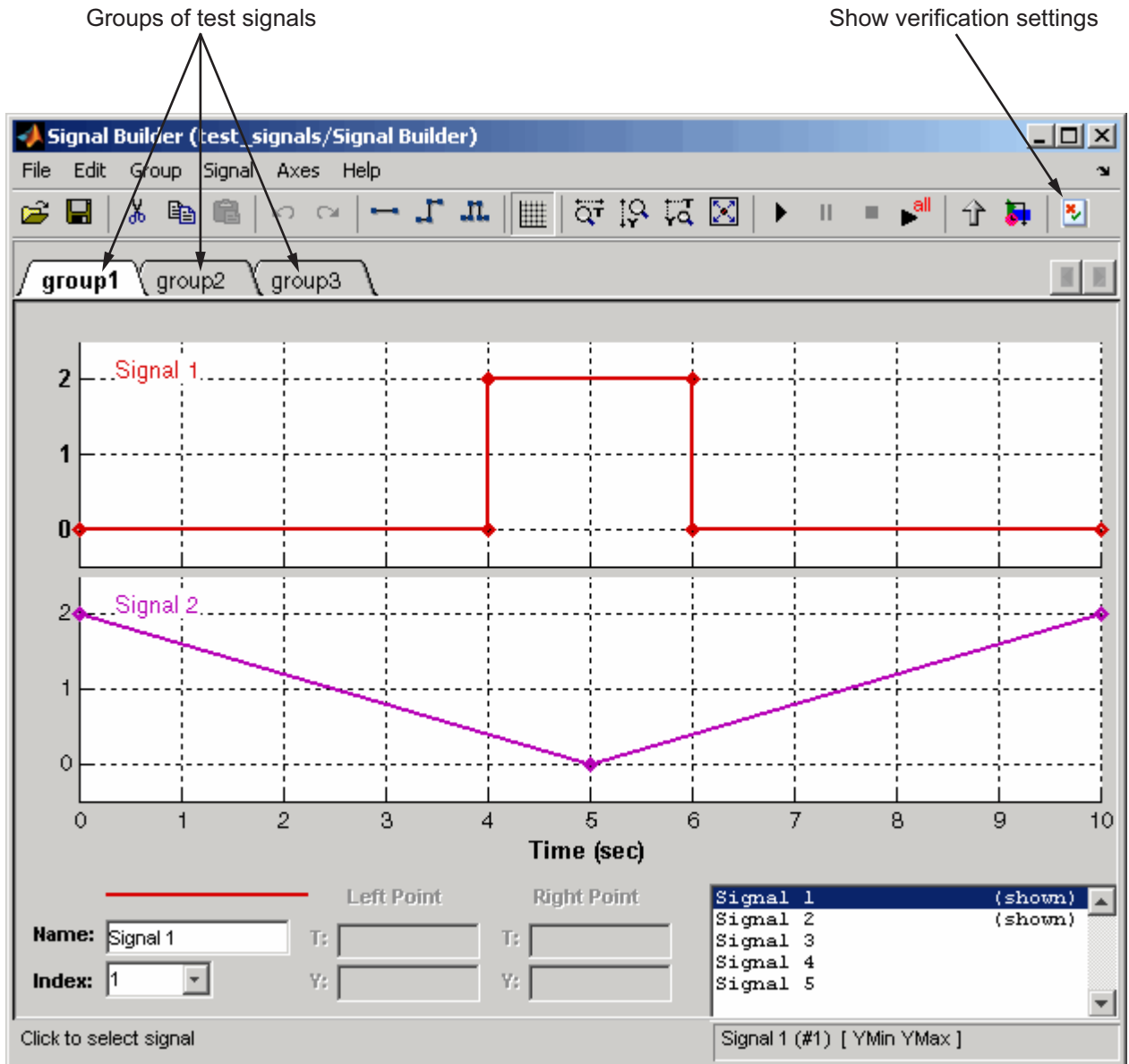
Typically, a Signal Builder block provides test signals for an entire model from one location. The example model contains a Signal Builder block feeding five test signals to Model Verification blocks. Signals 1 through 4 are sent directly to Check Static Upper Bound Model Verification blocks. The fifth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

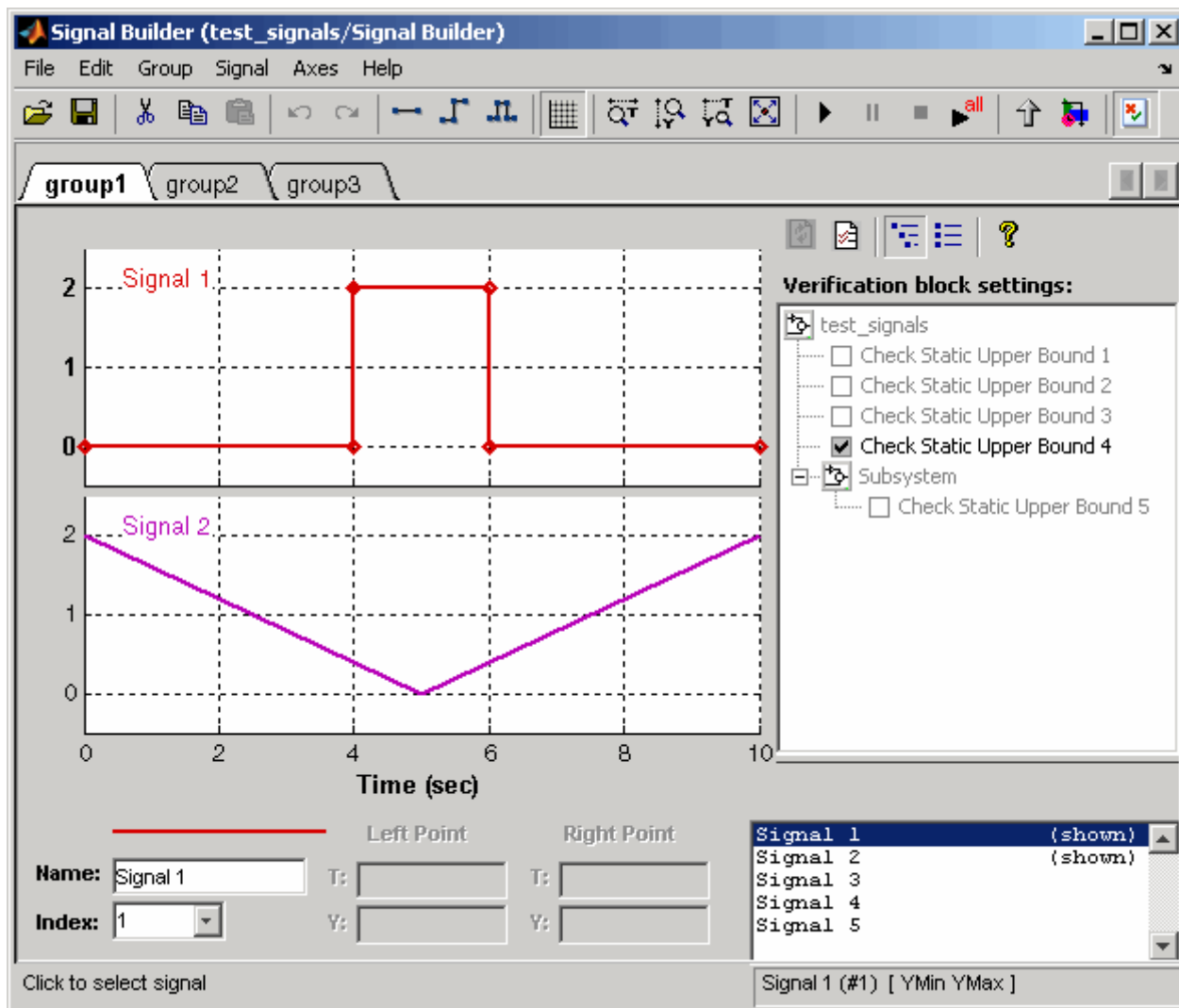
The **Verification block settings** pane and the **Requirements** pane appear as shown.

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


- Window Title:** Signal Builder (test\_signals/Signal Builder)
- Menu Bar:** File, Edit, Group, Signal, Axes, Help
- Toolbar:** Includes icons for file operations, navigation, and execution.
- Group Tabs:** group1, group2, group3
- Signal Waveforms:**
  - Signal 1 (Red):** A step function that is 0 from 0 to 4 seconds, jumps to 2 at 4 seconds, and returns to 0 at 6 seconds.
  - Signal 2 (Purple):** A triangular wave starting at 2 at 0 seconds, reaching 0 at 5 seconds, and returning to 2 at 10 seconds.
- Verification pane (right side):**
  - Verification block settings:**
    - test\_signals
      - Check Static Upper Bound 1
      - Check Static Upper Bound 2
      - Check Static Upper Bound 3
      - Check Static Upper Bound 4
      - Subsystem
        - Check Static Upper Bound 5
  - Requirements:** No requirements in this group
- Bottom Panel:**
  - Fields for Name, Index, Left Point (T, Y), and Right Point (T, Y).
  - A list of signals: Signal 1 (shown), Signal 2 (shown), Signal 3, Signal 4, Signal 5.
  - Text: "Click to select signal"

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

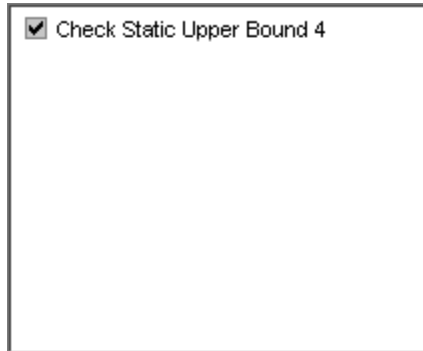
- 4 Just above the **Verification block settings** pane, select  to close the **Requirements** pane.




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

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

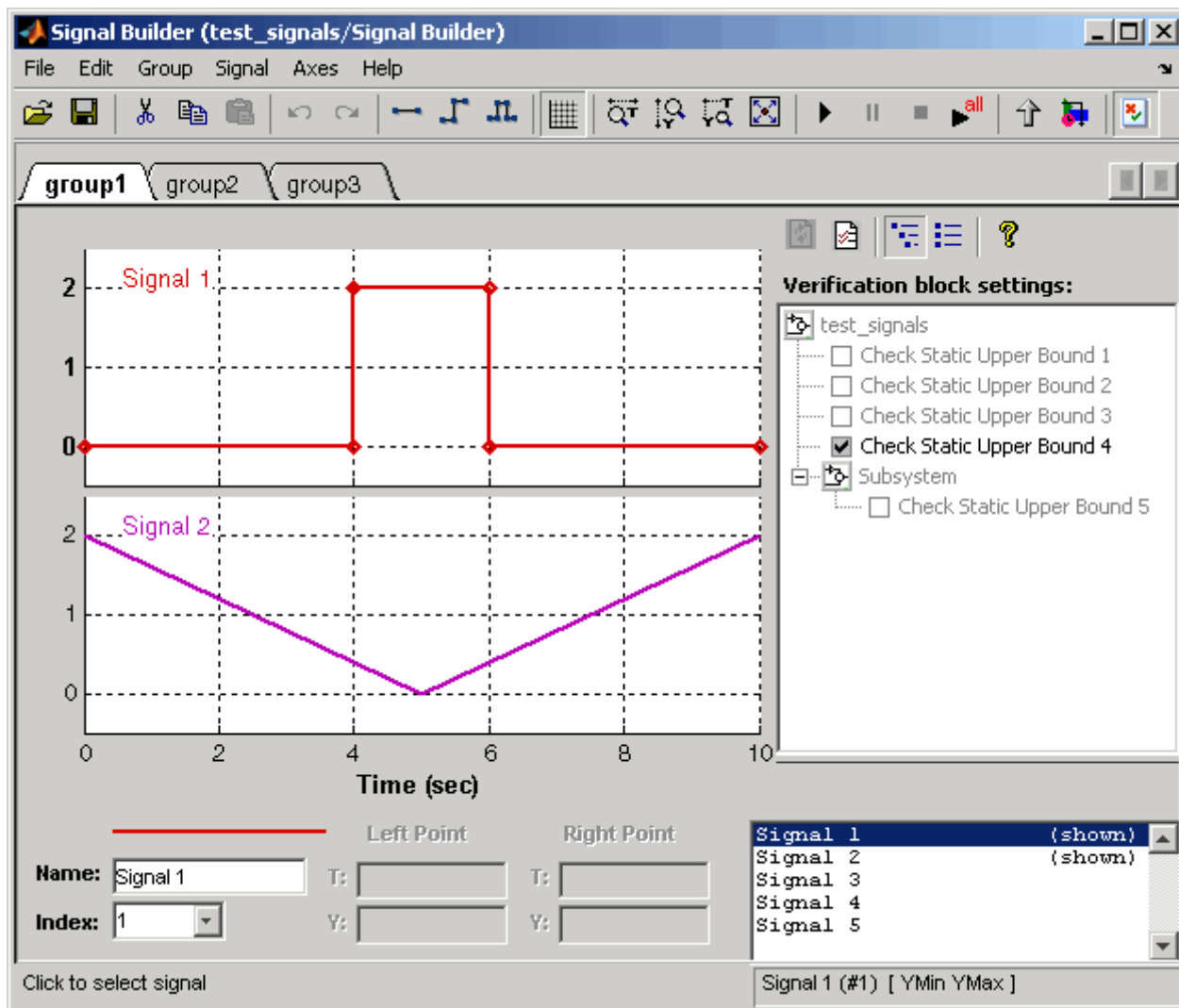
The **Verification block settings** pane now shows only the enabled Model Verification blocks for the current group, as shown.



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

## Enabling and Disabling Model Verification Blocks with the Verification Manager

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



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

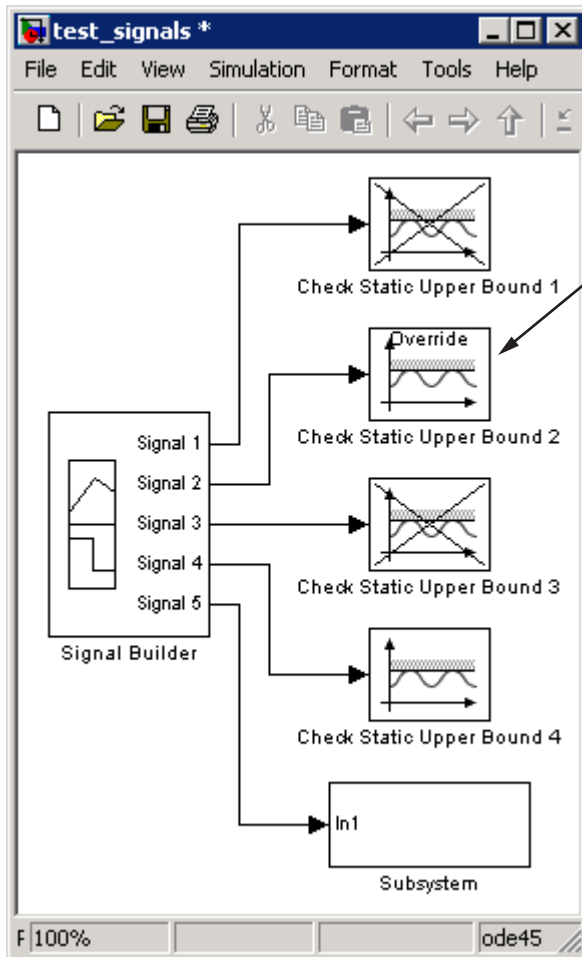


Node	Status
<input type="checkbox"/>	Verification block is disabled for this group. Click to enable for current group.
<input checked="" type="checkbox"/>	Verification block is enabled for the current group. Click to disable for current group.
<input checked="" type="checkbox"/>	Verification block is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

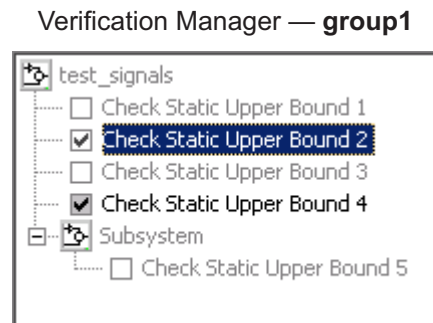
Use the Verification Manager to enable or disable model verification blocks in the `test_signals` model you created in “Opening the Verification Manager” on page 4-7, as follows:

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

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

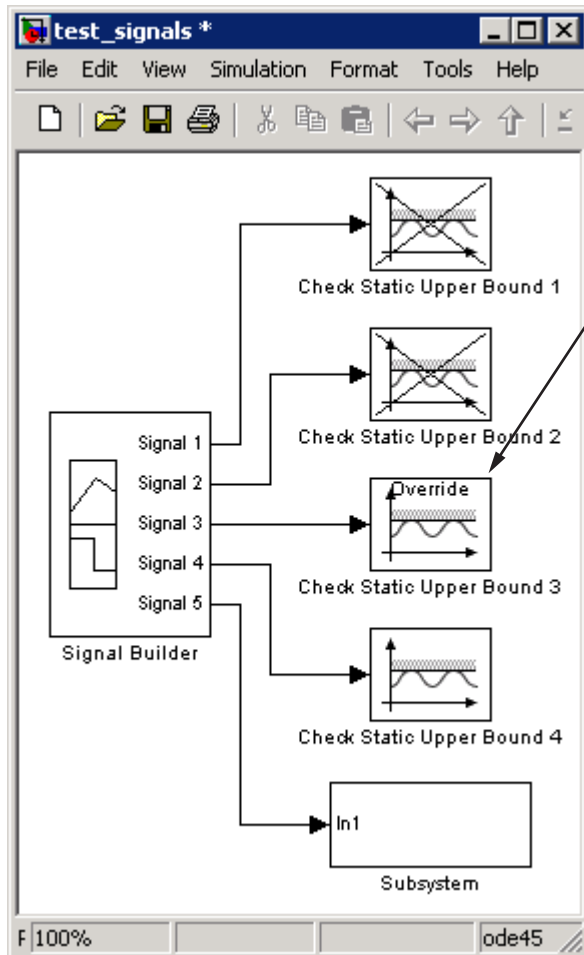


Disabled but enabled in current group (**group1**)



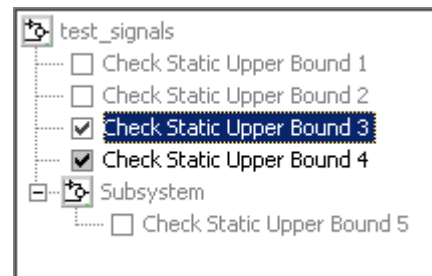
Because it is enabled in the current group, the Check Static Upper Bound 2 block gains an **Override** label and loses its cross-out. The meaning behind the change in appearance becomes clearer when another group is selected.

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






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

#### Verification Manager — **group2**




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

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

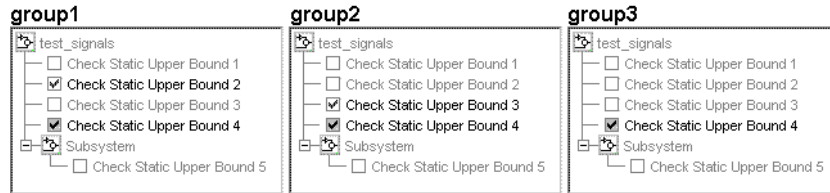
Assertion Block	Description
	Enabled for all groups
	Disabled in current group
	Enabled in current group

### Using Enabling and Disabling Tools in the Verification Manager

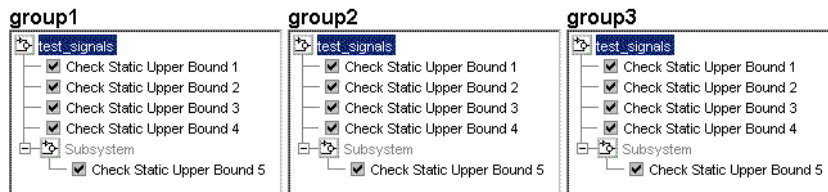
If you have many verification blocks, it is tedious to enable and disable blocks individually. For this reason, the Verification Manager lets you enable and disable blocks through selections from a context menu. These selections vary with the node as follows:

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

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

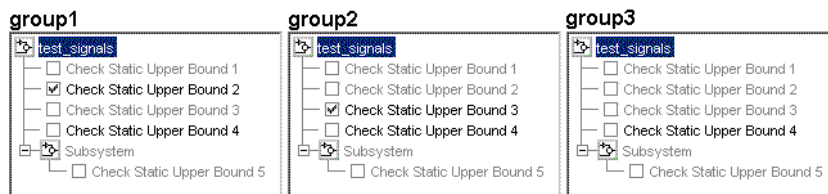


- 1 Right-click the `test_signals` node and select **Contents enable for all groups**.



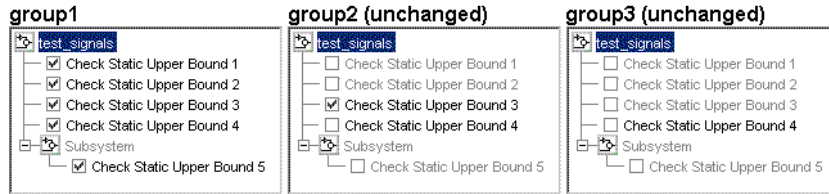
Applying the **Contents enable for all groups** selection to the model node enables all contained Model Verification blocks, for all test groups, in all contained subsystems.

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



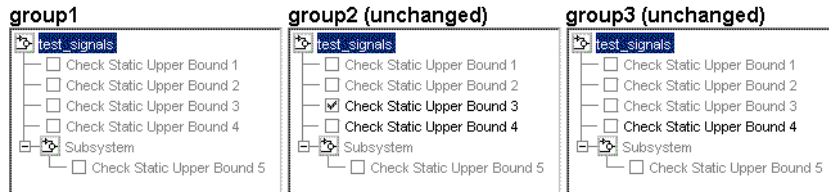
Applying the **Contents enable by group** selection to the model node restores the previous individually enabled/disabled settings for each block in each group.

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



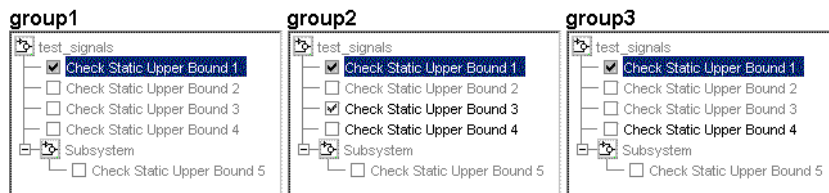
Applying **Contents group enable** to the `test_signals` model node in **group1** individually enables all contained blocks for **group1**, but leaves the other groups untouched.

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



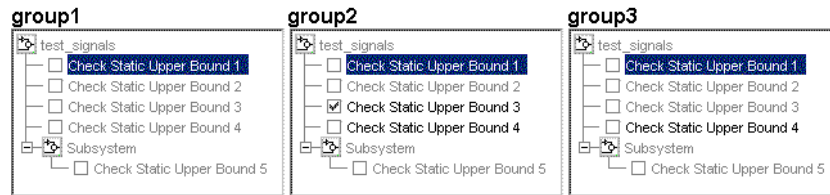
Applying **Contents group disable** to the `test_signals` model node in **group1** individually disables all contained blocks for **group1**, but leaves the other groups untouched.

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



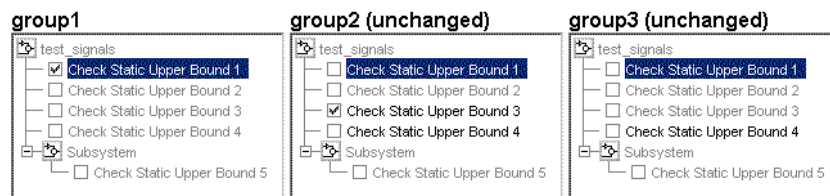
Applying **Block enable for all groups** to the individual **group1** block node for `Check Static Upper Bound 1` in **group1** enables this block for all groups.

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



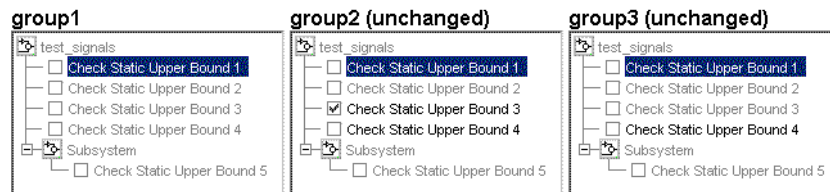
Applying **Block enable by group** to the individual **group1** block node for Check Static Upper Bound 1 in **group1** restores the previous individually enabled/disabled state to this block for all groups. This lets you enable or disable this node individually for each group.

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



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

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



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

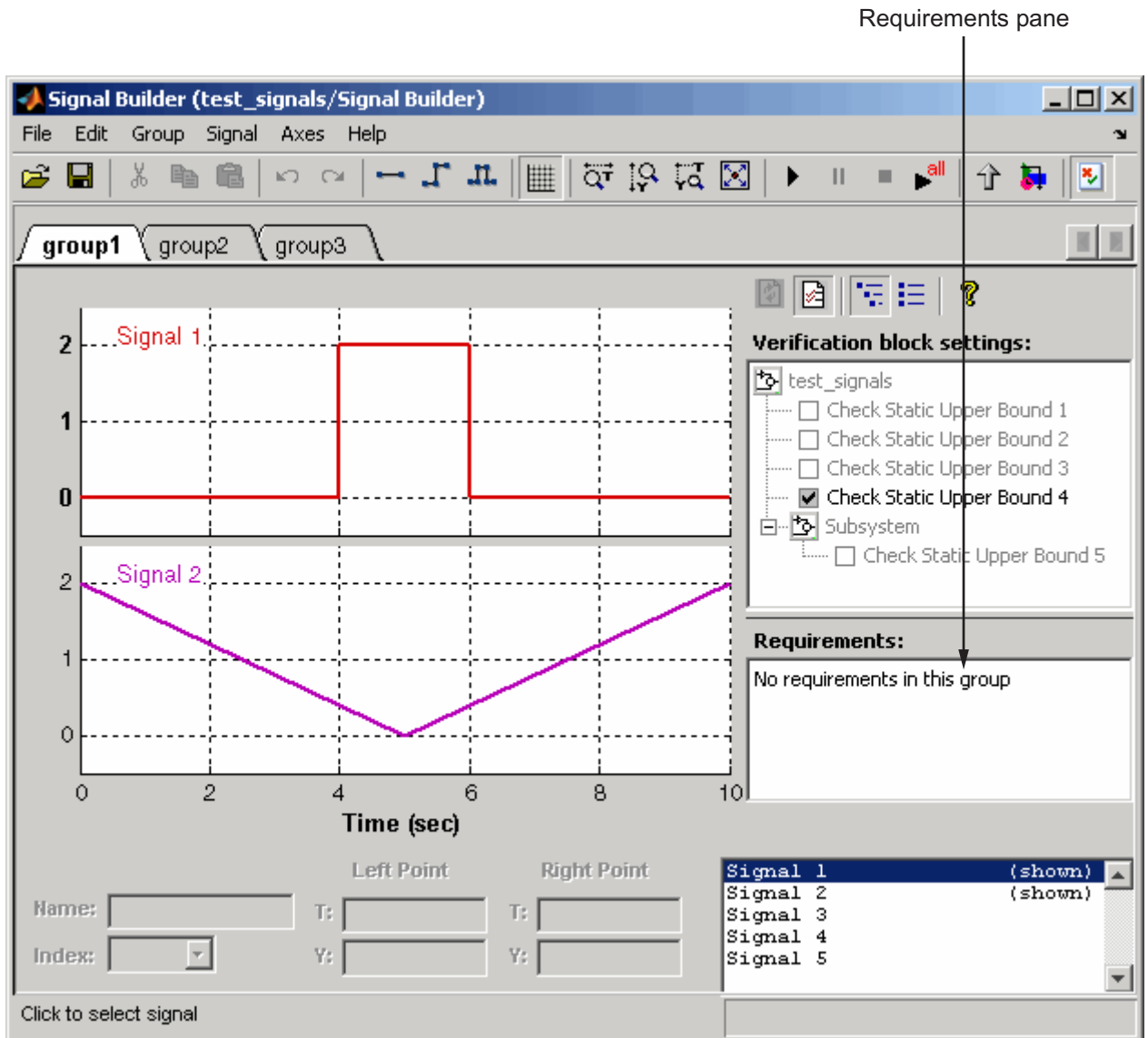
# Managing Verification Requirements

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

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

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



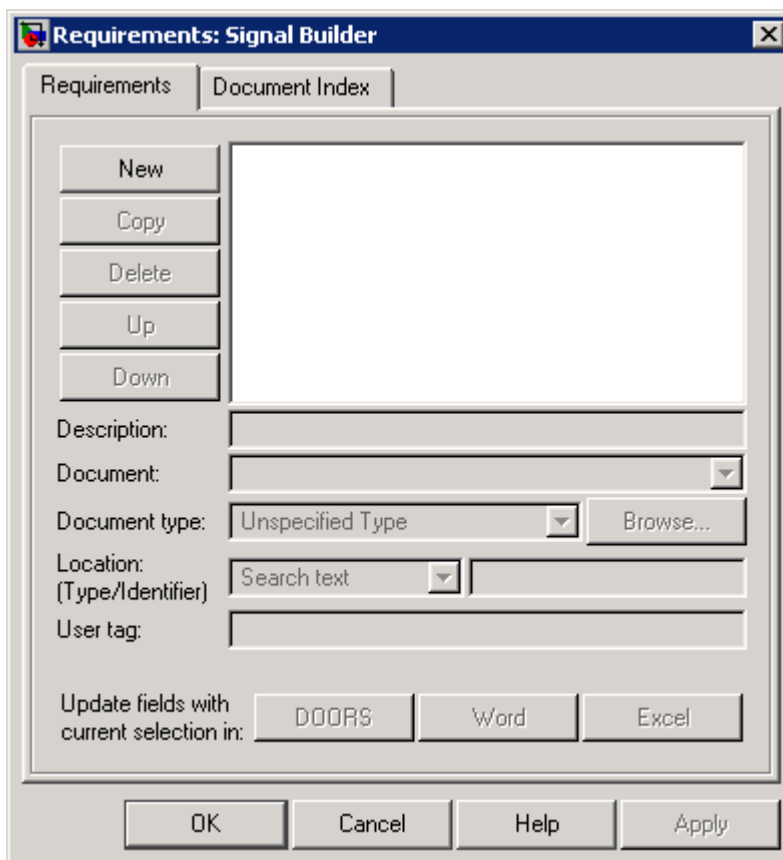


1 Right-click anywhere in the **Requirements** pane.

A pop-up menu appears.

- 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 “Adding Requirement Links to an Object” on page 2-7.

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

New requirements

The screenshot displays the Signal Builder application window. The main area contains two vertically stacked plots. The top plot, labeled 'Signal 1', shows a red square wave pulse between 4 and 6 seconds with a value of 2. The bottom plot, labeled 'Signal 2', shows a purple V-shaped signal starting at 2, reaching 0 at 5 seconds, and returning to 2 at 10 seconds. The x-axis for both plots is 'Time (sec)' from 0 to 10. To the right of the plots is a 'Verification block settings' panel with a tree view containing 'test\_signals' and 'Subsystem'. Under 'test\_signals', there are five 'Check Static Upper Bound' options, with 'Check Static Upper Bound 4' selected. Below this is a 'Requirements:' panel listing 'Requirement 1' and 'Requirement 2'. An arrow labeled 'New requirements' points from the text above to 'Requirement 2'. At the bottom right, a signal list shows 'Signal 1' through 'Signal 5', with 'Signal 1' and 'Signal 2' marked as '(shown)'. At the bottom left, there are input fields for 'Name:', 'Index:', 'Left Point' (T, Y), and 'Right Point' (T, Y).

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

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

# Using Model Coverage

---

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

- “Introduction to Model Coverage” on page 5-2
- “Using Model Coverage” on page 5-7
- “Specifying Model Coverage Reporting Options” on page 5-11
- “Understanding Model Coverage Reports” on page 5-25
- “N-Dimensional Lookup Table Report” on page 5-36
- “Signal Range Analysis Report” on page 5-43
- “Colored Simulink Diagram Coverage Display” on page 5-47
- “Using Model Coverage Commands” on page 5-52
- “Using Model Coverage Commands for Referenced Models” on page 5-59
- “Model Coverage for Embedded MATLAB Function Blocks” on page 5-64

## Introduction to Model Coverage

In this section...
“What Is Model Coverage?” on page 5-2
“How Model Coverage Works” on page 5-2
“Types of Model Coverage” on page 5-2
“Blocks That Receive Model Coverage” on page 5-4

### What Is Model Coverage?

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

### How Model Coverage Works

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

See “Understanding Model Coverage Reports” on page 5-25 for an example of a model coverage report along with descriptions of the coverages it contains. Before you do, you might need to review the types of coverages that model coverage performs in “Types of Model Coverage” on page 5-2.

### Types of Model Coverage

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

- “Cyclomatic Complexity” on page 5-3

- “Decision Coverage (DC)” on page 5-3
- “Condition Coverage (CC)” on page 5-3
- “Modified Condition/Decision Coverage (MC/DC)” on page 5-4
- “Lookup Table Coverage (LUT)” on page 5-4

## Cyclomatic Complexity

Cyclomatic complexity is a measure of the structural complexity of a model. It approximates the McCabe complexity measure for code generated from the model. In general, the McCabe complexity measure is slightly higher because of error checks that the model coverage analysis does not consider.

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

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

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

## Decision Coverage (DC)

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

## Condition Coverage (CC)

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

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

### **Modified Condition/Decision Coverage (MC/DC)**

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

### **Lookup Table Coverage (LUT)**

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

---

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

---

### **Blocks That Receive Model Coverage**

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

<b>Block</b>	<b>Decision</b>	<b>Condition</b>	<b>MC/DC</b>	<b>LUT</b>
1D Lookup				•



<b>Block</b>	<b>Decision</b>	<b>Condition</b>	<b>MC/DC</b>	<b>LUT</b>
2D Lookup				•
ND Lookup				•
Interpolation Using Prelookup				•
ND Direct Lookup				•
Abs	•			
Combin. Logic	•	•		
Discrete-Time Integrator (when saturation limits are enabled)	•			
Embedded MATLAB™ Function	•	•	•	
Fcn (Boolean operators only)		•		
For	•			
If	•			
Logic		•	•	
MinMax	•			
Model	•	•	•	•
Multiport Switch	•			
Rate Limiter	•  (Relative to slew rates)			
Relay	•			
Saturation	•			
Stateflow (see note below)	•	•	•	
Subsystem	•	•	•	
Switch	•			

<b>Block</b>	<b>Decision</b>	<b>Condition</b>	<b>MC/DC</b>	<b>LUT</b>
SwitchCase	•			
While	•			

---

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

---

# Using Model Coverage

**In this section...**

“Basic Workflow for Using Model Coverage” on page 5-7

“Creating and Running Test Cases” on page 5-7

## Basic Workflow for Using Model Coverage

To develop effective tests with model coverage:

- 1** Develop one or more test cases for your model. (See “Creating and Running Test Cases” on page 5-7.)
- 2** Run the test cases to verify that the model behavior is correct.
- 3** Analyze the coverage reports produced by the Simulink Verification and Validation software.
- 4** Using the information in the coverage reports, modify the test cases to increase their coverage or add new test cases that cover areas not covered by the current set of test cases.
- 5** Repeat the preceding steps until you are satisfied with the coverage of your test set.

---

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

---

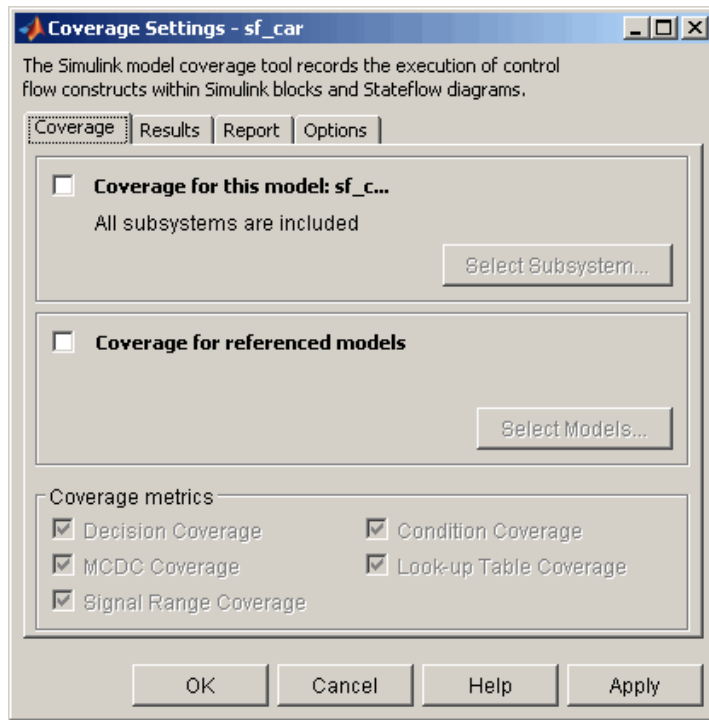
## Creating and Running Test Cases

Model coverage provides two MATLAB commands, `cvtest` and `cvsim`, for creating and running test cases. The `cvtest` command creates test cases that the `cvsim` command runs. (See “Creating Tests with `cvtest`” on page 5-52 and “Running Tests with `cvsim`” on page 5-54.)

You can also run the coverage tool interactively as follows:

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

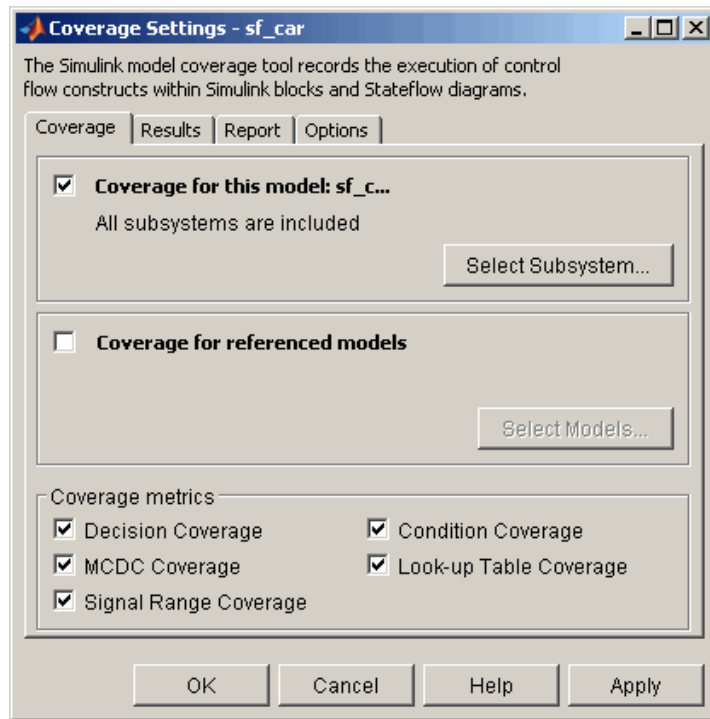
The Coverage Settings dialog box appears.



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

- 2 Select **Coverage for this model**.

Selecting this option enables the **Select Subsystem** button and the check boxes of the **Coverage metrics** section.



Selecting **Coverage for this model** also enables fields on the other tabs of the Coverage Settings dialog box.

- 3** Under **Coverage metrics**, select the coverages you want to appear in the coverage report.

For a complete inventory of coverage selections in all four tabs of the Coverage Settings dialog box, see “Specifying Model Coverage Reporting Options” on page 5-11.

- 4** Select **OK** to close the dialog box.
- 5** In the Simulink model window, select **Start > Simulation** or click the **Start** button on the Simulink toolbar to start simulating the model.

If you specify to report model coverage, the Simulink Verification and Validation software saves coverage data for the current run

in the workspace object `covdata` and cumulative coverage data in `covCumulativeData` by default. This data appears in an HTML report that opens in a browser window at the end of simulation.

---

**Note** You cannot run simulations with both model coverage reporting and acceleration options enabled. The Simulink Verification and Validation software disables model coverage reporting if an acceleration mode is enabled.

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

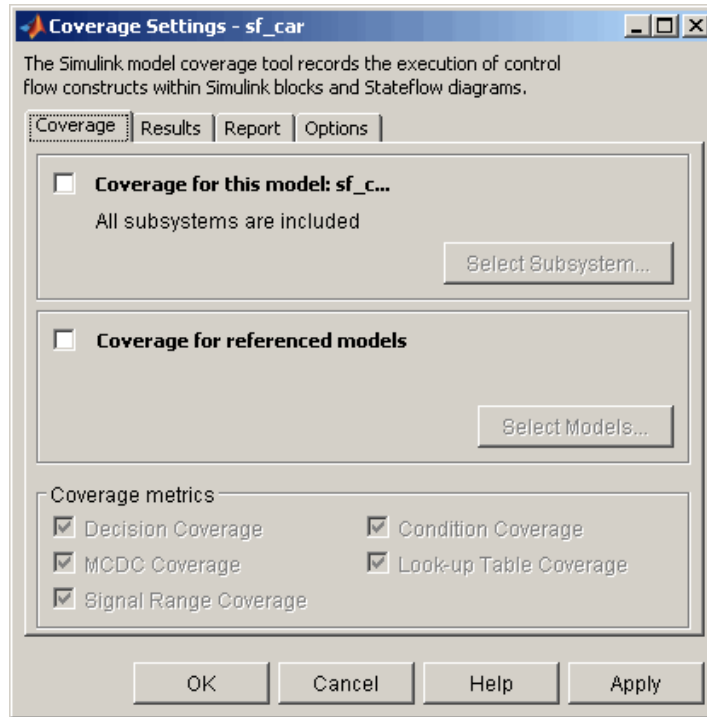
---

## Specifying Model Coverage Reporting Options

In this section...
“Coverage Settings Dialog Box” on page 5-11
“Coverage Tab” on page 5-12
“Results Tab” on page 5-17
“Report Tab” on page 5-18
“Options Tab” on page 5-22

### Coverage Settings Dialog Box

Before starting a model coverage analysis, you need to specify model coverage reporting options. To do this, in a Simulink model window, select **Tools > Coverage Settings**. The Coverage Settings dialog box appears, with the **Coverage** pane displayed.



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

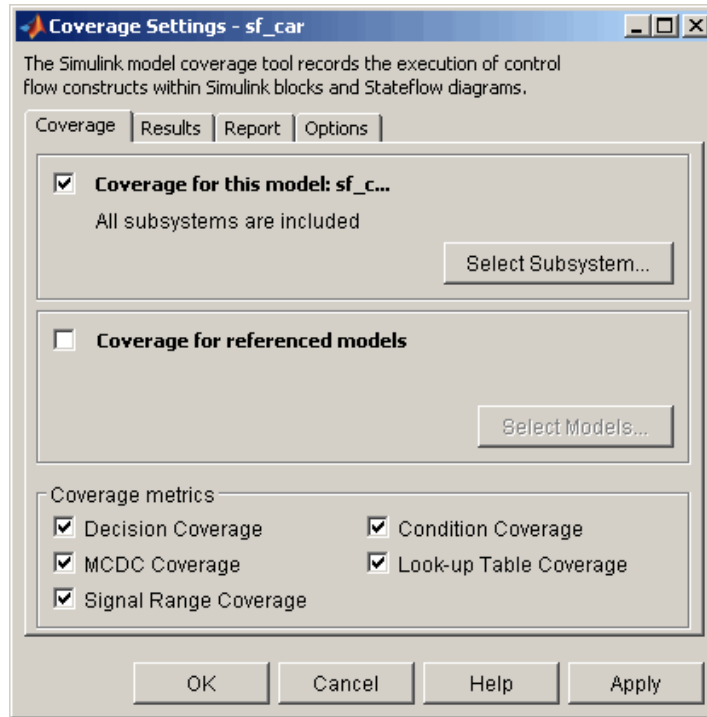
### Coverage Tab

Select the model coverages calculated during simulation in the **Coverage** pane of the Coverage Settings dialog box.

#### Coverage for this model

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





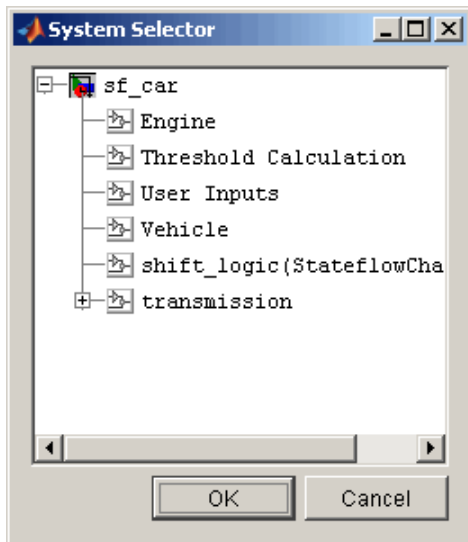
## Select Subsystem

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

To restrict coverage reporting to a particular subsystem:

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

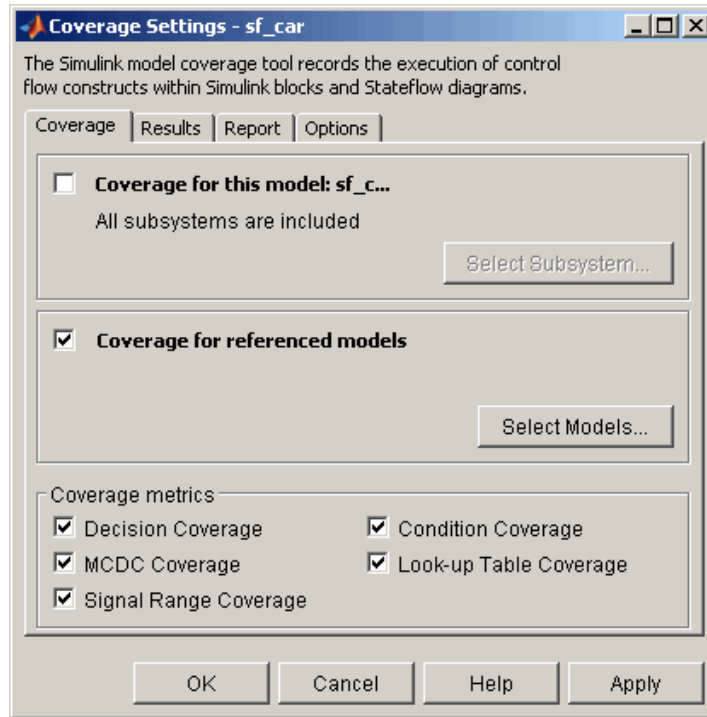
The System Selector dialog box appears.



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

### Coverage for referenced models

Causes the Simulink Verification and Validation software to gather and report the specified model coverages for referenced models during simulation. Selecting the **Coverage for referenced models** option enables the **Select Models** button and the **Coverage metrics** section of the **Coverage** pane.



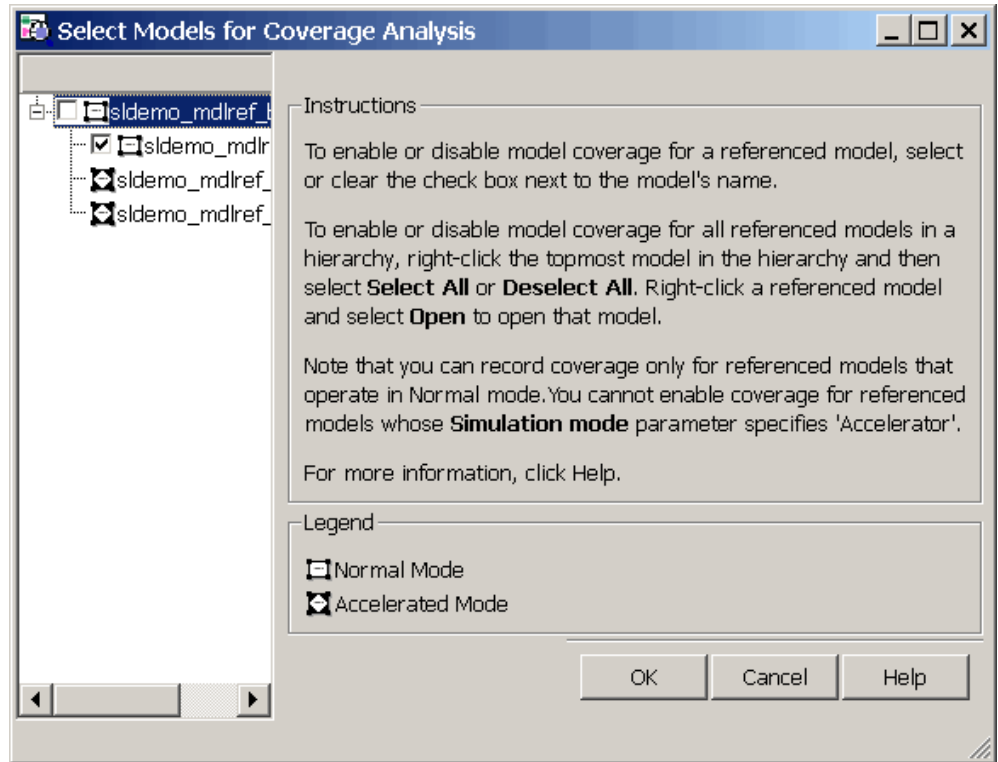
## Select Models

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

To enable coverage reporting for particular referenced models:

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

The Select Models for Coverage Analysis dialog box appears.



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

---

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

---

### Coverage metrics

Select the types of test case coverage analysis that you want the tool to perform (see “Types of Model Coverage” on page 5-2). The Simulink Verification and Validation software gathers and reports the selected types of

coverage for the subsystem, model, and referenced models that you specified elsewhere on the **Coverage** pane.

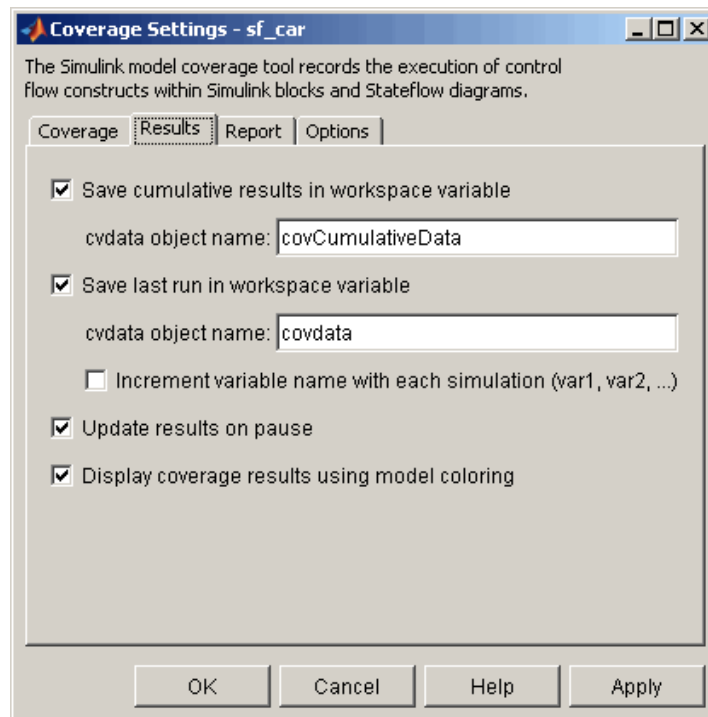
---

**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 5-59.

---

## Results Tab

Select the destination of model coverage results from model coverage in the **Results** pane of the Coverage Settings dialog box.



### **Save Cumulative Results in Workspace Variable**

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

### **Save Last Run in Workspace Variable**

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

### **Increment Variable Name with Each Simulation**

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

### **Update Results on Pause**

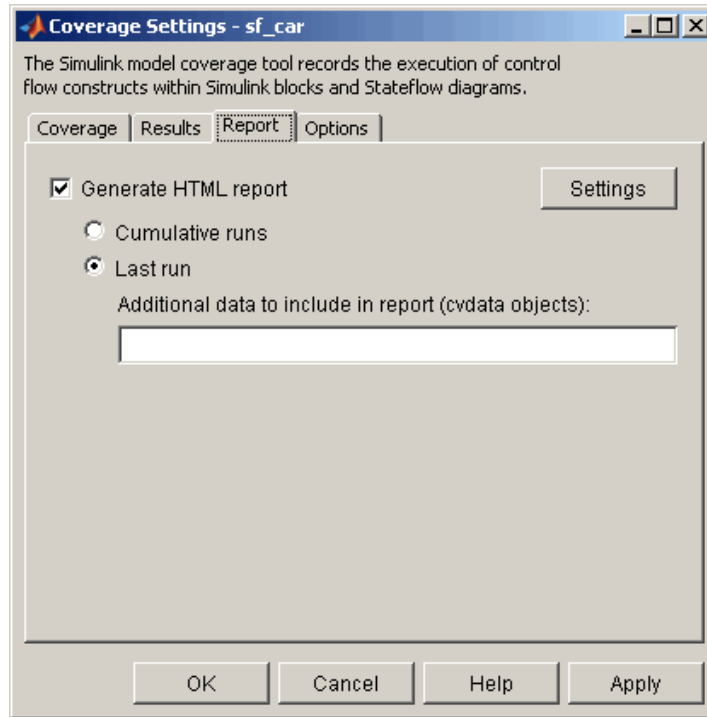
When you pause during simulation the first time, causes the HTML model coverage report to appear with model coverage results recorded up to the pause point. When you resume simulation and later pause or stop simulation, the model coverage report reappears in updated form with coverage results up to the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

### **Display Coverage Results Using Model Coloring**

After simulation, causes coloring of Simulink blocks according to their level of model coverage. Blocks highlighted in light green received full coverage during testing. Blocks highlighted in light red received incomplete coverage. In addition, model coverage results for each block receiving it is available in context-sensitive form. See “Colored Simulink Diagram Coverage Display” on page 5-47 for a complete description.

### **Report Tab**

Select the model coverage test sessions (runs) reported by model coverage in the **Report** pane of the Coverage Settings dialog box.

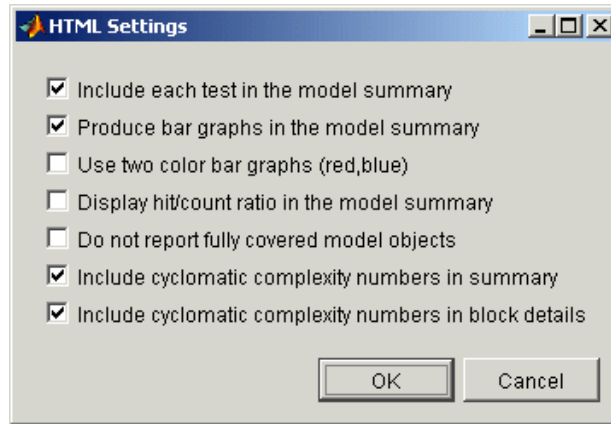


## Generate HTML Report

Causes the Simulink Verification and Validation software to create an HTML report containing the coverage data. The software displays the report in the MATLAB Help browser at the end of the simulation. Click the **Settings** button to select various reporting options (see “Settings” on page 5-19).

## Settings

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



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

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

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

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

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



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

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

### Cumulative Runs

Display the coverage results from successive simulations in the report. For information about this report, see “Cumulative Coverage Reports” on page 5-33

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

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

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

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

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

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

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

### **Last Run**

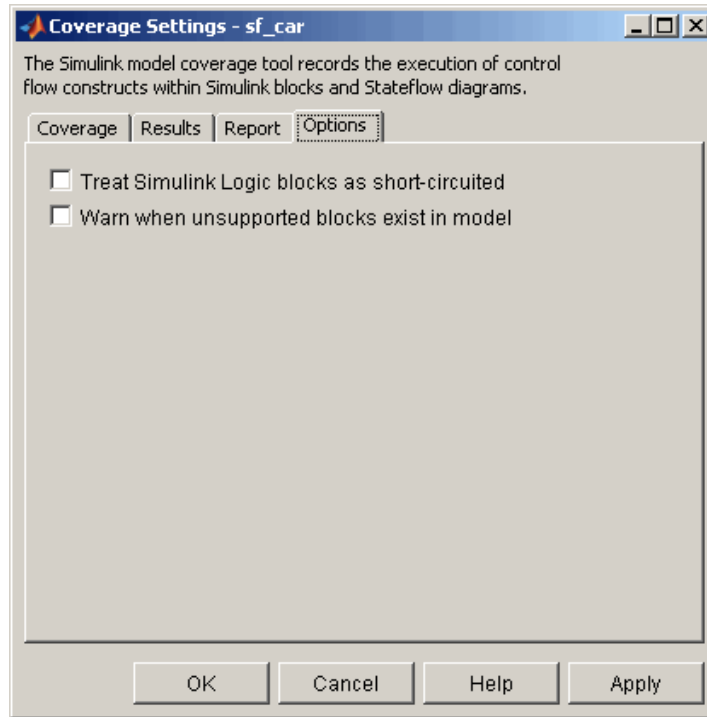
Display only the results of the most recent simulation run in the report.

### **Additional Data to Include in Report**

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

### **Options Tab**

Select important options for model coverage reports in the **Options** pane of the Coverage Settings dialog box.



### Treat Simulink Logic Blocks as Short-Circuited

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

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

---

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

---

### **Warn When Unsupported Blocks Exist in a Model**

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

# Understanding Model Coverage Reports

## In this section...

“About Model Coverage Reports” on page 5-25

“Summary Report Section” on page 5-27

“Details Report Section” on page 5-28

“Decisions Analyzed Table” on page 5-30

“Conditions Analyzed Table” on page 5-31

“MC/DC Analysis Table” on page 5-31

“Cumulative Coverage Reports” on page 5-33




## About Model Coverage Reports

If you enable the **Generate HTML report** option on the **Report** tab of the Coverage Settings dialog box, the Simulink Verification and Validation software creates a model coverage report after it completes a simulation.

If you are recording coverage for reference models, the report displays a summary of each referenced model’s coverage results under **Coverage by Model**.

## Coverage by Model

Top Model: `sldemo_mdhref_fncall`

	Complexity	Decision Coverage
TOTAL COVERAGE		92% 
1. . . <a href="#">sldemo_mdhref_fncall_cntf</a>	3	75% 
2. . . <a href="#">sldemo_mdhref_fncall</a>	6	100% 

Click a model name to view the coverage report for that model.

# Coverage Report for sldemo\_mdref\_fcncall\_cntr

## Model Information

Model Version	1.116
Author	The MathWorks Inc.
Last Saved	Wed May 14 18:41:33 2008

## Simulation Optimization Options

Inline Parameters	on
Block Reduction	forced off
Conditional Branch Optimization	on


## Coverage Options

Logic block short circuiting	off
------------------------------	-----

## Tests

Started Execution: 02-Jun-2008 13:45:15  
Ended Execution: 02-Jun-2008 13:45:16

## Summary

Model Hierarchy/Complexity:	Current Run	Delta	Cumulative
	D1		
1. <a href="#">sldemo_mdref_fcncall_cntr</a>	3	75%	

Otherwise, the Simulink Verification and Validation software displays the coverage report for a single model. The model coverage report contains several parts, each of which is described in the sections that follow.

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

## Summary Report Section

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

- Model Information:
  - Model Version
  - Author
  - Last Saved
- Simulation Optimization Options:
  - Inline Parameters
  - Block Reduction
  - Conditional Branch Optimization
- Coverage Options:
  - Logic block short circuiting

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 a detailed report for a specific subsystem, click the subsystem name in the Summary subsection.

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

## Coverage Report for fuelsys

### Model Information

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

### Simulation Optimization Options

Inline Parameters	off
Block Reduction	forced off
Conditional Branch Optimization	on

### Coverage Options








Logic block short circuiting	off
------------------------------	-----

## Tests

### Test 1

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

## Summary

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

## Details Report Section

The Details section reports the model coverage results in detail.



## Details:

### 1. Model "fuelsys"

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

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	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

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

Parent: [/fuelsys](#)

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

### Switch block "[SwitchControl](#)"

Parent: [fuelsys/EGO sensor](#)

Uncovered Links: 

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

#### Decisions analyzed:

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

The Details section contains a summary of results for the model as a whole, followed by a list of subsystems and charts that the model contains. Subsections on each subsystem and chart follow. Click the name of a subsystem or chart in the model summary to see a detailed report.

Each subsystem 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 block reports, one for each block that contains a decision point in the subsystem.

Each section of the detailed report summarizes the results for the metrics used to test the object (model, subsystem, chart, or block) to which the section applies. The sections for models and subsystems list results for the model and subsystem considered as a covered object and for the contents of the model or subsystem.

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

- 1 Right-click a Simulink block.

A pop-up menu appears.

- 2 In the pop-up menu, select **Coverage > Report**.

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

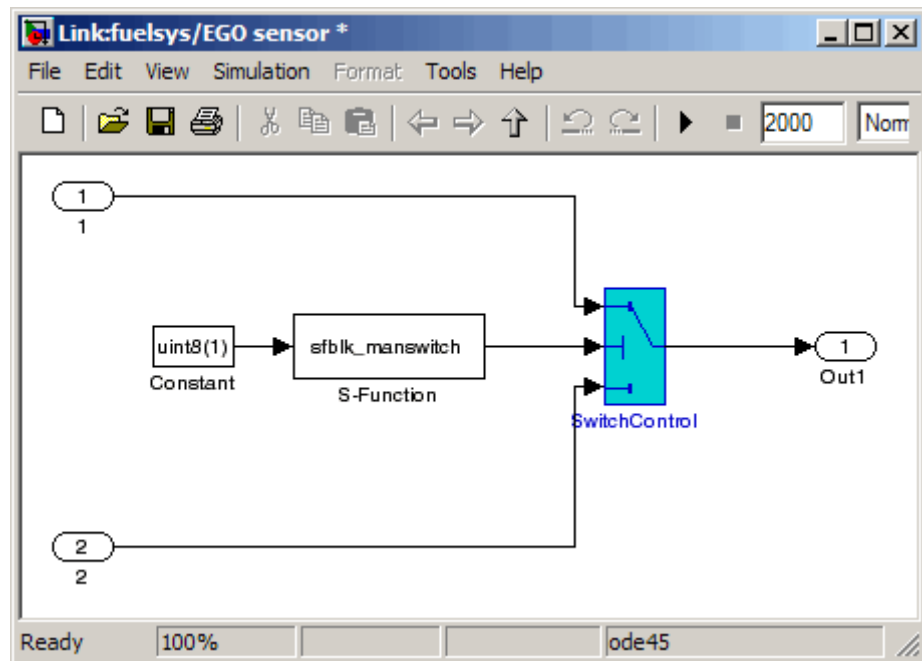
### Decisions Analyzed Table

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

#### Decisions analyzed:

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

The table highlights outcomes that did not occur in red. 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.



## Conditions Analyzed Table

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

### Conditions analyzed:

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

## MC/DC Analysis Table

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

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

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

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

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

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

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

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

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

block to output a false value and whether the value actually occurred during the first (or only) test case included in the report.

If you select **Treat Simulink Logic blocks as short-circuited** in the Coverage Settings dialog box (see “Specifying Model Coverage Reporting Options” on page 5-11), MC/DC coverage analysis does not check whether short-circuited inputs actually occur. The MC/DC details table uses an x in a condition expression (e.g., TFxxx) to indicate short-circuited inputs that were not analyzed by the tool.

**Navigation Arrows.** The section for each block contains a backward and a forward arrow. 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 Reports

To identify untested blocks, states, or transitions, you simulate a series of test cases on your model to maximize the coverage of your model. If you select **Save cumulative results in workspace variable** in the **Results** pane and **Cumulative** runs in the **Report** pane, the results of each simulation are saved and recorded in the report.

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

A cumulative coverage report contains information about:

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

The Summary report after running three test cases for the `slvrv_autopilot_test_harness` model 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. <a href="#">slvndemo_autopilot_test_harness</a>	31	38%	41%	17%	8%	6%	0%	51%	41%	17%
2. <a href="#">Logic</a>	25	34%	38%	17%	9%	8%	0%	47%	38%	17%
3. <a href="#">SF: Logic</a>	24	34%	38%	17%	9%	8%	0%	47%	38%	17%
4. <a href="#">SF: Altitude</a>	11	64%	67%	33%	21%	17%	0%	93%	67%	33%
5. <a href="#">SF: Active</a>	4	38%	NA	NA	13%	NA	NA	88%	NA	NA
6. <a href="#">SF: GS</a>	13	11%	8%	0%	0%	0%	0%	11%	8%	0%
7. <a href="#">SF: Active</a>	6	0%	NA	NA	0%	NA	NA	0%	NA	NA
8. <a href="#">SF: Coupled</a>	3	0%	NA	NA	0%	NA	NA	0%	NA	NA
9. <a href="#">Verify Outputs</a>	5	60%	50%	NA	0%	0%	NA	80%	50%	NA
10. <a href="#">Subsystem1</a>	1	0%	NA	NA	0%	NA	NA	100%	NA	NA
11. <a href="#">Capture time</a>	1	0%	NA	NA	0%	NA	NA	100%	NA	NA
12. <a href="#">Subsystem2</a>	1	100%	NA	NA	0%	NA	NA	100%	NA	NA
13. <a href="#">Capture time</a>	1	100%	NA	NA	0%	NA	NA	100%	NA	NA
14. <a href="#">Subsystem3</a>	1	0%	NA	NA	0%	NA	NA	0%	NA	NA
15. <a href="#">Capture time</a>	1	0%	NA	NA	0%	NA	NA	0%	NA	NA
16. <a href="#">Verification</a>	2	100%	50%	NA	0%	0%	NA	100%	50%	NA

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

Decisions analyzed:

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

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

**Conditions analyzed:**

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

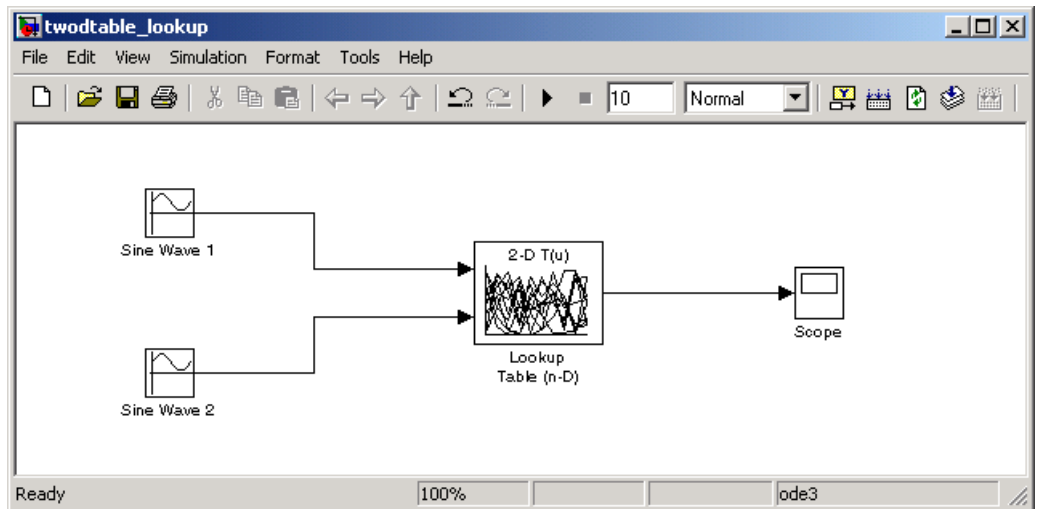
The MC/DC analysis **#n True Out** and **#n False Out** columns show the condition cases for each test case. The **Total Out T** and **Total Out F** column show the cumulative results.

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

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

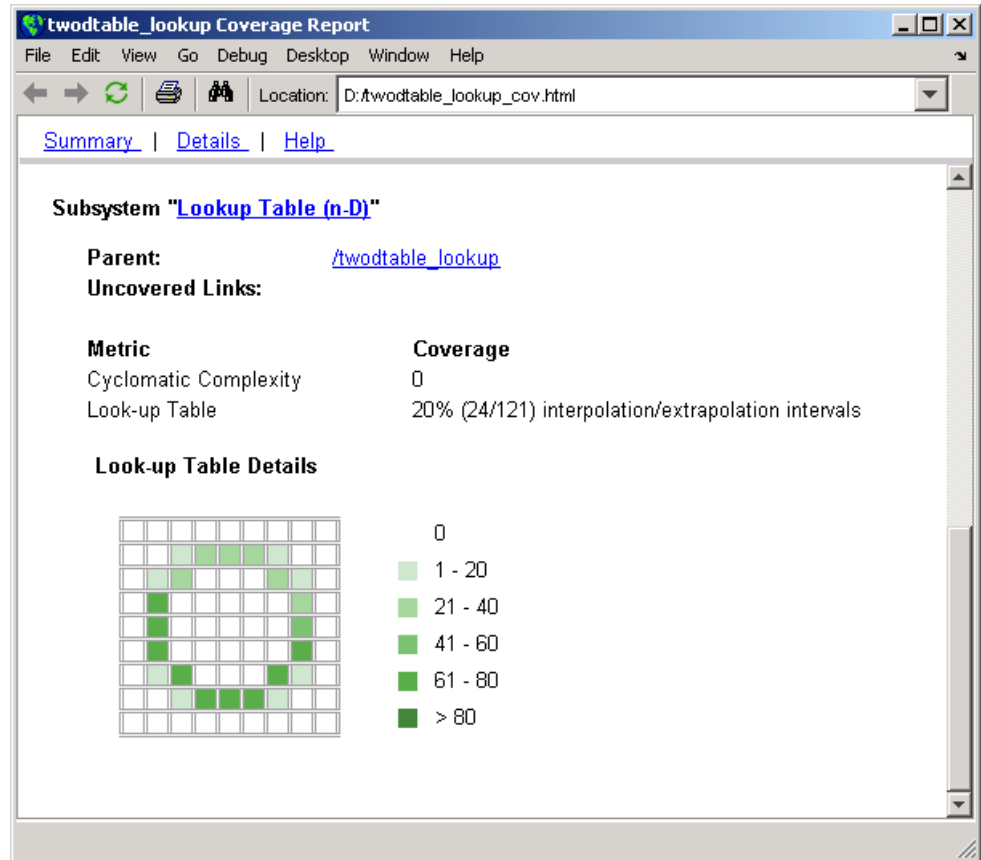
## N-Dimensional Lookup Table Report

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



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



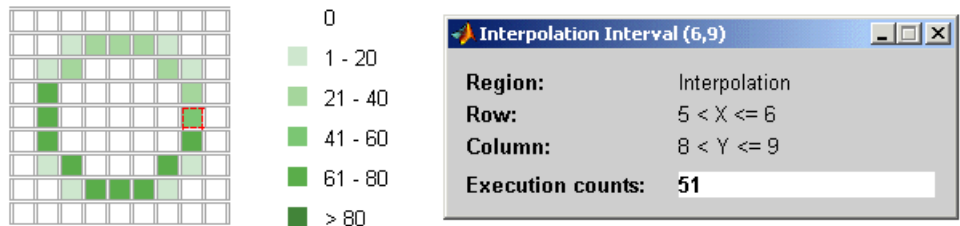


The report contains a two-dimensional table representing the elements of the lookup table. The element indices are represented by the cell border grid lines, which number 10 in each dimension. Areas where the lookup table interpolates between table values are represented by the cell areas. Areas of extrapolation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

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

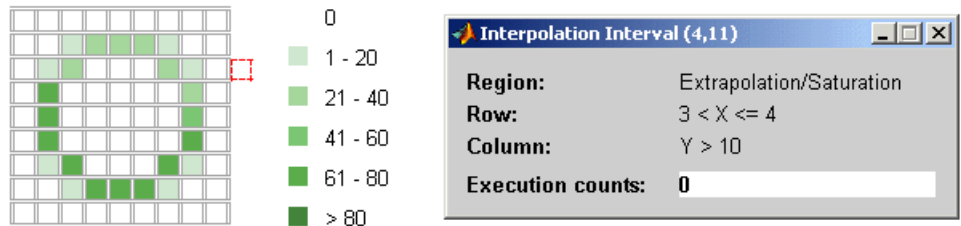
If you click an individual table cell, you receive a dialog that displays the index location of the cell and the exact number of execution counts generated for it during testing. The following example shows the contents of a color shaded cell on the right edge of the circle:

**Lookup Table Details**



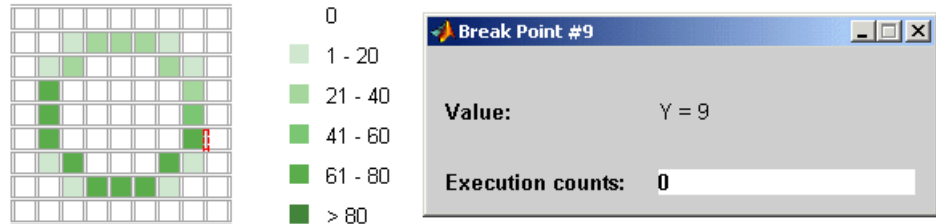
The selected cell is outlined in red. You can also click the extrapolation cells on the edge of the table.

**Lookup Table Details**

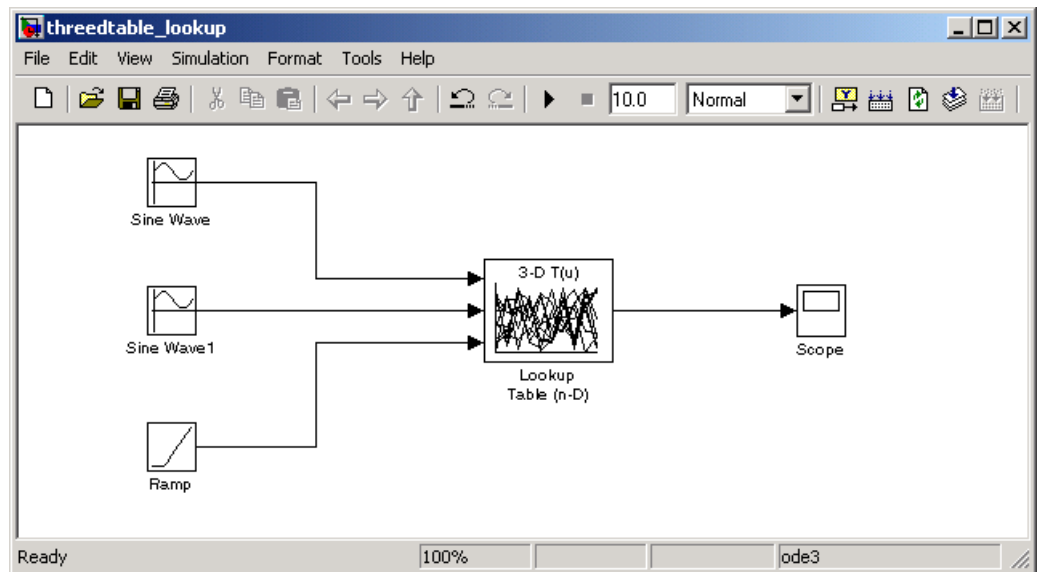


A bold grid line indicates that at least one block input equal to its exact index value occurred during the simulation. Click the border to display the exact number of hits for that index value.

### Lookup Table Details

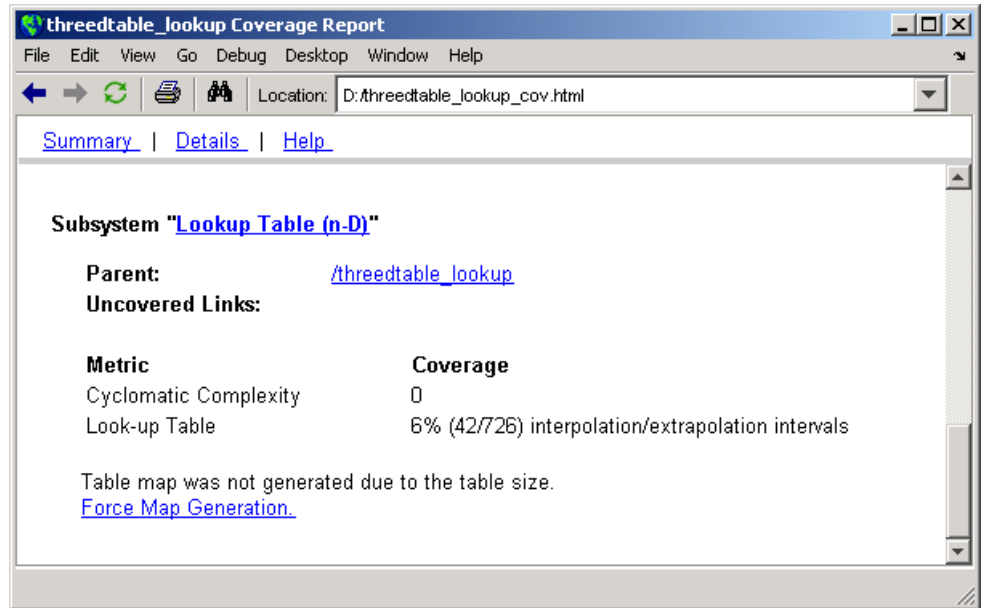


The following example model uses a Lookup Table (n-D) block of 10-by-10-by-5 elements filled with random values.

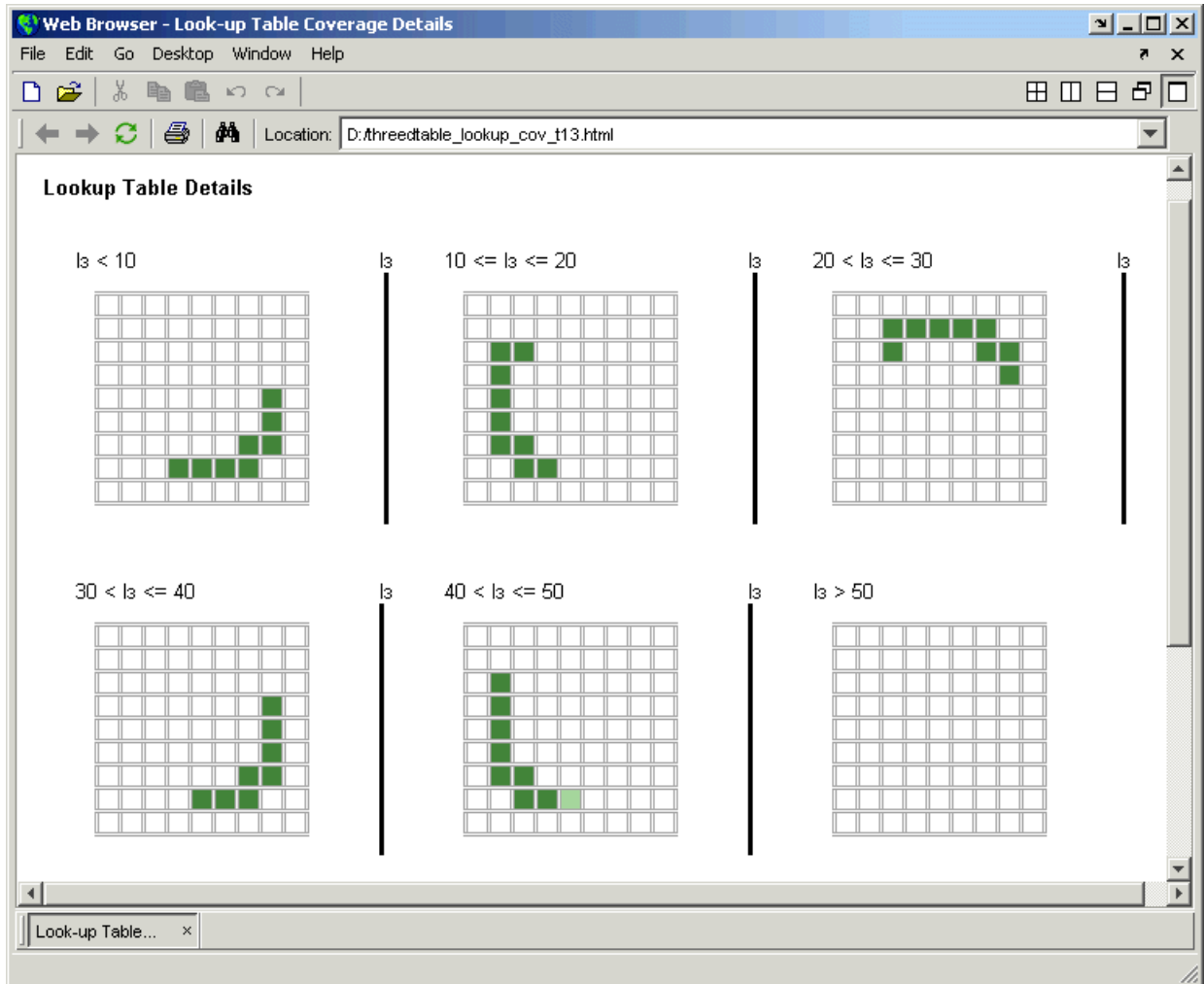


Both the  $x$  and  $y$  table axes have the indices 1, 2, ..., 10, while the  $z$  axis has the indices 10, 20, ..., 50. Lookup table values are accessed with  $x$  and  $y$  indices generated from the two Sine Wave blocks in the preceding example, and a  $z$  index generated from a Ramp block.

After simulation, the following lookup table report appears.



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



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

The vertical bars represent the exact  $z$  index values: 10, 20, 30, 40, 50. If a vertical bar is bold, this indicates that at least one block input was equal to

the exact index value it represents during the simulation. Click a bar to get a report of coverage for the exact index value it represents.

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

## Signal Range Analysis Report

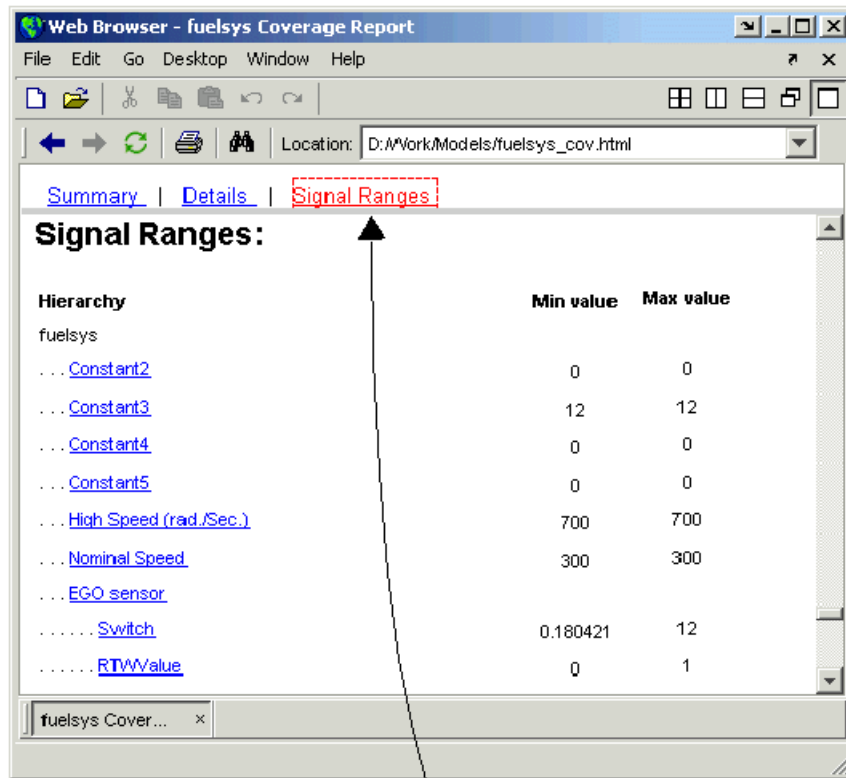
If you select **Signal Range Coverage** in the Coverage Settings dialog box, you receive a Signal Range Analysis report at the bottom of the model coverage report. This report gives you the maximum and minimum signal values at each block in the model measured during simulation.

---

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

---

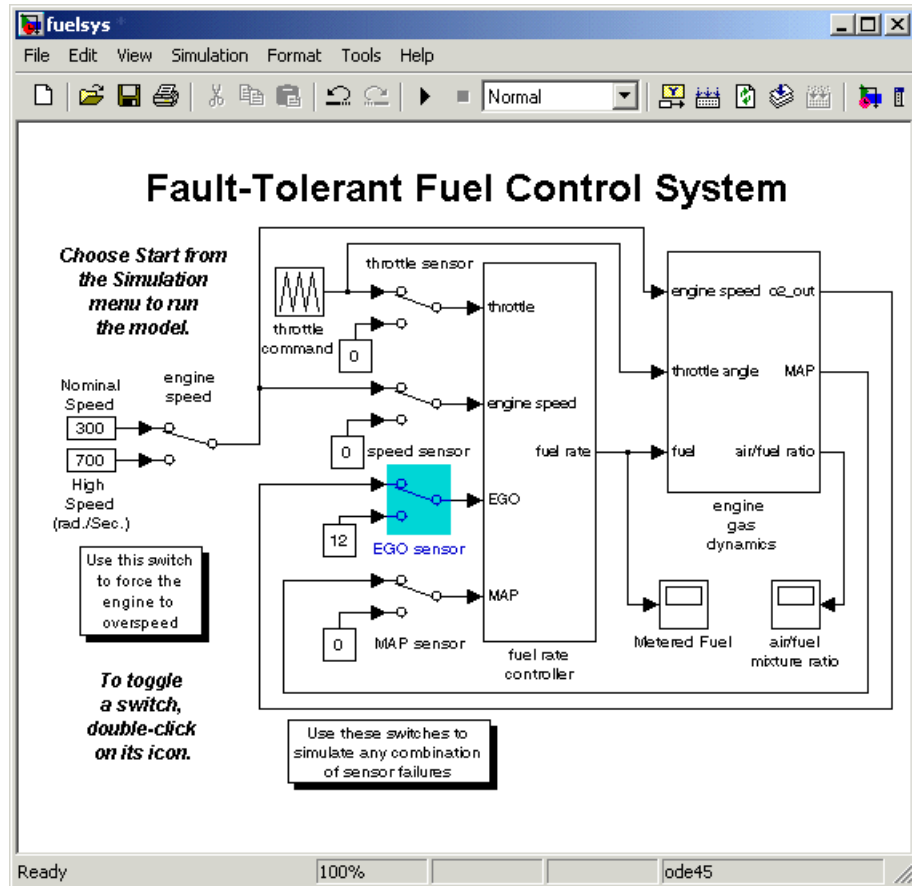
You can access the Signal Range Analysis report quickly with the **Signal Ranges** link in the nonscrolling region at the top of the model coverage report, as shown for the fuelsys model.



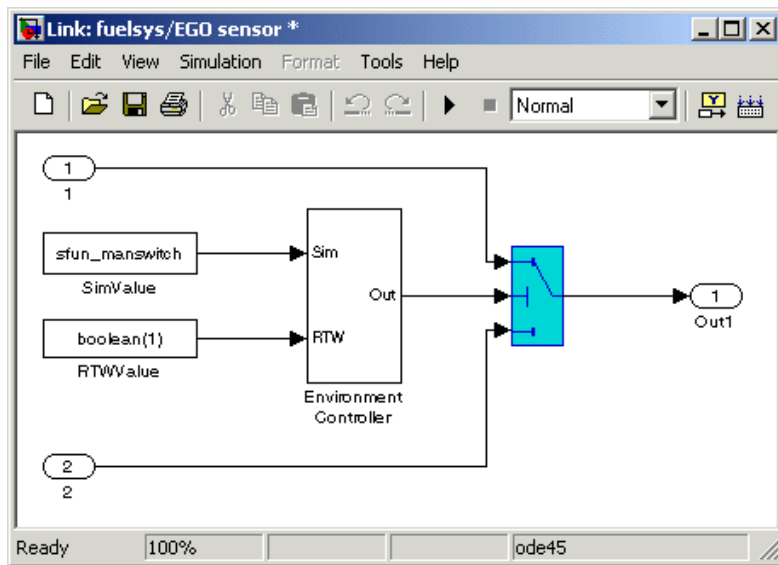
Link to Signal Range Coverage report

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





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



## Colored Simulink Diagram Coverage Display

### In this section...

- “How Model Coverage Highlighting Works” on page 5-47
- “Enabling the Colored Diagram Display” on page 5-47
- “Displaying Model Coverage with Model Coloring” on page 5-48
- “Accessing Coverage Information for Colored Blocks” on page 5-50

### How Model Coverage Highlighting Works

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

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

Coloring is used to highlight structural coverage in Simulink models. When you enable coloring for model coverage results (see “Enabling the Colored Diagram Display” on page 5-47), the tool highlights blocks that received the following types of model coverage:

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

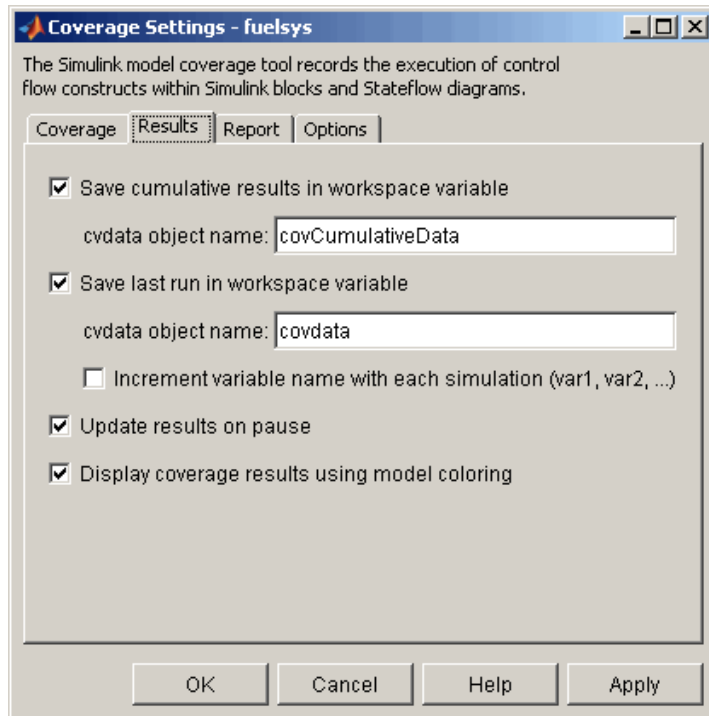
### Enabling the Colored Diagram Display

You enable the model coverage colored diagram display as follows:

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

The Coverage Settings dialog box appears.

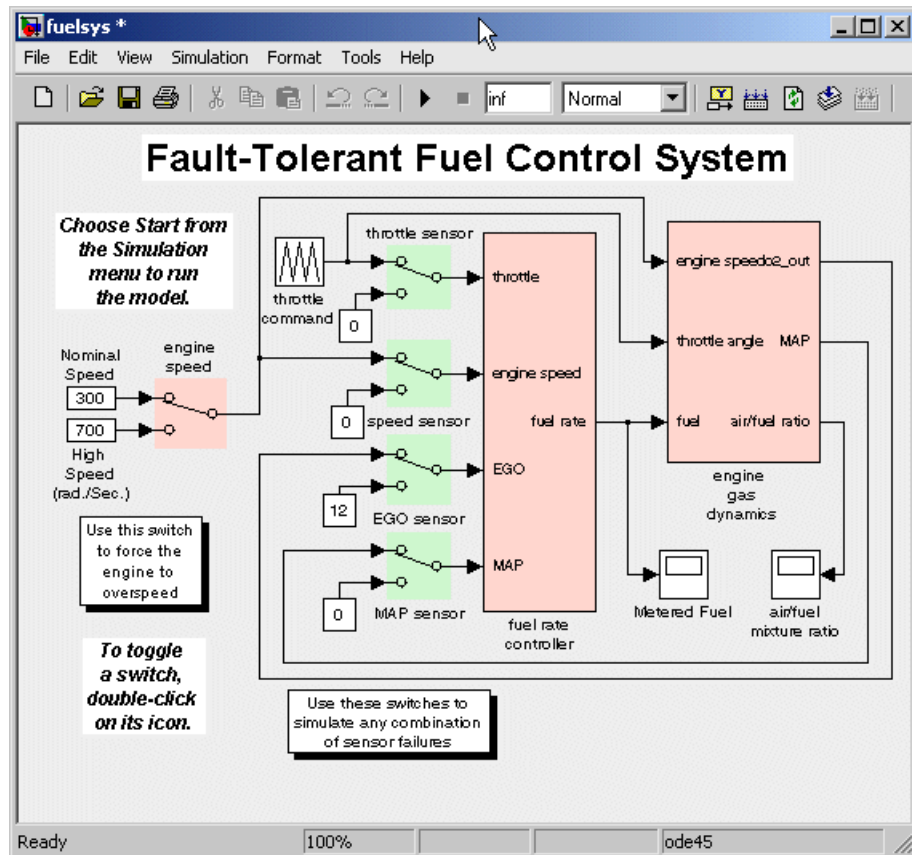
- 2 In the **Coverage** tab of the Coverage Settings dialog box, select **Coverage for this model**.
- 3 Select the **Results** tab, as shown.



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

### Displaying Model Coverage with Model Coloring

You enable display coverage as described in “Enabling the Colored Diagram Display” on page 5-47. After you enable this display, any time that the model generates a model coverage report, individual blocks receiving coverage are highlighted with light green or light red.

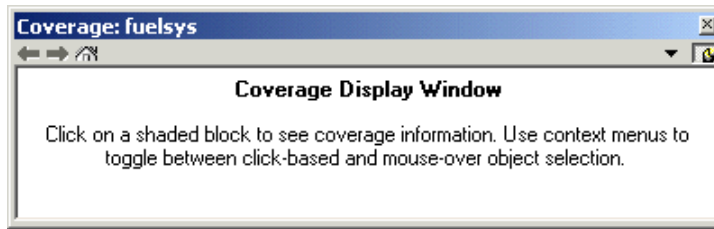


The light green Manual Switch blocks received full coverage during testing. The light red blocks (the engine speed Manual Switch block, and the fuel rate controller and engine gas dynamics subsystems) received incomplete coverage during testing. Blocks with no color highlighting (the Constant blocks,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

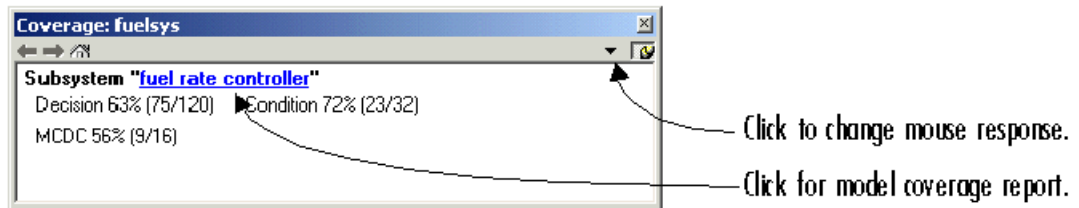
**Note** To restore the Simulink diagram to its original colors, right-click a colored block and select **Coverage** from the resulting context menu followed by **Remove information** from the resulting submenu. Alternatively, you can select **Remove Highlighting** from the Simulink **View** menu or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

## Accessing Coverage Information for Colored Blocks

“Displaying Model Coverage with Model Coloring” on page 5-48 describes the highlighted Simulink diagram that appears after simulation when you enable display coverage with model coloring in the coverage settings for the model. Along with the highlighted Simulink diagram, a Coverage Display window appears, as shown.



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



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

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

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

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

---

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

---

## Using Model Coverage Commands

### In this section...

“About Model Coverage Commands” on page 5-52

“Creating Tests with `cvtest`” on page 5-52

“Running Tests with `cvsim`” on page 5-54

“Producing HTML Reports with `cvhtml`” on page 5-55

“Saving Test Runs to a File with `cvsave`” on page 5-56

“Loading Stored Coverage Test Results with `cvload`” on page 5-56

“Coverage Script Example” on page 5-57

### About Model Coverage Commands

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

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

### Creating Tests with `cvtest`

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

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

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

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



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

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

The following command creates a test with a setup command:

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

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

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

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

Field	Description
modelRefSettings.enable	String specifying one of the following values: <ul style="list-style-type: none"> <li>• <code>Off</code> — Disables coverage for all referenced models</li> <li>• <code>all</code> — Enables coverage for all referenced models</li> <li>• <code>filtered</code> — Enables coverage only for referenced models not listed in the <code>excludedModels</code> subfield</li> </ul>
modelRefSettings.exclude-TopModel	Set to 1 if excluding coverage for the top model is desired
modelRefSettings.excluded-Models	String specifying a comma-separated list of referenced models for which coverage is disabled when <code>modelRefSettings.enable</code> specifies <code>filtered</code>

## Running Tests with `cvsim`

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

---

**Note** You do not have to enable model coverage reporting for the model (see “Creating and Running Test Cases” on page 5-7) to use the `cvsim` command.

---

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

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

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

`cvdo`. But when recording coverage for multiple models in a hierarchy, `cvsim` returns its results in a `cv.cvdatagroup` object.

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

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

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

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

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

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

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

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

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

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

## Producing HTML Reports with `cvhtml`

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

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

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

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

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

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

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

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

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

## **Saving Test Runs to a File with `cvsave`**

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

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

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

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

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

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

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

## **Loading Stored Coverage Test Results with `cvload`**

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

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

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

In the following example, if `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**

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

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

### **Coverage Script Example**

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

```
mdl = 'slvndemo_ratelim_harness';

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

testObj2 = cvtest([mdl, '/Adjustable Rate Limiter']);
testObj2.label='Rising gain that temporarily exceeds slew limit';
```

```
testObj2.setupCmd = 'load(''rising_gain.mat'');';  
testObj2.settings.mcdc = 1;  
  
[dataObj1,T,X,Y] = cvsim(testObj1,[0 2]);  
[dataObj2,T,X,Y] = cvsim(testObj2,[0 2]);  
  
cvhtml('ratelim_report',dataObj1,dataObj2);  
cumulative = dataObj1+dataObj2;  
cvsave('ratelim_testdata',cumulative);
```

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

## Using Model Coverage Commands for Referenced Models

### In this section...

- “Introduction” on page 5-59
- “Creating a Test Group with `cv.cvtestgroup`” on page 5-62
- “Running Tests with `cvsimref`” on page 5-62
- “Extracting Results from `cv.cvdatagroup`” on page 5-63

### Introduction

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

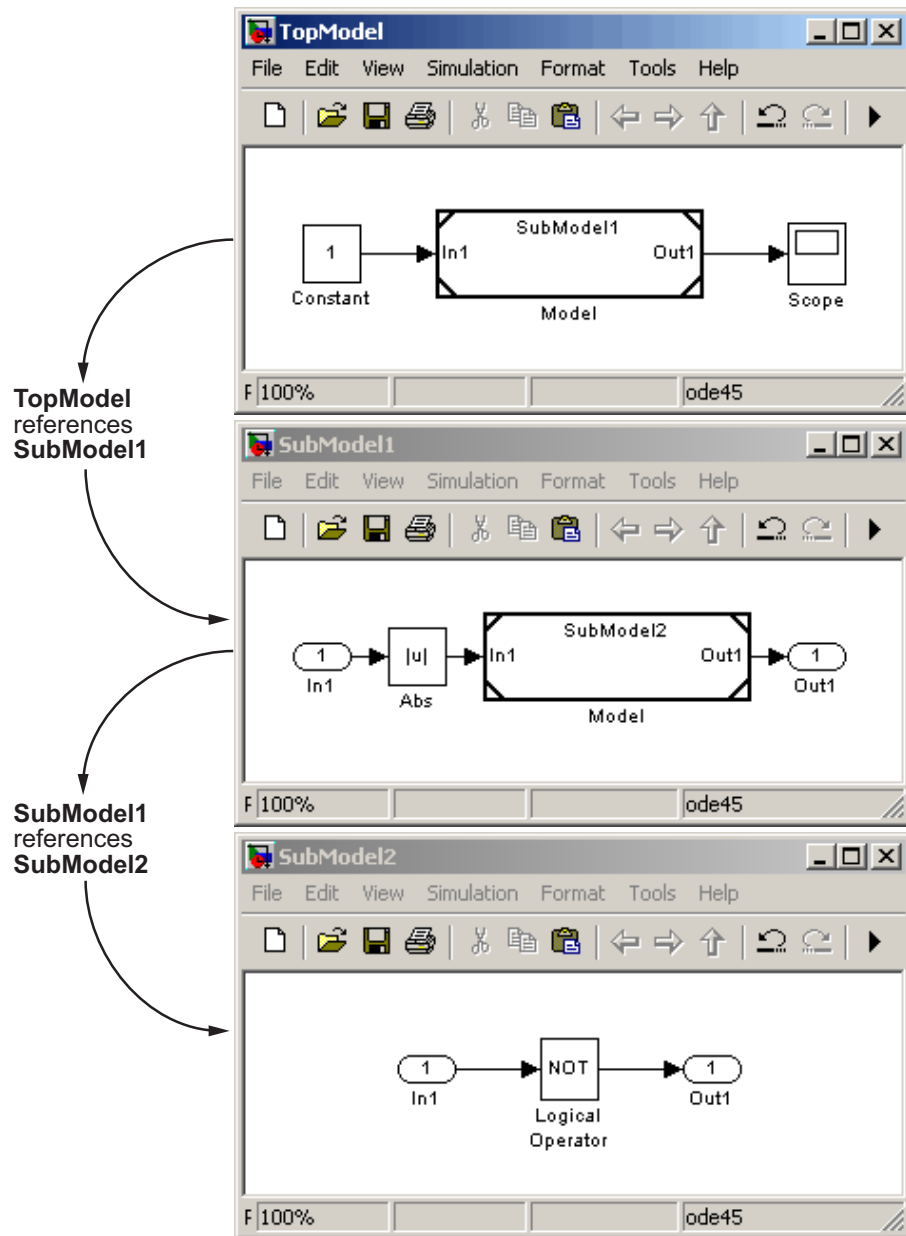
Model coverage supports referenced models that operate in Normal mode. That is, you can record coverage only for those Model blocks whose **Simulation mode** parameter specifies Normal. You can use model coverage commands to record coverage for referenced models (see “Using Model Coverage Commands” on page 5-52). 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 5-62

<b>Step</b>	<b>Description</b>	<b>See...</b>
2	Use <code>cvsimref</code> to simulate the top model in a hierarchy and record coverage results for its referenced models.	“Running Tests with <code>cvsimref</code> ” on page 5-62
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 5-63

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 5-52). But if your model references other models, you might use a different test specification object for each model in the hierarchy. In this case, the `cv.cvtestgroup` object allows you to group together multiple test specification objects. After you create a group of test specification objects, you simulate it using the `cvsimref` function.

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

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

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

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

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

## Running Tests with `cvsimref`

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

---

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

---

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

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

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

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

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

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

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

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

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

## Extracting Results from `cv.cvdatalogroup`

Once you simulate a test group with `cvsimref`, the function returns results that reside in a `cv.cvdatalogroup` 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.cvdatalogroup` object provides methods, such as `allNames` and `get`, which allow you to extract individual `cvdata` objects. For example, enter the following command to obtain a cell array that lists all model names associated with the data group `cvdg`:

```
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 operate on the coverage data of a particular model. For example, you can use the `cvhtml` function to create and display an HTML report of the coverage results (see “Producing HTML Reports with `cvhtml`” on page 5-55).

## Model Coverage for Embedded MATLAB Function Blocks

**In this section...**

“Types of Model Coverage in Embedded MATLAB Function Blocks” on page 5-64

“Creating a Model with Embedded MATLAB Function Block Decisions” on page 5-65

“Understanding Embedded MATLAB Function Block Model Coverage” on page 5-69

### Types of Model Coverage in Embedded MATLAB Function Blocks

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

---

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

---

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

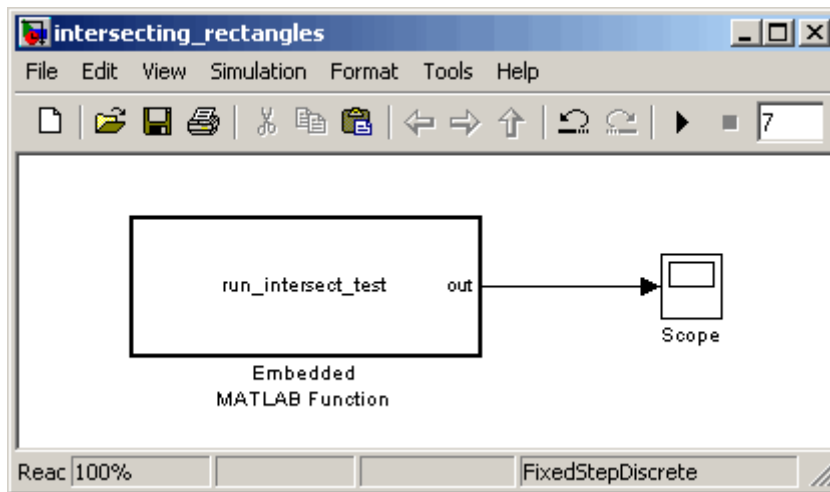
- **Function header** — Decision coverage is 100% if the function or subfunction is executed.
- **if** — Decision coverage is 100% if the `if` expression evaluates to true at least once, and false at least once.
- **switch** — Decision coverage is 100% if every `switch` case is taken, including the fall-through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

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

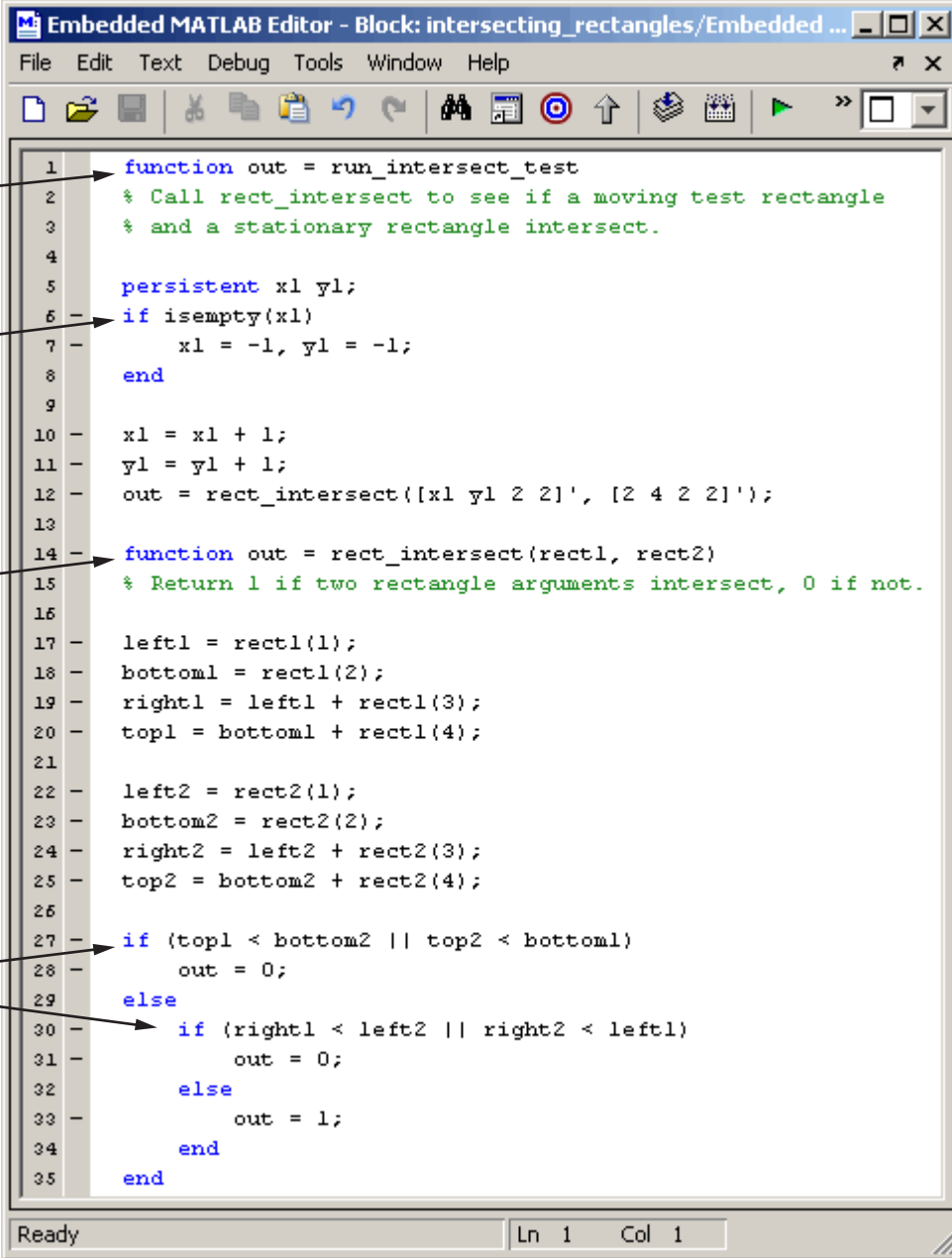
- if statement conditions
- while statement conditions, if present

## Creating a Model with Embedded MATLAB Function Block Decisions

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



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



```
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
```

Function

Decision

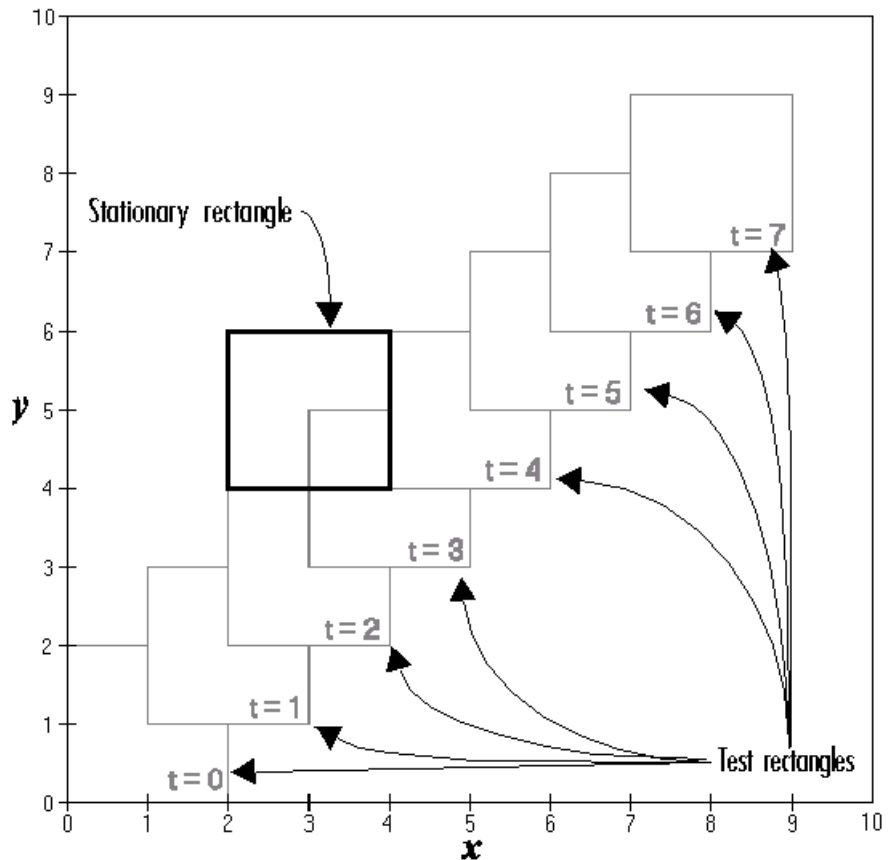
Subfunction

Decisions

Ready Ln 1 Col 1

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

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



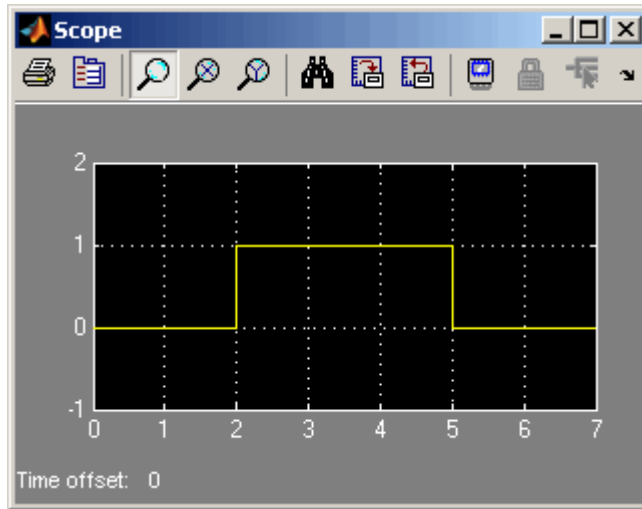
The fixed rectangle is shown in bold with a lower left origin of (2, 4) and a width and height of 2. At time  $t = 0$ , the first test rectangle has an origin of (0, 0) and a width and height of 2. For each succeeding sample, the origin of the test rectangle is incremented by (1, 1). The rectangles at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

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



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

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



## Understanding Embedded MATLAB Function Block Model Coverage

Model coverage reports are generated automatically after a simulation if you specify them. See “Creating and Running Test Cases” on page 5-7 for instructions on how to specify a model coverage report.

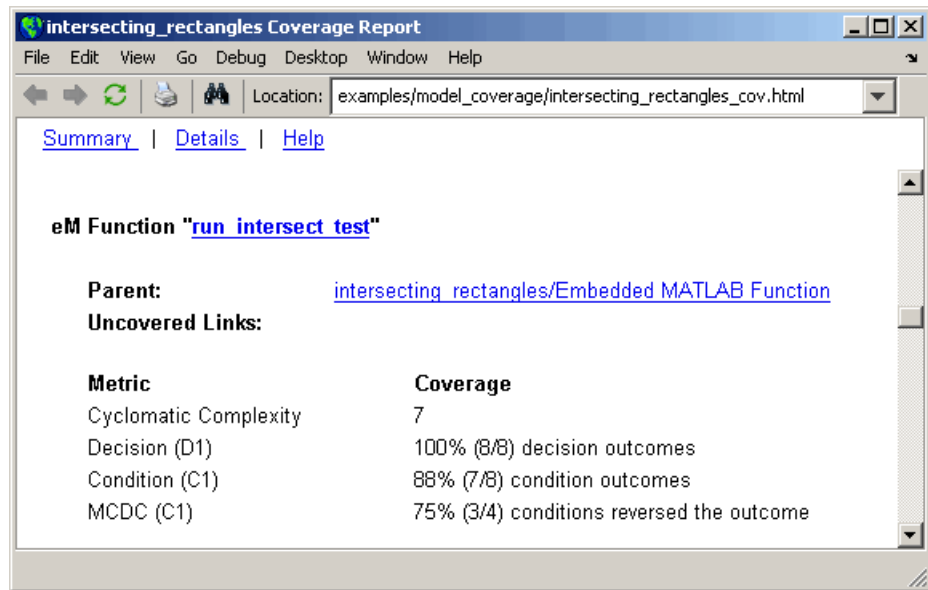
When simulation is finished, the model coverage report appears in a browser window. After the summary for the model, the Details section of the model coverage report reports on each of the parts of the model. Model coverage for the parts of the example model in “Creating a Model with Embedded MATLAB Function Block Decisions” on page 5-65 appears in the following model-block-function order.

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

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

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

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



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

```

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

```

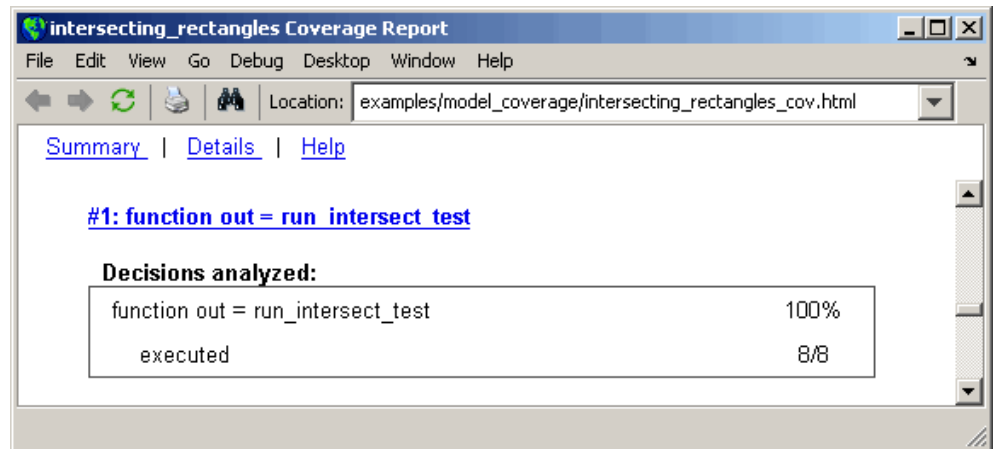
Lines with coverage elements are marked by a highlighted line number in the listing. Line 1 receives decision coverage on whether the top-level function `run_intersect_test` is executed. Line 6 receives decision coverage for its `if` statement. Line 14 receives decision coverage on whether the subfunction

`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 answered by the report for the decision in line 30.

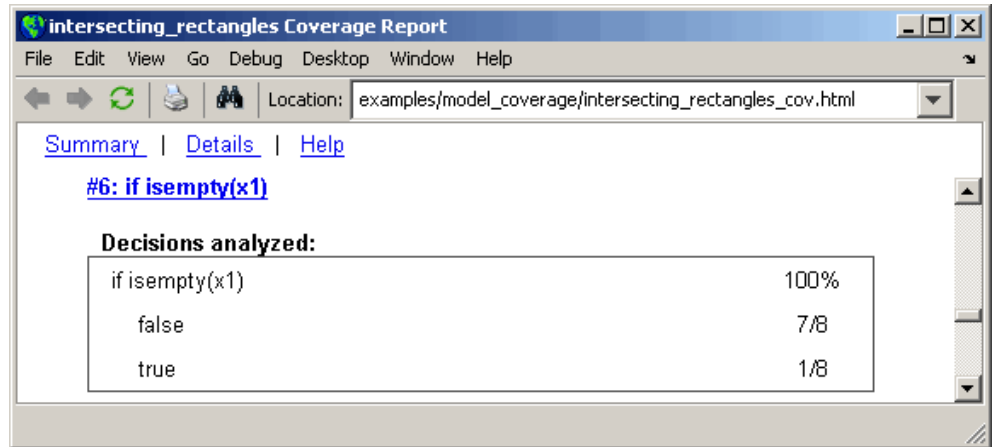
The following subtopics display the coverage for each decision line of `run_intersect_test`. The coverage for each line is titled with the line itself, which is linked to display the function with the line highlighted.

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



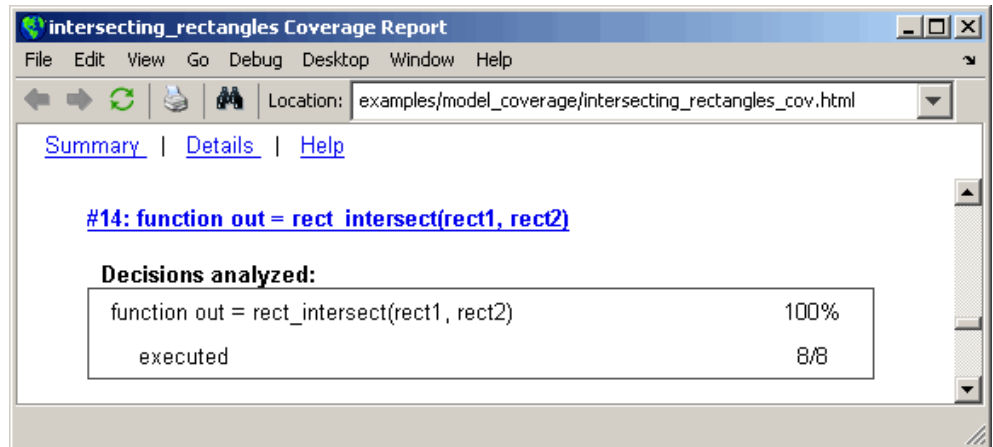
The first line of every function receives coverage analysis indicative of the decision to run the function in response to a call. Coverage for `run_intersect_test` indicates that it executed during testing.

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



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

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



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

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

The 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 three tabs: "Summary", "Details", and "Help". The main content area displays the following information:

**#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, and the remaining three times, it

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

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

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

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



**intersecting\_rectangles Coverage Report**

File Edit View Go Debug Desktop Window Help

Location: examples/model\_coverage/intersecting\_rectangles\_cov.html

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

**#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

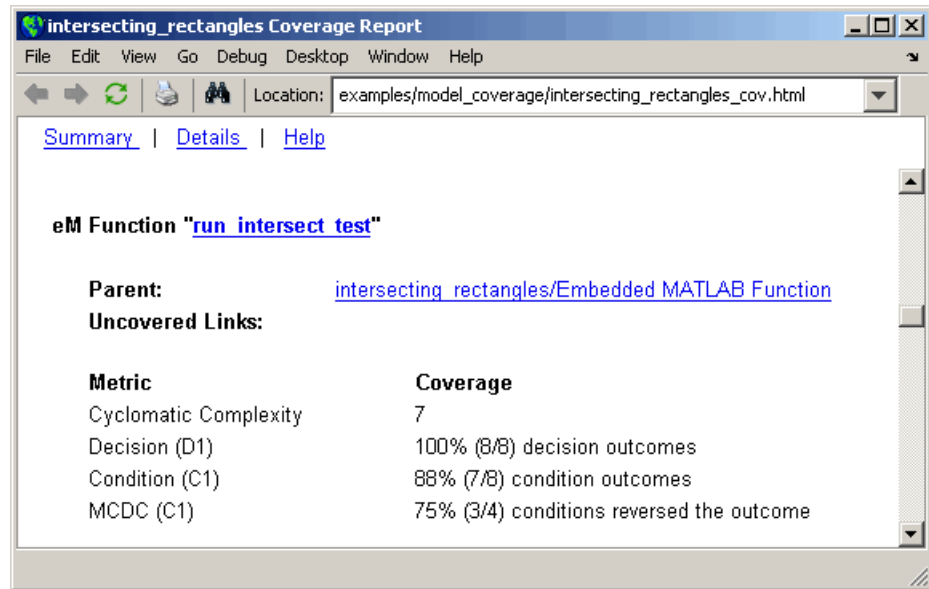
The line 30 decision, `if (right1 < left2 || right2 < left1)`, is nested in the `if` statement of the line 27 decision and is evaluated only if the line 27 decision is false. Because the line 27 decision evaluated false five times, line 30 is evaluated five times, three of which were false. Because both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

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

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

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

**Coverage for run\_intersect\_test.** The metrics that summarize coverage for the entire run\_intersect\_test function are reported prior to its listing and are repeated here as shown.



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

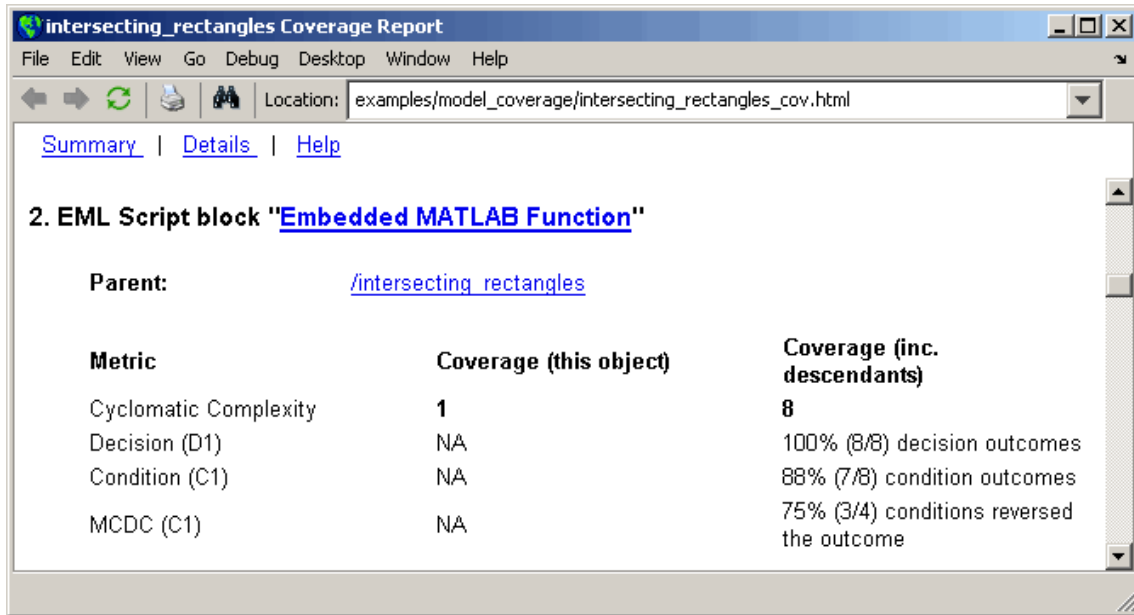
- There are eight decision outcomes reported for run\_intersect\_test in the line reports: one for line 1 (executed), two for line 6 (true and false), one for line 14 (executed), two for line 27 (true and false), and two for line 30 (true and false). The decision coverage for each line shows 100% decision

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

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

### **Model Coverage for the Embedded MATLAB Function Block and the Model**

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



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

**intersecting\_rectangles Coverage Report**

File Edit View Go Debug Desktop Window Help

Location: examples/model\_coverage/intersecting\_rectangles\_cov.html

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

### 1. Model "intersecting\_rectangles"

**Child Systems:** [Embedded MATLAB Function](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	<b>1</b>	<b>9</b>
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

---

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

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

- “Customization Process and Guidelines” on page 6-3
- “Demo and Code Example” on page 6-6
- “Registering Custom Checks, Tasks, and Groups” on page 6-7
- “Creating Callback Functions for Checks” on page 6-10
- “Defining Custom Checks” on page 6-17
- “Defining Check Input Parameters” on page 6-26
- “Defining Check List Views” on page 6-31
- “Defining Check Actions” on page 6-33
- “Defining Custom Tasks” on page 6-37
- “Defining Custom Groups” on page 6-41
- “Defining a Process Callback Function” on page 6-45

- “Formatting Model Advisor Outputs” on page 6-48



## Customization Process and Guidelines

To customize the Model Advisor, create an M-file called `sl_customization.m` and include this file on your MATLAB path.

---

**Note** Do not place an `sl_customization.m` file that customizes the Model Advisor in your root MATLAB directory or any of its subdirectories, with the exception of the `matlabroot/work` directory. Otherwise, the Model Advisor ignores the customizations that the file specifies.

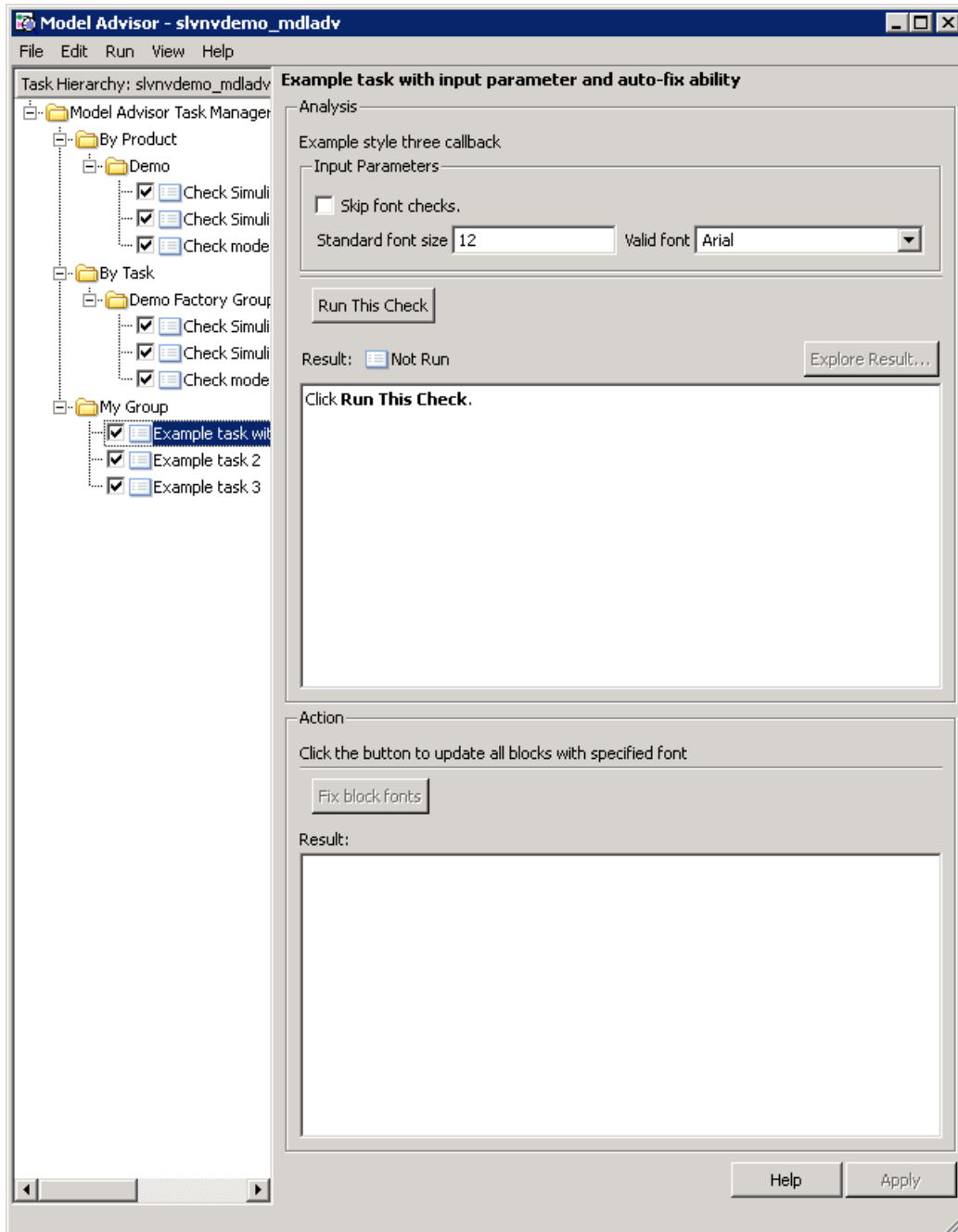
---

The M-file should contain a set of functions for registering and defining custom checks, tasks, and groups. Follow the guidelines in this table.

Function	Description	When Required
<code>sl_customization()</code>	Registers custom checks and tasks with the Simulink customization manager at startup (see “Registering Custom Checks, Tasks, and Groups” on page 6-7).	Required for all customizations to the Model Advisor.
One or more check definition functions	Defines all custom checks (see “Defining Custom Checks” on page 6-17).	Required for custom checks and to add custom checks to the <b>By Product</b> folder.
One or more task definition functions	Defines all custom tasks (see “Defining Custom Tasks” on page 6-37).	Required to add custom checks to the Model Advisor tree, except when adding the checks to the <b>By Product</b> folder. You must write one task for each custom check.

<b>Function</b>	<b>Description</b>	<b>When Required</b>
One or more group definitions	Defines all custom groups (see “Defining Custom Groups” on page 6-41).	Required to add custom tasks to new folders in the Model Advisor tree, except when adding a new subfolder to the <b>By Product</b> folder. You must write one group definition for each new folder.
Check callback functions	Defines the actions of the custom checks (see “Creating Callback Functions for Checks” on page 6-10).	Required for custom checks. You must write one callback function for each custom check.
One process callback function	Specifies actions to be performed at different stages of execution (see “Defining a Process Callback Function” on page 6-45).	Optional.
One or more calls to check input parameters	Specifies input parameters to custom checks (see “Defining Check Input Parameters” on page 6-26).	Optional.
One or more calls to check list views	Specifies calls to the Model Advisor Result Explorer for custom checks (see “Defining Check List Views” on page 6-31).	Optional.
One or more calls to check actions	Specifies actions to take for custom checks (see “Defining Check Actions” on page 6-33).	Optional.

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



### Demo and Code Example

The Simulink Verification and Validation software provides a demo that shows how to customize the Model Advisor by adding:

- Custom checks
- Check input parameters
- Check actions
- Check list views to call the Model Advisor Result Explorer
- Custom tasks to include the custom checks in the Model Advisor tree
- Custom folders for grouping the checks
- A process callback function

The demo also provides the source code of the `sl_customization.m` file that executes the customizations. The following sections present excerpts from this source code to illustrate how to write functions for customizing the Model Advisor.

To run the demo:

- 1** Type `slvndemo_md1adv` at the MATLAB command line.
- 2** Follow the instructions in the model.

## Registering Custom Checks, Tasks, and Groups

### In this section...

“About Registering Custom Checks, Tasks, and Groups” on page 6-7

“Methods for Registering Custom Checks and Groups” on page 6-8

“Code Example: Methods for Registering Custom Checks and Tasks” on page 6-8

### About Registering Custom Checks, Tasks, and Groups

To register checks, tasks, and groups in the Model Advisor, you must create the function `sl_customization()` in the `sl_customization.m` file on your MATLAB path. This function accepts one argument, a handle to an object called `Simulink.CustomizationManager`, as in this example:

```
function sl_customization(cm)
```

The customization manager object includes methods for registering custom checks and tasks. You should use these methods to register customizations specific to your application, as described and demonstrated in the sections that follow.

Simulink reads `sl_customization.m` files when it starts. If you subsequently change the contents of your customization file, update your environment by performing these tasks:

- 1 Close the Model Advisor if it is open.
- 2 If you previously opened the Model Advisor, reinitialize it either by removing the `slprj` folder from your working directory or by closing your model.
- 3 Enter the following command at the MATLAB command line:

```
sl_refresh_customizations
```

- 4 Open your model.
- 5 Start the Model Advisor.

## Methods for Registering Custom Checks and Groups

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

- `addModelAdvisorCheckFcn (@checkDefinitionFcn)`

Adds the checks specified by the check definition function to the **By Product** folder of the Model Advisor unless otherwise specified using `ModelAdvisor.Root.publish`, `ModelAdvisor.Group` or `ModelAdvisor.FactoryGroup`. The `checkDefinitionFcn` argument is a handle to the function that defines all custom checks to be added to Model Advisor as instances of the `ModelAdvisor.Check` class (see “Defining Custom Checks” on page 6-17).

- `addModelAdvisorTaskFcn (@factorygroupDefinitionFcn)`

Adds the checks specified by the `factorygroupDefinitionFcn` to the **By Task** folder of the Model Advisor unless otherwise specified using `ModelAdvisor.Root.publish` or `ModelAdvisor.Group`. The `factorygroupDefinitionFcn` argument is a handle to the function that calls all custom checks to be added to Model Advisor as instances of the `ModelAdvisor.FactoryGroup` class (see “Defining Custom Groups” on page 6-41).

- `addModelAdvisorTaskAdvisorFcn (@taskDefinitionFcn)`

Adds the tasks specified by `taskDefinitionFcn` to the folder specified using `ModelAdvisor.Root.publish` or `ModelAdvisor.Group`. The `taskDefinitionFcn` argument is a handle to the function that defines all custom tasks to be added to Model Advisor as instances of the `ModelAdvisor.Task` class (see “Defining Custom Tasks” on page 6-37).

- `addModelAdvisorProcessFcn (@modelAdvisorProcessFcn)`

Adds the process callback function for the Model Advisor checks (see “Defining a Process Callback Function” on page 6-45).

## Code Example: Methods for Registering Custom Checks and Tasks

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

```
function sl_customization(cm)
```

```
% register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

% register custom factory group
cm.addModelAdvisorTaskFcn(@defineModelAdvisorTasks);

% register custom tasks.
cm.addModelAdvisorTaskAdvisorFcn(@defineTaskAdvisor);

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

## Creating Callback Functions for Checks

### In this section...

“About Check Callback Functions” on page 6-10

“Simple Check Callback Function” on page 6-10

“Detailed Check Callback Function” on page 6-11

“Check Callback Function with Hyperlinked Results” on page 6-13

### About Check Callback Functions

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

- “Simple Check Callback Function” on page 6-10
- “Detailed Check Callback Function” on page 6-11
- “Check Callback Function with Hyperlinked Results” on page 6-13

All styles of check callback functions provide one or more return arguments for displaying the results after executing the check. In some cases, return arguments are strings or cell arrays of strings that support embedded HTML tags for text formatting. It is recommended that you use the Model Advisor Formatting API to format your outputs, as described in “Formatting Model Advisor Outputs” on page 6-48, and limit HTML tags to be compatible with alternate output formats.

### Simple Check Callback Function

Use a simple callback function to return a simple status, perhaps to indicate whether the model passed or failed the check, or to provide a recommendation for correcting an issue. The keyword for the simple callback function is `StyleOne`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-17).

The simple 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.

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

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

if strcmp(get_param(bdroot(system), 'ScreenColor'), 'white')
    result = ModelAdvisor.Text('Passed', {'pass'});
    mdladvObj.setCheckResultStatus(true); % set to pass
    mdladvObj.setActionEnable(false);
else
    result = ModelAdvisor.Text(['It is recommended to select a Simulink '...
        'window screen color of white to ensure a readable and printable model.']);
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
```

## Detailed Check Callback Function

Use the detailed check callback function to return and organize results as strings in a layered, hierarchical fashion. The function provides two output arguments that allow you to associate text descriptions with one or more paragraphs of detail. The keyword for the simple callback function is `StyleTwo`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-17).

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.

Argument	I/O Type	Description
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 to make it easy to locate problem areas in your model or subsystem. The keyword for this type of callback function is `StyleThree`. This keyword is required for the check definition (see “Defining Custom Checks” on page 6-17).

This callback function takes the following arguments:

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

Argument	I/O Type	Description
ResultDescription	Output	Cell array of MATLAB strings that supports the Model Advisor Formatting API calls or embedded HTML tags for text formatting.
ResultDetails	Output	Cell array of cell arrays, each of which contains one or more Simulink objects such as blocks, ports, lines, and Stateflow charts. The objects must be in the form of a handle or Simulink path.

---

**Note** The `ResultDetails` cell array must be the same length as the `ResultDescription` cell array.

---

The Model Advisor automatically concatenates each string from `ResultDescription` with the corresponding array of objects from `ResultDetails`. The Model Advisor displays the contents of `ResultDetails` as a set of hyperlinks, one for each object returned in the cell arrays. When you click a hyperlink, the Model Advisor displays the target object highlighted in your Simulink model.

The following is an example of a check callback function with hyperlinked results. This example checks a model for consistent use of font type and font size in its blocks. It also contains input parameters, actions, and a call to the Model Advisor Result Explorer, which are discussed in later sections.

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

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);
needEnableAction = false;
% get input parameters
inputParams = mdladvObj.getInputParameters;
skipFontCheck = inputParams{1}.Value;
```

```

regularFontSize = inputParams{2}.Value;
regularFontName = inputParams{3}.Value;
if skipFontCheck
    ResultDescription{end+1} = ModelAdvisor.Paragraph('Skipped. ');
    ResultDetails{end+1}    = {};
    return
end
regularFontSize = str2double(regularFontSize);
if regularFontSize<1 || regularFontSize>=99
    mdladvObj.setCheckResultStatus(false);
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['Invalid font size. '...
        'Please enter a value between 1 and 99']);
    ResultDetails{end+1}    = {};
end

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

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

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

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font other than ' regularFontName ': ']);
    ResultDetails{end+1}    = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult, 'object');
    myLVParam.Attributes = {'FontName'}; % name is default property
    mdladvObj.setListViewParameters({myLVParam});
    needEnableAction = true;
else

```

```

        ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font names '...
            'are identical.']);
        ResultDetails{end+1}     = {};
    end

% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font size for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font size other than ' ...
        num2str(regularFontSize) ': ']);
    ResultDetails{end+1}     = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font size blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult,'object');
    myLVParam.Attributes = {'FontSize'}; % name is default property
    mdladvObj.setListViewParameters...
        ({mdladvObj.getListViewParameters{:}}, myLVParam);
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font sizes '...
        'are identical.']);
    ResultDetails{end+1}     = {};
end

mdladvObj.setActionEnable(needEnableAction);
mdladvObj.setCheckErrorSeverity(1);

```

If you run **Example task with input parameter and auto-fix ability** for the `slvndemo_mdladv` model in the Model Advisor, you can view the hyperlinked results. Clicking the first hyperlink, `slvndemo_mdladv/Input`, displays the Simulink model with the Input block highlighted.

## Defining Custom Checks

### In this section...

“About Custom Checks” on page 6-17

“Properties of Custom Checks” on page 6-17

“Defining Where Custom Checks Appear” on page 6-22

“Code Example: Check Definition Function” on page 6-23

### About Custom Checks

Custom checks allow you to create a new check to use in the Model Advisor. You define custom checks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Check` class. You must define one instance of this class for each custom check that you want to add to the Model Advisor, and register the custom check as described in “Registering Custom Checks, Tasks, and Groups” on page 6-7. The sections that follow describe how to define custom checks.

### Properties of Custom Checks

The following table describes the properties of the `ModelAdvisor.Check` class:

Property	Data Type	Default Value	Description
Title	String	'' (null string)	Name of the check as it should appear in the Model Advisor.

Property	Data Type	Default Value	Description
ID	String	' ' (null string)	<p>Permanent, unique identifier for the check that you must specify.</p> <hr/> <p><b>Caution</b></p> <ul style="list-style-type: none"> <li>• The value of ID must remain constant.</li> <li>• The Model Advisor generates an error if ID is not unique.</li> <li>• Tasks and factory groups should refer to checks by ID.</li> </ul> <hr/>
TitleTips	String	' ' (null string)	Description of the check, which the Model Advisor displays in its right pane when you view details about the check.
CallbackHandle	Function handle	[] (empty handle)	Handle to the callback function for the check.
CallbackContext	Enumeration	'None'	<p>Context for checking the model or subsystem:</p> <ul style="list-style-type: none"> <li>• None = No special requirements</li> <li>• PostCompile = Model must be compiled</li> </ul>



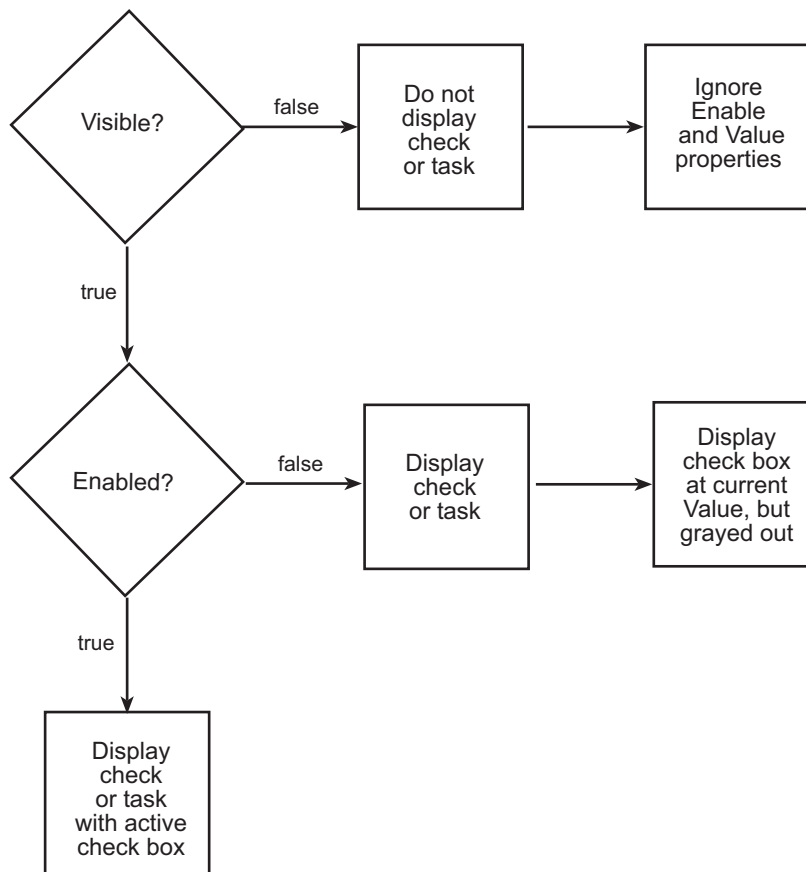
<b>Property</b>	<b>Data Type</b>	<b>Default Value</b>	<b>Description</b>
CallbackStyle	Enumeration	'StyleOne'	Type of callback function: <ul style="list-style-type: none"> <li>• StyleOne = Simple check callback function</li> <li>• StyleTwo = Detailed check callback function</li> <li>• StyleThree = Check callback function with hyperlinked results</li> </ul>
Visible	Boolean	true	Show or hide check? <ul style="list-style-type: none"> <li>• true = Display check in the Model Advisor</li> <li>• false = Hide check</li> </ul>
Enable	Boolean	true	Can user enable and disable check? <ul style="list-style-type: none"> <li>• true = Display check box control</li> <li>• false = Check box control is unavailable</li> </ul>
Value	Boolean	true	Initial status: <ul style="list-style-type: none"> <li>• true = Check is enabled</li> <li>• false = Check is disabled</li> </ul>

Property	Data Type	Default Value	Description
LicenseName	Cell array	{ } (empty cell array)	<p>Cell array of names of product licenses required to enable the check. The Model Advisor does not display the check if license requirements are not met.</p> <hr/> <p><b>Tip</b> To find the correct text for license strings, type <code>help license</code> at the MATLAB command line.</p> <hr/>

Property	Data Type	Default Value	Description
Result	Cell array	{ } (empty cell array)	<p>Cell array used for storing the results returned by the callback function referenced by <code>CallbackHandle</code>.</p> <hr/> <p><b>Tip</b> To set the icon associated with the check, use the <code>Simulink.ModelAdvisor setCheckResultStatus</code> and <code>setCheckErrorSeverity</code> methods.</p> <hr/>
ListViewVisible	Boolean	false	<p>A boolean value that sets the status of the Explore Result button.</p> <ul style="list-style-type: none"> <li>• <code>true</code> = Display the <b>Explore Result</b> button.</li> <li>• <code>false</code> = Hides the <b>Explore Result</b> button.</li> </ul>

### How Visible, Enable, and Value Properties Interact

Typically, you modify the behavior of the `Visible`, `Enable`, and `Value` properties in a process callback function (see “Defining a Process Callback Function” on page 6-45). The following chart illustrates how these properties interact:



### Defining Where Custom Checks Appear

You can specify where the Model Advisor places custom checks within the Model Advisor tree using the following guidelines.

- To place a check in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class. See “Defining Custom Groups” on page 6-41.
- To place a check in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See “Defining Custom Groups” on page 6-41.

- To place a check in the **By Product** folder, use the `ModelAdvisor.Root.publish` method.

When you add a check using the `ModelAdvisor.Root.publish` method, the Model Advisor creates a `ModelAdvisor.Task` object for the check. You use the `ModelAdvisor.Task` object when you specify an action callback function for the check. See “Action Callback Function” on page 6-34.

## 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 for Checks” on page 6-10. The check definition function returns a cell array of custom checks to be added to the Model Advisor.

```
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;

% --- 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]);
% 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});
% define action (fix) operation
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
myAction.Description=...
    'Click the button to update all blocks with specified font';
rec.setAction(myAction);
% add 'Explore Result' button
rec.ListViewVisible = true;
% publish check into By Product > Demo group.
mdladvRoot.publish(rec, 'Demo');

% --- sample check 2
rec = ModelAdvisor.Check('com.mathworks.sample.Check2');
rec.Title = 'Check Simulink window screen color';
rec.TitleTips = 'Example style one callback';
rec.setCallbackFcn(@SampleStyleOneCallback,'None','StyleOne');
% define action (fix) operation
myAction2 = ModelAdvisor.Action;
myAction2.setCallbackFcn(@sampleActionCB2);
myAction2.Name='Fix window screen color';
myAction2.Description=...
    'Click the button to change Simulink window screen color to white';
rec.setAction(myAction2);
% publish check into By Product > Demo group.
mdladvRoot.publish(rec, 'Demo');

% --- sample check 3
rec = ModelAdvisor.Check('com.mathworks.sample.Check3');
rec.Title = 'Check model optimization settings';
rec.TitleTips = 'Example style two callback';
rec.setCallbackFcn(@SampleStyleTwoCallback,'None','StyleTwo');
% define action (fix) operation
myAction3 = ModelAdvisor.Action;
```

```
myAction3.setCallbackFcn(@sampleActionCB3);
myAction3.Name='Fix model optimization settings';
myAction3.Description='Click the button to turn on model optimization settings';
rec.setAction(myAction3);
% publish check into By Product > Demo group.
mdladvRoot.publish(rec, 'Demo');
```

## Defining Check Input Parameters

### In this section...

“About Input Parameters” on page 6-26

“Properties of Input Parameters” on page 6-26

“Specifying Input Parameter Layout” on page 6-28

“Code Example: Input Parameter Definition” on page 6-29

### About Input Parameters

Input parameters allow you to request input from the user for a Model Advisor check. You define input parameters using the `ModelAdvisor.InputParameter` class inside a custom check function (see “Defining Custom Checks” on page 6-17). You must define one instance of this class for each input parameter that you want to add to a Model Advisor check.

### Properties of Input Parameters

The following table describes the properties of the `ModelAdvisor.InputParameter` class:

Property	Data Type	Default Value	Description
Name	String	'' (null string)	Name of the input parameter as it should appear in the custom check.
Type	Enumeration	'' (null string)	Type of input parameter. Used with <code>Value</code> or <code>Entries</code> to define input parameters. See the following table for details.



Property	Data Type	Default Value	Description
Value	Depends on Type. See following table.	Depends on Type. See following table.	Value of the input parameter. This property is valid only when the Type is Bool, String, Enum, or ComboBox. See following table.
Entries	Depends on Type. See following table.	Depends on Type. See following table.	This property is valid only when the Type is Enum, ComboBox, or PushButton. See following table.
Description	String	' ' (null string)	Description of the parameter, which the Model Advisor displays in the right pane when you view details about the check.

### Types of Input Parameters

Type	Data Type	Default Value	Description
Bool	Boolean	false	A check box.
String	String	' ' (null string)	A text box.
Enum	Cell array	First entry in the menu	A drop-down menu. <ul style="list-style-type: none"> <li>• Use Entries to define the entries in the menu.</li> <li>• Use Value to indicate a specific entry in the menu.</li> </ul>

**Types of Input Parameters (Continued)**

Type	Data Type	Default Value	Description
ComboBox	Cell array	First entry in the menu	<p>A drop-down menu that allows the user to type in a value or choose a value from the menu.</p> <ul style="list-style-type: none"> <li>• Use <code>Entries</code> to define the entries in the menu.</li> <li>• Use <code>Value</code> to indicate a specific entry in the menu or to enter a value not in the menu.</li> </ul>
PushButton	N/A	N/A	<p>A button.</p> <p>When you click the button, the callback function specified by <code>Entries</code> is called.</p>

**Specifying Input Parameter Layout**

You can specify the layout of input parameters in the right pane of the Model Advisor window in an input parameter definition. Use the following methods to place input parameters.

- `ModelAdvisor.Check.setInputParametersLayoutGrid`  
Specifies the size of the input parameter grid.
- `ModelAdvisor.InputParameter.setRowSpan`  
Specifies the number of rows the parameter occupies in the Input Parameter layout grid.
- `ModelAdvisor.InputParameter.setColSpan`

Specifies the number of columns the parameter occupies in the Input Parameter layout grid.

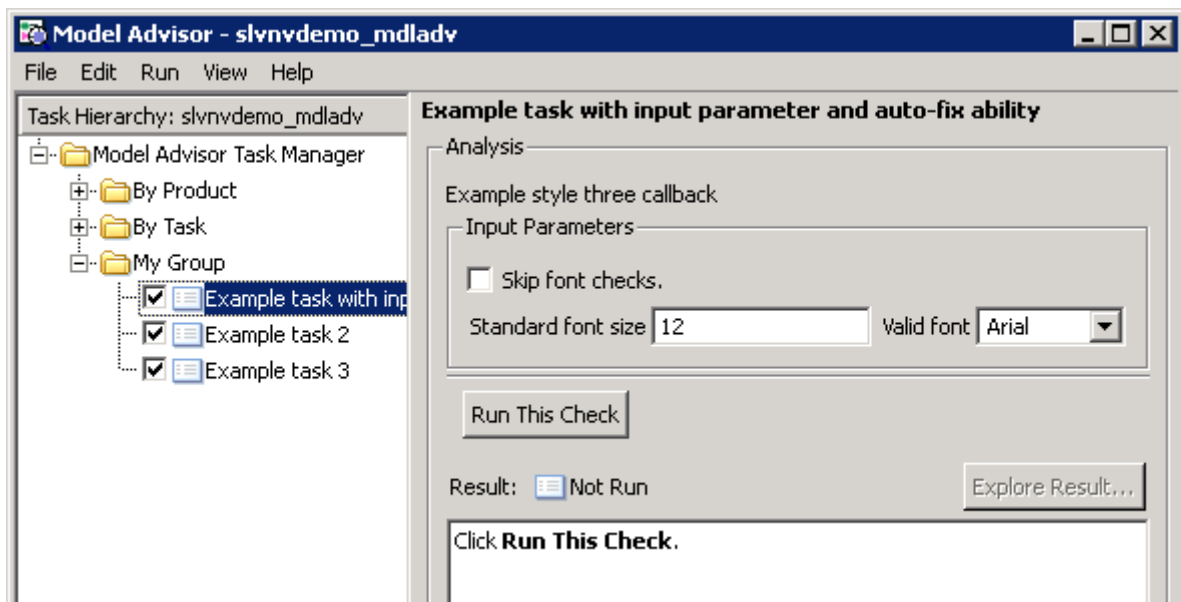
See the `ModelAdvisor.Check` and `ModelAdvisor.InputParameter` class references for information on using these methods.

## Code Example: Input Parameter Definition

The following is an example of defining input parameters to add to a custom check. You must include input parameter definitions inside a custom check definition (see “Code Example: Check Definition Function” on page 6-23). 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 an **Input Parameters** box in the right pane.



## Defining Check List Views

### In this section...

“About List Views” on page 6-31

“Properties of List Views” on page 6-31

“Code Example: List View Definition” on page 6-32

### About List Views

List views allow you to provide the information to populate the Model Advisor Result Explorer window, and adds the **Explore Result** button to a custom check in the Model Advisor window. You define list views using the `ModelAdvisor.ListViewParameter` class inside a custom check function (see “Defining Custom Checks” on page 6-17). You must define one instance of this class for each list view that you want to add to a Model Advisor Result Explorer. See “Batch-Fixing Warnings or Failures” in the Simulink documentation for information on using the dialog box.

### Properties of List Views

The following table describes the properties of the `ModelAdvisor.ListViewParameter` class:

Property	Data Type	Default Value	Description
Name	String	' ' (null string)	Entry in the <b>Show</b> drop-down menu in the Model Advisor Result Explorer.
Data	Array of Simulink objects	[] (empty array)	The objects shown in the Model Advisor Result Explorer.
Attributes	Cell array	{ } (empty cell array)	The attributes to show in the center pane of the Model Advisor Result Explorer.

### Code Example: List View Definition

The following is an example of defining list views. You must make the **Explore Result** button visible using the `ModelAdvisor.Check ListViewVisible` property inside a custom check function, and include list view definitions inside a check callback function (see “Detailed Check Callback Function” on page 6-11).

The following code, when included in a custom check function, adds the **Explore Result** button to the check in the Model Advisor window.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
% add 'Explore Result' button
rec.ListViewVisible = true;
```

The following code, when included in a check callback function, provides the information to populate the Model Advisor Result Explorer window.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object');
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

## Defining Check Actions

### In this section...

“About Actions” on page 6-33

“Properties of Actions” on page 6-33

“Action Callback Function” on page 6-34

“Code Example: Action Definition” on page 6-34

“Code Example: Action Callback Function” on page 6-35

### About Actions

Actions allow you to specify an action to take for a Model Advisor check. When you define an action, the Model Advisor window includes an Action box below the Analysis box. You define actions using the `ModelAdvisor.Action` class inside a custom check function (see “Defining Custom Checks” on page 6-17). You must define

- One instance of this class for each action that you want to take.
- One action callback function for each action (see “Action Callback Function” on page 6-34).

---

**Note** Each check can contain only one action.

---

### Properties of Actions

The following table describes the properties of the `ModelAdvisor.Action` class:

Property	Data Type	Default Value	Description
Name	String	'' (null string)	The action button label. This field is required.
Description	String	'' (null string)	The message displayed in the Action box.

## Action Callback Function

An action callback function specifies the actions 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 <code>ModelAdvisor.Task</code> 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.

See “Code Example: Action Callback Function” on page 6-35 for an example of an action callback function.

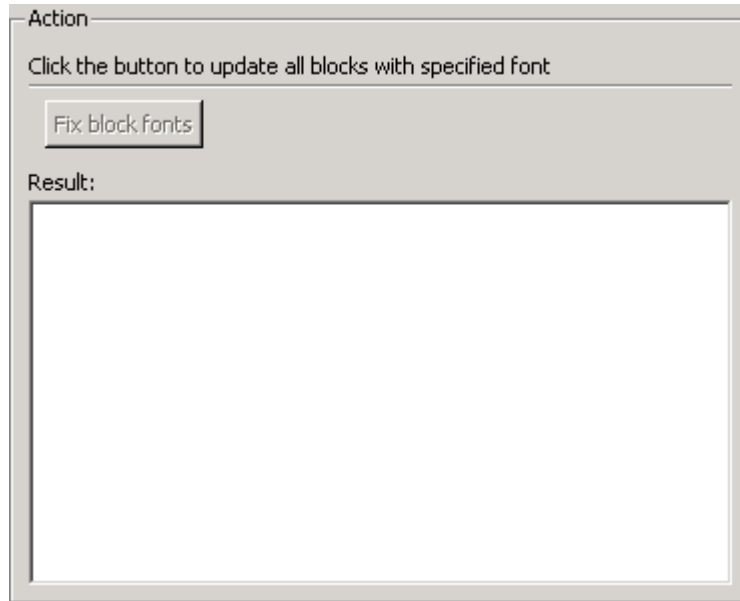
## Code Example: Action Definition

The following is an example of defining actions to do within a custom check. You must include action definitions inside a check definition function (see “Code Example: Check Definition Function” on page 6-23). The following code, when included in a check definition function, provides the information to populate the Action box in the Model Advisor window.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
% 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';
rec.setAction(myAction);
```

The Model Advisor displays an Action box in the right pane.





### Code Example: Action Callback Function

The following is an example of an action callback function that updates all of the blocks in the model with the font specified in the Input Parameter defined in “Code Example: Input Parameter Definition” on page 6-29.

```
function result = sampleActionCB(taskobj)
mdladvObj = taskobj.MAObj;
system = getfullname(mdladvObj.System);

% get input parameters
inputParams = mdladvObj.getInputParameters;
regularFontSize = inputParams{2}.Value;
regularFontName = inputParams{3}.Value;

% find all blocks inside current system
allBlks = find_system(system);
% block diagram itself 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
fixBlks = setdiff(allBlks, regularBlks);
% fix them one by one
for i=1:length(fixBlks)
    set_param(fixBlks{i},'FontName',regularFontName);
end
% save result
resultText1 = ModelAdvisor.Text([num2str(length(fixBlks)), ...
    ' blocks has been updated with specified font ', regularFontName]);

% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',str2double(regularFontSize));
% look for different font size blocks in the system
fixBlks = setdiff(allBlks, regularBlks);
% fix them one by one
for i=1:length(fixBlks)
    set_param(fixBlks{i},'FontSize',regularFontSize);
end
% save result
resultText2 = ModelAdvisor.Text([num2str(length(fixBlks)), ...
    ' blocks has been updated with specified font size ', regularFontSize]);
result = ModelAdvisor.Paragraph;
result.addItem([resultText1 ModelAdvisor.LineBreak resultText2]);
mdladvObj.setActionEnable(false);
```

## Defining Custom Tasks

### In this section...

“About Custom Tasks” on page 6-37

“Properties of Custom Tasks” on page 6-37

“Defining Where Tasks Appear” on page 6-40

“Code Example: Task Definition Function” on page 6-40

### About Custom Tasks

Custom tasks provide a method for adding checks to the Model Advisor tree. You define custom tasks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Task` class. You must define one instance of this class for each custom task that you want to add to the Model Advisor, and register the custom task as described in “Registering Custom Checks, Tasks, and Groups” on page 6-7. The sections that follow describe how to define custom tasks.

### Properties of Custom Tasks

The following table describes the properties of the `ModelAdvisor.Task` class:

Property	Data Type	Default Value	Description
<code>DisplayName</code>	String	'' (null string)	Name of the task as it should appear in Model Advisor.

Property	Data Type	Default Value	Description
ID	String	' ' (null string)	<p>Permanent, unique identifier for the task. The Model Advisor automatically assigns a string to ID if you do not specify it.</p> <hr/> <p><b>Caution</b></p> <ul style="list-style-type: none"> <li>• The value of ID must remain constant.</li> <li>• The Model Advisor generates an error if ID is not unique.</li> <li>• Groups should refer to tasks by ID.</li> </ul> <hr/>
Description	String	' ' (null string)	Description of the task, which Model Advisor displays in the Analysis box.
Visible	Boolean	true	<p>Show or hide task?</p> <ul style="list-style-type: none"> <li>• true = Display task in Model Advisor</li> <li>• false = Hide task</li> </ul>
Enable	Boolean	true	<p>Can user enable and disable task?</p> <ul style="list-style-type: none"> <li>• true = Display check box control for task</li> <li>• false = Hide check box control for task</li> </ul>

Property	Data Type	Default Value	Description
Value	Boolean	true	Initial status: <ul style="list-style-type: none"> <li>• true = Task is enabled</li> <li>• false = Task is disabled</li> </ul>
LicenseName	Cell array	{ } (empty cell array)	Cell array of names of product licenses required to enable the check. Model Advisor does not display the check if license requirements are not met.If ModelAdvisor.CheckLicenseName is specified, the Model Advisor displays the check when the union of both properties is true. <p><b>Tip</b> To find the correct text for license strings, type <code>help license</code> at the MATLAB command line.</p>
MAObj	Simulink.ModelAdvisor object	Handle to Simulink.ModelAdvisor object	The Model Advisor object you are working on.

### How Visible, Enable, and Value Properties Interact for Tasks

These properties interact the same way for tasks as for checks (see “How Visible, Enable, and Value Properties Interact” on page 6-21).

### Defining Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor tree using the following guidelines.

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class. See “Defining Custom Groups” on page 6-41.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. See “Defining Custom Groups” on page 6-41.

### Code Example: Task Definition Function

The following is an example of a task definition function. This function defines three tasks. See “Code Example: Group Definition” on page 6-43 for an example of placing these tasks into a custom group.

```
function defineTaskAdvisor
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);
```

## Defining Custom Groups

### In this section...

“About Custom Groups” on page 6-41

“Defining Where Custom Groups Appear” on page 6-41

“Properties of Model Advisor Groups” on page 6-42

“Code Example: Group Definition” on page 6-43

### About Custom Groups

Groups are used to consolidate checks in the Model Advisor by functionality or usage. You define custom groups in

- One or more functions that specify the properties of each instance of the `ModelAdvisor.FactoryGroup` class.
- One or more task definition functions that specify the properties of each instance of the `ModelAdvisor.Group` class. See “Defining Custom Tasks” on page 6-37 for more information about task definition functions.

You must define one instance of the group classes for each grouping that you want to add to the Model Advisor, and register the custom group as described in “Registering Custom Checks, Tasks, and Groups” on page 6-7. The sections that follow describe how to define custom groups.

### Defining Where Custom Groups Appear

You can specify where custom groups are placed within the Model Advisor tree using the following guidelines:

- To define a new group in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To define a new group in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

---

**Note** To define a new group in the **By Product** folder, you must use the `ModelAdvisor.Root.publish` method within a custom check. See “Defining Where Custom Checks Appear” on page 6-22 for more information.

---

## Properties of Model Advisor Groups

The following table describes the properties of the `ModelAdvisor.Group` and `ModelAdvisor.FactoryGroup` classes:

Property	Data Type	Default Value	Description
DisplayName	String	'' (null string)	Name of the group as it should appear in Model Advisor.
ID	String	You must provide ID	<p>Permanent, unique identifier for the group that you must specify.</p> <hr/> <p><b>Caution</b></p> <ul style="list-style-type: none"> <li>• The value of ID must remain constant.</li> <li>• The Model Advisor generates an error if ID is not unique.</li> <li>• Groups should refer to other groups by ID.</li> </ul> <hr/>



Property	Data Type	Default Value	Description
Description	String	' ' (null string)	Description of the group, which Model Advisor displays in the right pane when you view details about the group.
MAObj	Simulink. ModelAdvisor object	Handle to Simulink. ModelAdvisor object	The Model Advisor object you are working on.

## Code Example: Group Definition

The following is an example of a group definition that places the tasks defined in “Code Example: Task Definition Function” on page 6-40 inside a folder called **My Group** under the **Model Advisor Task Manager** folder. This group definition is included in the task definition function.

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.DisplayName='My Group';
MAG.addTask(MAT1);
MAG.addTask(MAT2);
MAG.addTask(MAT3);
mdladvRoot.publish(MAG); % publish under Model Advisor Task Manager
```

The following is an example of a factory group definition function that places the checks defined in “Code Example: Check Definition Function” on page 6-23 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('com.mathworks.sample.Check1');
```

```
rec.addCheck('com.mathworks.sample.Check2');  
rec.addCheck('com.mathworks.sample.Check3');  
mdladvRoot.publish(rec); % publish inside By Task
```

## Defining a Process Callback Function

In this section...
“About Process Callback Functions” on page 6-45
“Process Callback Function Arguments” on page 6-45
“Code Example: Process Callback Function” on page 6-46

### About Process Callback Functions

The process callback function is an optional function that lets you modify the appearance of checks and tasks in the Model Advisor, and process check results at run time. The process callback function specifies actions to be performed at different stages of Model Advisor execution:

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

If you create a process callback function, you must register it as described in “Registering Custom Checks, Tasks, and Groups” on page 6-7. The sections that follow provide more information about defining your own process callback functions.

### Process Callback Function Arguments

The process callback function takes the following arguments:

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

### Code Example: Process Callback Function

Here is an example of a process callback function that specifies actions in the `configure` stage to enable only the custom checks assigned to the Demo group in “Code Example: Check Definition Function” on page 6-23. In the `process_results` stage, this function displays an informative dialog box for checks that do not pass.

```
function [checkCellArray taskCellArray] = ...
    ModelAdvisorProcessFunction(stage, system, checkCellArray, taskCellArray)
switch stage
case 'configure'
    for i=1:length(checkCellArray)
```

```
% hidden all checks that do not belong to Demo group
if ~(strcmp(checkCellArray{i}.Group, 'Demo'))
    checkCellArray{i}.Visible = false;
    checkCellArray{i}.Value = false;
end
end
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'))
        if isempty(strfind(checkCellArray{i}.Result, 'Passed'))
            disp('Example message from Model Advisor Process callback.');
        end
    end
end
end
end
```

## Formatting Model Advisor Outputs

In this section...
“What Is the Model Advisor Formatting API?” on page 6-48
“Formatting Text” on page 6-48
“Formatting Lists” on page 6-49
“Formatting Tables” on page 6-50
“Formatting Paragraphs” on page 6-50
“Code Example: Model Advisor Formatted Output” on page 6-51

### What Is the Model Advisor Formatting API?

You can use the Model Advisor Formatting API to produce formatted outputs in Model Advisor. The following constructors of the `ModelAdvisor` class provide the ability to format the output. For more information on each constructor and associated methods, click the link in the Constructor column.

Constructor	Description
<code>ModelAdvisor.Text</code>	Formats element text.
<code>ModelAdvisor.Paragraph</code>	Combines elements into paragraph format.
<code>ModelAdvisor.List</code>	Creates a list of elements.
<code>ModelAdvisor.LineBreak</code>	Adds a line break between elements.
<code>ModelAdvisor.Table</code>	Creates a table.
<code>ModelAdvisor.Image</code>	Adds an image to the output.

### Formatting Text

Text is the simplest form of output, but you can format text in many different ways. The default text formatting is:

- Empty
- Default color (black)

- Unformatted (that is, not bold, italicized, underlined, linked, subscripted, or superscripted)

To change text formatting, use the `ModelAdvisor.Text` constructor. When you want one type of formatting for all text, you can achieve this using the syntax

```
ModelAdvisor.Text(content, {attributes})
```

When you want multiple types of formatting, you must build the text, as shown in the next example:

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' to ensure uniform appearance of model.');
```

```
result = [t1, t2, t3, t4, t5];
```

You can add ASCII and Extended ASCII characters using the MATLAB `char` command. See the `ModelAdvisor.Text` constructor reference for more information.

## Formatting Lists

You can create two types of lists, numbered and bulleted. The default list formatting is bulleted. Use the `ModelAdvisor.List` constructor to create and format lists (see `ModelAdvisor.List`). You can create lists with indented subsections, formatted as either numbered or bulleted, as shown in the next example:

```
subList = ModelAdvisor.List();
subList.setType('numbered');
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass', 'bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass', 'bold'}));
```

```
topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1', {'keyword', 'bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2', {'keyword', 'bold'}), subList]);
```

## Formatting Tables

The default table formatting is:

- Default color (black)
- Left justified
- Bold title, row, and column headings

You can change table formatting using the `ModelAdvisor.Table` constructor (see `ModelAdvisor.Table`). The following example code creates a subtable within a table, as shown in the figure:

```
table1 = ModelAdvisor.Table(1,1);  
table2 = ModelAdvisor.Table(2,3);  
table2.setHeading('Table 2');  
table2.setHeadingAlign('center');  
table2.setColHeading(1, 'Header 1');  
table2.setColHeading(2, 'Header 2');  
table2.setColHeading(3, 'Header 3');  
table1.setHeading('Table 1');  
table1.setEntry(1,1,table2);
```

Table 1		
Table 2		
Header 1	Header 2	Header 3

## Formatting Paragraphs

Paragraphs need to be handled explicitly because most markup languages do not support line breaks. The default paragraph formatting is:

- Empty
- Default color (black)



- Unformatted, that is, not bold, italicized, underlined, linked, subscripted, or superscripted
- Aligned left

If you want to change paragraph formatting, use the `ModelAdvisor.Paragraph` constructor (see `ModelAdvisor.Paragraph`).

## Code Example: Model Advisor Formatted Output

The following is the example from “Simple Check Callback Function” on page 6-10, 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
```



# Function Reference

---

Requirements Management  
Interface (p. 7-2)

Model Coverage (p. 7-3)

Model Advisor Customization API  
(p. 7-4)

Model Advisor Formatting API  
(p. 7-5)

Access Requirements Management  
Interface

Configure and execute model  
coverage tests; store and report test  
results

Customize the Model Advisor

Format Model Advisor outputs

## **Requirements Management Interface**

<code>rmi</code>	Requirements Management Interface API
<code>rminav</code>	Start Requirements Management Interface

## Model Coverage

<code>conditioninfo</code>	Display condition coverage information for model object
<code>cv.cvdatagroup</code>	Group together multiple cvdata objects
<code>cv.cvtestgroup</code>	Group together multiple cvtest objects
<code>cvexit</code>	Exit model coverage environment
<code>cvhtml</code>	Produce HTML report from model coverage objects in memory
<code>cvload</code>	Load coverage tests and results stored in file
<code>cvmodelview</code>	Display model coverage results with model coloring
<code>cvsave</code>	Save coverage tests and results to file
<code>cvsim</code>	Simulate and return model coverage results for test objects
<code>cvsimref</code>	Simulate and return model coverage results for referenced models
<code>cvtest</code>	Create model coverage test specification object
<code>decisioninfo</code>	Display decision coverage information for model object
<code>mcdcinfo</code>	Display modified condition/decision coverage information for model object
<code>sigrangeinfo</code>	Display signal range coverage information for model object
<code>tableinfo</code>	Display lookup table coverage information for model object

## Model Advisor Customization API

<code>ModelAdvisor.Action</code>	Add actions to custom checks
<code>ModelAdvisor.Check</code>	Create custom checks
<code>ModelAdvisor.FactoryGroup</code>	Define group in <b>By Task</b> folder
<code>ModelAdvisor.Group</code>	Define custom groups
<code>ModelAdvisor.InputParameter</code>	Add input parameters to custom checks
<code>ModelAdvisor.ListViewParameter</code>	Add list view parameters to custom checks
<code>ModelAdvisor.Root</code>	Identify root node
<code>ModelAdvisor.Task</code>	Define custom tasks

## Model Advisor Formatting API

<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 class
<code>ModelAdvisor.Text</code>	Create Model Advisor text output





# Functions — Alphabetical List

---

# conditioninfo

---

**Purpose** Display condition coverage information for model object

**Syntax**

```
coverage = conditioninfo(cvdo, object)
coverage = conditioninfo(cvdo, object, ignore_descendants)
[coverage, description] = conditioninfo(cvdo, object)
```

**Description** `coverage = conditioninfo(cvdo, object)` returns condition coverage results from the cvdata object `cvdo` for the model component specified by `object`. See “Specifying a Model Object” on page 8-3 for more information about the `object` argument. The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — the number of condition outcomes satisfied for `object`
- `total_outcomes` — the total number of condition outcomes for `object`

---

**Note** `coverage` is empty if `cvdo` does not contain condition coverage results for `object`.

---

`coverage = conditioninfo(cvdo, object, ignore_descendants)` returns condition coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, description] = conditioninfo(cvdo, object)` returns condition coverage results and textual descriptions of each condition in `object`. `description` is a structure array containing the following fields:

- `text` — string describing a condition or the block port to which it applies
- `trueCnts` — number of times the condition was true in a simulation
- `falseCnts` — number of times the condition was false in a simulation

## Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
BlockPath	Full path to a Simulink model or block
BlockHandle	Handle to a Simulink model or block
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
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

### Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable condition coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.condition = 1;
```

## conditioninfo

---

```
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the condition coverage results for the Logic block (in the Gain subsystem) and determine its percentage of condition outcomes covered.

```
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');  
cov = conditioninfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

### See Also

decisioninfo, mcdcinfo

**Purpose** Group together multiple cvdata objects

**Description** Instances of this class contain a collection of cvdata objects. For more information, see “Extracting Results from cv.cvdatagroup” on page 5-63.

## Property Summary

Name	Description
name	Name of the cv.cvdatagroup object.

## Method Summary

Name	Description
allNames	Get all model names associated with cv.cvdatagroup object.
get	Get cvdata objects.

## Properties

### name

#### Description

Name of the cv.cvdatagroup object.

#### Data Type

string

#### Access

RW

## Methods

### allNames

#### Purpose

Get all model names associated with cv.cvdatagroup object.

#### Syntax

```
allNames
```

#### Description

Returns a cell array of strings identifying all model names associated with a cv.cvdatagroup object.

## cv.cvdatagroup

---

### **get**

#### **Purpose**

Get cvdata objects.

#### **Syntax**

```
get(modelName)
```

#### **Arguments**

modelName

String specifying the name of a model whose cvdata object is to be returned.

#### **Description**

Returns the cvdata object that corresponds to modelName.

### **See Also**

cvsimref

**Purpose** Group together multiple cvtest objects

**Description** Instances of this class contain a collection of cvtest objects. For more information, see “Creating a Test Group with cv.cvtestgroup” on page 5-62.

## Property Summary

Name	Description
name	Name of the cv.cvtestgroup object.

## Method Summary

Name	Description
add	Add cvtest objects.
allNames	Get all model names associated with cv.cvtestgroup object.
get	Get cvtest objects.

## Properties

### name

#### Description

Name of the cv.cvtestgroup object.

#### Data Type

string

#### Access

RW

## Methods

### add

#### Purpose

Add cvtest objects.

#### Syntax

```
add(cvto1, cvto2, ...)
```

## cv.cvtestgroup

---

### Arguments

cvto1

String specifying the name of a `cvtest` object to be added to the `cv.cvtestgroup` object.

cvto2

String specifying the name of another `cvtest` object to be added to the `cv.cvtestgroup` object.

### Description

Adds `cvtest` objects to a `cv.cvtestgroup` object.

### allNames

#### Purpose

Get all model names associated with `cv.cvtestgroup` object.

#### Syntax

`allNames`

#### Description

Returns a cell array of strings that identify all model names associated with a `cv.cvtestgroup` object.

### get

#### Purpose

Get `cvtest` objects.

#### Syntax

`get(modelName)`

#### Arguments

`modelName`

String specifying the name of a model whose `cvtest` object is to be returned.

#### Description

Returns the `cvtest` object that corresponds to `modelName`.

### See Also

`cvsimref`, `cvtest`



**Purpose** Exit model coverage environment

**Syntax** `cvexit`

**Description** `cvexit` exits the model coverage environment. Issuing this command causes the Model Coverage Tool to close the Coverage Display window and remove coloring from a block diagram that displays its model coverage results.

# cvhtml

---

**Purpose** Produce HTML report from model coverage objects in memory

**Syntax**

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

**Description** Use the `cvhtml` command to produce an HTML report from `cv.cvdagroup` or `cvdata` objects you produce when you run a model coverage test in simulation.

---

**Note** The model must be open when using the `cvhtml` command to generate its coverage report.

---

`cvhtml(file, cvdo)` creates an HTML report of the coverage results in the `cvdata` or `cv.cvdagroup` object `cvdo`, which is written to the file `file` in the current MATLAB directory.

`cvhtml(file, cvdo1, cvdo2,...)` creates a combined report of several `cvdata` objects. The results from each object are displayed in a separate column of the HTML report. Each `cvdata` object must correspond to the same root model or subsystem, or the function produces errors.

`cvhtml(file, cvdo1, cvdo2,..., options)` creates a combined report of several `cvdata` objects using the report options specified by the `options` string. The table in “Report Options” on page 8-11 lists available options and their default settings. To enable an option, set it equal to 1 (e.g., `'-hTR=1'`); to disable an option, set it equal to 0 (e.g., `'-bRG=0'`). To specify multiple report options, list individual options in a single `options` string separated by commas or spaces (e.g., `'-hTR=1 -bRG=0 -scm=0'`).

`cvhtml(file, cvdo1, cvdo2,..., options, detail)` creates a combined report of several `cvdata` objects and specifies the detail level of the report with the value of `detail`, an integer between 0 and 3.

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

## Report Options

The following table summarizes the report options that you can specify using cvhtml. See “Settings” on page 5-19 under the “Report Tab” section in the *Simulink Verification and Validation User’s Guide* for more information.

Option	Description	Default Setting
-aTS	Include each test in the model summary	on
-bRG	Produce bar graphs in the model summary	on
-bTC	Use two color bar graphs (red, blue)	off
-hTR	Display hit/count ratio in the model summary	off
-nFC	Do not report fully covered model objects	off
-scm	Include cyclomatic complexity numbers in summary	on
-bcm	Include cyclomatic complexity numbers in block details	on

# cvload

---

**Purpose** Load coverage tests and results stored in file

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

**Description** The `cvload` command loads into memory the coverage tests and results stored in a file by the `cvsave` command.

`[cvtos, cvdos] = cvload(filename)` loads the tests and data stored in the text file `filename.cvt`. The `cvtest` objects that are successfully loaded are returned in `cvtos`, a cell array of `cvtest` objects. The `cvdata` objects that are successfully loaded are returned in `cvdos`, a cell array of `cvdata` objects. `cvdos` has the same size as `cvtos`, but can contain empty elements if a particular test has no results.

`[cvtos, cvdos] = cvload(filename, restoretotal)` restores the cumulative results from prior runs if `restoretotal` is 1. If `restoretotal` is unspecified or 0, the model's cumulative results are cleared.

## **cvload Special Considerations**

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

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

**Purpose** Display model coverage results with model coloring

**Syntax** `cvmodelview(cvdo)`

**Description** `cvmodelview(cvdo)` displays coverage results from the `cvdata` object `cvdo` by coloring the Simulink model (see “Displaying Model Coverage with Model Coloring” on page 5-48).

**Example** The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
data = cvsim(testObj)
```

Afterward, issue the following command to display the model coverage results by coloring the block diagram.

```
cvmodelview(data)
```

## **cvsave**

---

### **Purpose**

Save coverage tests and results to file

### **Syntax**

```
cvsave(filename, model)
cvsave(filename, cvto1, cvto2, ...)
cvsave(filename, cvdo1, cvdo2, ...)
```

### **Description**

Save the coverage tests and results from simulations to a file with the function `cvsave`.

`cvsave(filename, model)` saves all the tests (`cvtest` objects) and results (`cvdata` objects) in memory related to the model `model` in the text file `filename.cvt`.

`cvsave(filename, cvto1, cvto2, ...)` saves the tests in the `cvtest` objects `cvto1`, `cvto2`, ... in the text file `filename.cvt`. Information about the referenced models is also saved.

`cvsave(filename, cvdo1, cvdo2, ...)` saves the tests, test results, and referenced models' structure for `cvdata` objects `cvdo1`, `cvdo2`, ... to the text file `filename.cvt`.

---

<b>Purpose</b>	Simulate and return model coverage results for test objects
<b>Syntax</b>	<pre>cvdo = cvsim(cvto) [cvdo,t,x,y] = cvsim(cvto) [cvdo,t,x,y] = cvsim(cvto, timespan, options) [cvdo,t,x,y] = cvsim(cvto, label, setupcmd) [cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)</pre>
<b>Description</b>	You simulate a test specification object (a <code>cvtest</code> object) with the <code>cvsim</code> command.

---

**Note** You do not have to enable model coverage reporting for the model to use the `cvsim` command.

---

`cvdo = cvsim(cvto)` executes the `cvtest` object `cvto` by starting a simulation run for the corresponding model. The results are returned in the `cvdata` object `cvdo`. But when recording coverage for multiple models in a hierarchy, `cvsim` returns its results in a `cv.cvdagroup` 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. Refer to the `sim` command in the Simulink documentation for descriptions of the parameters `t`, `x`, and `y`.

`[cvdo,t,x,y] = cvsim(cvto, timespan, options)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation, and overrides default simulation values with the values for `timespan` and `options`. Refer to the `sim` command in the *Simulink Reference* for descriptions of the parameters `t`, `x`, `y`, `timespan`, and `options`.

`[cvdo,t,x,y] = cvsim(cvto, label, setupcmd)` creates the `cvtest` object `cvto` and simulates it in one command. The arguments `label` and `setupcmd` are passed directly to the `cvtest` command, which creates the `cvtest` object `cvto`.

## **cvsim**

---

`[cvdo1, cvdo2, ...] = cvsim(cvto1, cvto2, ...)` executes the `cvtest` objects `cvto1`, `cvto2`, ... and returns the results in the set of `cvdata` objects `cvdo1`, `cvdo2`, ....



**Purpose** Simulate and return model coverage results for referenced models

**Syntax**

```
cvdg = cvsimref(topmodelName)
cvdg = cvsimref(topmodelName, cvtg)
[cvdg,t,x,y] = cvsimref(topmodelName, cvtg)
[cvdg,t,x,y] = cvsimref(topmodelName, cvtg, timespan,
    options)
[cvdg1, cvdg2, ...] = cvsimref(topmodelName, cvtg1, cvtg2,
    ...)
```

**Description** Use the `cvsimref` function to record coverage for referenced models in a hierarchy. For more information, see “Using Model Coverage Commands for Referenced Models” on page 5-59.

---

**Note** You do not have to enable model coverage reporting for any of the models in a model hierarchy to use the `cvsimref` command.

---

`cvdg = cvsimref(topmodelName)` simulates the top model that `topmodelName` specifies, collects model coverage data, and returns the results in the `cv.cvdagroup` object `cvdg`.

`cvdg = cvsimref(topmodelName, cvtg)` executes the `cv.cvtestgroup` object `cvtg` by starting a simulation run for the corresponding top model, `topmodelName`. The results are returned in the `cv.cvdagroup` object `cvdg`.

`[cvdg,t,x,y] = cvsimref(topmodelName, cvtg)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation. Refer to the `sim` function in the *Simulink Reference* for descriptions of the parameters `t`, `x`, and `y`.

`[cvdg,t,x,y] = cvsimref(topmodelName, cvtg, timespan, options)` returns the time vector `t`, matrix of state values `x`, and matrix of output values `y` from the simulation, and overrides default simulation values with the values for `timespan` and `options`. Refer to the `sim`

## cvsimref

---

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

```
[cvdg1, cvdg2, ...] = cvsimref(topModelName, cvtg1, cvtg2,  
...) executes the cv.cvtestgroup objects cvtg1, cvtg2, ... and  
returns the results in the set of cv.cvdgroup objects cdvg1, cdvg2,  
....
```

### See Also

`cv.cvdgroup`, `cv.cvtestgroup`

**Purpose** Create model coverage test specification object

**Syntax**

```

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

**Description** The cvtest command creates a test specification object, that you simulate with the cvsim command.

cvto = cvtest(root) creates a test object with the handle cvto. root is the name of, or a handle to, a Simulink model or a subsystem of a model. Only the specified model or subsystem and its descendants are subject to model coverage testing.

cvto = cvtest(root, label) creates a test object with the label label, which is used for reporting results.

cvto = cvtest(root, label, setupcmd) creates a test object with the setup command setupcmd and labels it with label. The setup command is executed in the base MATLAB workspace just prior to running the instrumented simulation. This command is useful for loading data prior to a test.

A test object has the following structure.

Field	Description
id	Read-only internal data-dictionary ID
modelcov	Read-only internal data-dictionary ID
rootPath	Name of the system or subsystem instrumented for analysis
label	String used when reporting results
setupCmd	Command executed in the base workspace just prior to simulation

Field	Description
<code>settings.condition</code>	Set to 1 if condition coverage is desired
<code>settings.decision</code>	Set to 1 if decision coverage is desired
<code>settings.mcdc</code>	Set to 1 if MC/DC coverage is desired
<code>settings.sigrange</code>	Set to 1 if signal range coverage is desired
<code>settings.tableExec</code>	Set to 1 if lookup table coverage is desired
<code>modelRefSettings.enable</code>	String specifying one of the following values: <ul style="list-style-type: none"> <li>• <code>Off</code> — Disables coverage for all referenced models</li> <li>• <code>all</code> — Enables coverage for all referenced models</li> <li>• <code>filtered</code> — Enables coverage only for referenced models not listed in the <code>excludedModels</code> subfield</li> </ul>
<code>modelRefSettings.exclude-TopModel</code>	Set to 1 if excluding coverage for the top model is desired
<code>modelRefSettings.excluded-Models</code>	String specifying a comma-separated list of referenced models for which coverage is disabled

## Purpose

Display decision coverage information for model object

## Syntax

```
coverage = decisioninfo(cvdo, object)
coverage = decisioninfo(cvdo, object, ignore_descendants)
[coverage, description] = decisioninfo(cvdo, object)
```

## Description

`coverage = decisioninfo(cvdo, object)` returns decision coverage results from the cvdata object `cvdo` for the model component specified by `object`. See “Specifying a Model Object” on page 8-22 for more information about the `object` argument. The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — the number of decision outcomes satisfied for `object`
- `total_outcomes` — the total number of decision outcomes for `object`

---

**Note** `coverage` is empty if `cvdo` does not contain decision coverage results for `object`.

---

`coverage = decisioninfo(cvdo, object, ignore_descendants)` returns decision coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, description] = decisioninfo(cvdo, object)` returns decision coverage results and textual descriptions of decision points associated with `object`. `description` is a structure array containing the following fields:

- `decision.text` — string describing a decision point, e.g., 'U > LL'
- `decision.outcome.text` — string describing a decision outcome, i.e., 'true' or 'false'
- `decision.outcome.executionCount` — number of times a decision outcome occurred in a simulation

## Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
BlockPath	Full path to a Simulink model or block
BlockHandle	Handle to a Simulink model or block
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
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

## Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable decision coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
testObj.settings.decision = 1;
```

```
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the decision coverage results for the Saturation block and determine its percentage of decision outcomes covered.

```
blk_handle = get_param([mdl, '/Saturation'], 'Handle');  
cov = decisioninfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

## See Also

conditioninfo, mcdcinfo

# mcdcinfo

---

**Purpose** Display modified condition/decision coverage information for model object

**Syntax**

```
coverage = mcdcinfo(cvdo, object)
coverage = mcdcinfo(cvdo, object, ignore_descendants)
[coverage, description] = mcdcinfo(cvdo, object)
```

**Description** `coverage = mcdcinfo(cvdo, object)` returns modified condition/decision coverage results from the cvdata object `cvdo` for the model component specified by `object`. See “Specifying a Model Object” on page 8-25 for more information about the `object` argument. The value of `coverage` is a two-element vector of form `[covered_outcomes total_outcomes]`, the elements of which are defined as follows:

- `covered_outcomes` — the number of condition/decision outcomes satisfied for `object`
- `total_outcomes` — the total number of condition/decision outcomes for `object`

---

**Note** `coverage` is empty if `cvdo` does not contain modified condition/decision coverage results for `object`.

---

`coverage = mcdcinfo(cvdo, object, ignore_descendants)` returns modified condition/decision coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, description] = mcdcinfo(cvdo, object)` returns modified condition/decision coverage results and textual descriptions of each condition/decision in `object`. `description` is a structure array containing the following fields:

- `text` — string denoting whether the condition/decision is associated with a block output or Stateflow transition



- `condition.text` — string describing a condition/decision or the block port to which it applies
- `condition.achieved` — logical array indicating whether a condition case has been fully covered
- `condition.trueRslt` — string representing a condition case expression that produces a true result
- `condition.falseRslt` — string representing a condition case expression that produces a false result

See “MC/DC Analysis Table” on page 5-31 for more information about the data contained in these fields.

### Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
<code>BlockPath</code>	Full path to a Simulink model or block
<code>BlockHandle</code>	Handle to a Simulink model or block
<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 block and the ID of an object contained in that chart

Object Specification	Description
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

## Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable modified condition/decision coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
testObj.settings.mcdc = 1;  
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the modified condition/decision coverage results for the Logic block (in the Gain subsystem) and determine its percentage of condition/decision outcomes covered.

```
blk_handle = get_param([mdl, '/Gain/Logic'], 'Handle');  
cov = mcdcinfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

## See Also

`conditioninfo`, `decisioninfo`

**Purpose** Add actions to custom checks

**Class Description** You define actions to take when Model Advisor checks do not pass. Users access actions by clicking the action button that you define in the Model Advisor window.

**Syntax**

```
action_obj = ModelAdvisor.Action
```

**Arguments**

*action\_obj*  
A variable representing the action object you create.

**Method Summary**

Name	Description
“setCallbackFcn” on page 8-27	Specify action callback function

**Methods**

**setCallbackFcn**

**Purpose**

Specify action callback function

**Syntax**

```
action_obj.setCallbackFcn(@handle)
```

**Arguments**

*action\_obj*  
A variable representing the action object.

*handle*  
A handle to an action callback function.

**Description**

The setCallbackFcn method specifies the callback function to use with the action object.

# ModelAdvisor.Action

---

## See Also

“Action Callback Function” on page 6-34 in the Simulink® Verification and Validation™ User’s Guide on page 1

## 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.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
% 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';
rec.setAction(myAction);
```

## See Also

- `ModelAdvisor.Check`
- “Defining Check Actions” on page 6-33 in the Simulink® Verification and Validation™ User’s Guide on page 1
- “setActionEnable” in the Simulink User’s Guide.

## Purpose

Create custom checks

## Class Description

The `ModelAdvisor.Check` class creates a Model Advisor check object. All checks must have an associated `ModelAdvisor.Task` or `ModelAdvisor.Root` object to appear 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** appears in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

## Syntax

```
check_obj = ModelAdvisor.Check(check_ID)
```

## Arguments

*check\_obj*

A variable representing the check object you create.

*check\_ID*

A string that uniquely identifies the check. The value of *check\_ID* must remain constant.

## Method Summary

Name	Description
“getID” on page 8-30	Returns the check ID
“setAction” on page 8-31	Specify action
“setCallbackFcn” on page 8-32	Specify callback function
“setInputParameters” on page 8-33	Specify input parameters
“setInputParametersLayoutGrid” on page 8-34	Specify layout grid for input parameters

# ModelAdvisor.Check

---

## Methods

### getID

#### Purpose

Returns the check ID

#### Syntax

```
id = check_obj.getID
```

#### Arguments

*check\_obj*

A variable representing the check object.

#### Return Values

*id*

A unique string identifying the check.

#### Description

The `getID` method returns a string that uniquely identifies the check. You create this unique identifier when you create the check. This is the equivalent of the `ModelAdvisor.Check ID` property.

#### See Also

“Properties of Custom Checks” on page 6-17 in the Simulink® Verification and Validation™ User’s Guide on page 1

## **setAction**

### **Purpose**

Specify action

### **Syntax**

```
check_obj.setAction(action_obj)
```

### **Arguments**

*check\_obj*

A variable representing the check object.

*action\_obj*

The ModelAdvisor.Action object to use in the check.

### **Description**

The setAction method identifies the action you want to use in a check.

### **See Also**

ModelAdvisor.Action

## **setCallbackFcn**

### **Purpose**

Specify callback function

### **Syntax**

```
check_obj.setCallbackFcn(@handle, context, style)
```

### **Arguments**

*check\_obj*

A variable representing the check object.

*handle*

A handle to a check callback function.

*context*

Context for checking the model or subsystem.

- None — No special requirements.
- PostCompile — The model must be compiled.

*style*

Type of callback function:

- StyleOne — Simple check callback function
- StyleTwo — Detailed check callback function
- StyleThree — Check callback function with hyperlinked results

### **Description**

Specify the callback function to use with the check object.

### **See Also**

“Creating Callback Functions for Checks” on page 6-10 in the Simulink® Verification and Validation™ User’s Guide on page 1



## **setInputParameters**

### **Purpose**

Specify input parameters

### **Syntax**

```
check_obj.setInputParameters(params)
```

### **Arguments**

*check\_obj*

A variable representing the check object.

*params*

A cell array of `ModelAdvisor.InputParameter` objects.

### **Description**

Specify the input parameters to use in a check object.

### **See Also**

`ModelAdvisor.InputParameter`

## setInputParametersLayoutGrid

### Purpose

Specify layout grid for input parameters

### Syntax

```
check_obj.setInputParametersLayoutGrid([row col])
```

### Arguments

*check\_obj*

A variable representing the check object.

*row*

Total number of rows in the layout grid.

*col*

Total number of columns in the layout grid.

### Description

Specify the layout grid for input parameters in the Model Advisor. Use this method if there are multiple input parameters.

### See Also

- `ModelAdvisor.InputParameter`
- “setColSpan” on page 8-46
- “setRowSpan” on page 8-47

## 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.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

```
%Define a check
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback, 'None', 'StyleThree');

% Define input parameters
rec.setInputParametersLayoutGrid([3 2]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam2.Name = 'Standard font size';
inputParam2 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3 = ModelAdvisor.InputParameter;
rec.setInputParameters({inputParam1,inputParam2,inputParam3});

% define action (fix) operation
myAction = ModelAdvisor.Action;
rec.setAction(myAction);
```

## See Also

- “Defining Custom Checks” on page 6-17 in the Simulink® Verification and Validation™ User’s Guide on page 1
- ModelAdvisor.Action
- ModelAdvisor.InputParameter
- ModelAdvisor.ListViewParameter
- ModelAdvisor.Root
- ModelAdvisor.Task
- “setCheckErrorSeverity” in the Simulink User’s Guide.

# ModelAdvisor.FactoryGroup

---

**Purpose** Define group in **By Task** folder

**Class Description** The `ModelAdvisor.FactoryGroup` class defines a new subfolder to add to the **By Task** folder.

## Syntax

```
fg_obj = ModelAdvisor.FactoryGroup(fg_ID)
```

## Arguments

*fg\_obj*  
A variable representing the factory group object you are creating.

*fg\_ID*  
A unique string that identifies the group. The value of *fg\_ID* must remain constant.

## Method Summary

Name	Description
“addCheck” on page 8-37	Add check to group

## Methods

### addCheck

#### Purpose

Add check to group

#### Syntax

```
fg_obj.addCheck(check_ID)
```

#### Arguments

*fg\_obj*

A variable representing the factory group object.

*check\_ID*

The unique identification string representing the check to add.

#### Description

The addCheck method adds checks, identified by check\_ID, to the folder specified by the group object.

#### See Also

ModelAdvisor.Check

## Example

---

**Note** The following example is a fragment of code from the `sl_customization.m` file for the demo model `slvndemo_md1adv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

# ModelAdvisor.FactoryGroup

---

```
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('com.mathworks.sample.Check1');
rec.addCheck('com.mathworks.sample.Check2');
rec.addCheck('com.mathworks.sample.Check3');
mdladvRoot.publish(rec); % publish inside By Task
```

## See Also

- “Defining Custom Groups” on page 6-41 in the Simulink® Verification and Validation™ User’s Guide on page 1
- ModelAdvisor.Check
- ModelAdvisor.Root

## Purpose

Define custom groups

## Class Description

The `ModelAdvisor.Group` class defines a group that appears as a folder in the Model Advisor tree. Use groups to consolidate checks by functionality or usage.

## Syntax

```
group_obj = ModelAdvisor.Group(group_ID)
```

## Arguments

*group\_obj*

A variable representing the group object you create.

*group\_ID*

A string that uniquely identifies the group. The value of *group\_ID* must remain constant.

## Method Summary

Name	Description
“addGroup” on page 8-40	Add subgroup to group
“addTask” on page 8-41	Add task to group

# ModelAdvisor.Group

---

## Methods

### **addGroup**

#### **Purpose**

Add subgroup to group

#### **Syntax**

```
group_obj.addGroup(child_obj)
```

#### **Arguments**

*group\_obj*

A variable representing the group object.

*child\_obj*

The unique identifying string that represents the ModelAdvisor.Group object of the subgroup.

#### **Description**

The addGroup method adds a new subfolder, identified by *child\_obj*, to the folder specified by the group object.



## **addTask**

### **Purpose**

Add task to group

### **Syntax**

```
group_obj.addTask(task_obj)
```

### **Arguments**

*group\_obj*

A variable representing the group object.

*task\_obj*

The unique identifying string that represents the ModelAdvisor.Task object.

### **Description**

The addTask method adds a new check, identified by *task\_obj*, to the folder specified by the group object.

### **See Also**

ModelAdvisor.Task

## **Example**

---

**Note** The following example is a fragment of code from the `sl_customization.m` file for the demo model `slvnvdemo_mdadv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');  
MAG.DisplayName='My Group';  
MAG.addTask(MAT1);  
MAG.addTask(MAT2);  
MAG.addTask(MAT3);  
mdladvRoot.publish(MAG); % publish under Model Advisor Task Manager
```

# ModelAdvisor.Group

---

## See Also

- “Defining Custom Groups” on page 6-41 in the Simulink® Verification and Validation™ User’s Guide on page 1
- ModelAdvisor.Task

**Purpose** Include image in Model Advisor output

**Syntax** `object = ModelAdvisor.Image`

**Arguments** `object`  
A variable representing the image object created.

**Description** Specify an image to appear in the Model Advisor output. Model Advisor supports many image formats, including but not limited to JPG, BMP, and GIF.

## Method Summary

Name	Description
“setHyperlink” on page 8-43	Specify hyperlink location
“setImageSource” on page 8-43	Specify image location

## Methods

### setHyperlink

#### Purpose

Specify hyperlink location

#### Syntax

`setHyperlink(url)`

#### Arguments

*url*

A string that specifies the location of the link.

#### Description

Specifies the location of the hyperlink.

### setImageSource

#### Purpose

Specify image location

#### Syntax

`setImageSource(source)`

# ModelAdvisor.Image

---

## Arguments

*source*

A string specifying the location of the image.

## Description

Specifies the location of the image.

## Example

```
report_image = ModelAdvisor.Image;  
report_image = ModelAdvisor.Image;  
report_image.setImageSource(...  
    'http://www.mathworks.com/access/helpdesk/help/techdoc/learn_matlab/p09.gif');
```

## See Also

ModelAdvisor.LineBreak, ModelAdvisor.List,  
ModelAdvisor.Paragraph, ModelAdvisor.Table, ModelAdvisor.Text

**Purpose**

Add input parameters to custom checks

**Class Description**

You specify the input parameters a custom check uses in analyzing the model. Users access input parameters in the Model Advisor window.

**Syntax**

```
input_param = ModelAdvisor.InputParameter
```

**Arguments**

*input\_param*

A variable representing the input parameter object you create.

**Method Summary**

Name	Description
“setColSpan” on page 8-46	Specify columns parameter occupies
“setRowSpan” on page 8-47	Specify rows parameter occupies

# ModelAdvisor.InputParameter

---

## Methods

### setColSpan

#### Purpose

Specify columns parameter occupies

#### Syntax

```
input_param.setColSpan([start_col end_col])
```

#### Arguments

*input\_param*

A variable representing the input parameter object.

*start\_col*

A positive integer representing the first column the input parameter occupies in the layout grid.

*end\_col*

A positive integer representing the last column the input parameter occupies in the layout grid.

#### Description

Specify the number of columns the parameter occupies. Use this method to specify where an input parameter is located in the layout grid when there are multiple input parameters.

#### See Also

“setInputParametersLayoutGrid” on page 8-34

## setRowSpan

### Purpose

Specify rows parameter occupies

### Syntax

```
input_param.setRowSpan([start_row end_row])
```

### Arguments

*input\_param*

A variable representing the input parameter object.

*start\_row*

A positive integer representing the first row the input parameter occupies in the layout grid.

*end\_row*

A positive integer representing the last row the input parameter occupies in the layout grid.

### Description

Specify the number of rows the parameter occupies. Use this method to specify where an input parameter is located in the layout grid when there are multiple input parameters.

### See Also

“setInputParametersLayoutGrid” on page 8-34

## 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.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

# 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

- “Defining Check Input Parameters” on page 6-26 in the Simulink® Verification and Validation™ User’s Guide on page 1
- ModelAdvisor.Check
- “getInputParameters” in the Simulink User’s Guide.



<b>Purpose</b>	Insert line break
<b>Syntax</b>	<code>line_break_obj = ModelAdvisor.LineBreak</code>
<b>Arguments</b>	<code>line_break_obj</code> A variable representing the line break object created.
<b>Description</b>	Use instances of this class to insert line breaks in Model Advisor outputs.
<b>Example</b>	<pre>report_paragraph = ModelAdvisor.Paragraph; report_text = ModelAdvisor.Text('Model Advisor', {'bold'}); report_text.setItalic(true);  report_text2 = ModelAdvisor.Text('Check Report', {'bold'});  line_break = ModelAdvisor.LineBreak;  report_paragraph.addItem([report_text line_break report_text2]);</pre>
<b>See Also</b>	<code>ModelAdvisor.Image</code> , <code>ModelAdvisor.List</code> , <code>ModelAdvisor.Paragraph</code> , <code>ModelAdvisor.Table</code> , <code>ModelAdvisor.Text</code>

# ModelAdvisor.List

---

**Purpose** Create list class

**Syntax** `list = ModelAdvisor.List`

**Arguments** `list`  
A variable representing the list object created.

**Description** Use instances of this class to create list formatted outputs. Creates a new list object.

## Method Summary

Name	Description
“addItem” on page 8-50	Add list item
“setType” on page 8-50	Specify list type

## Methods

### addItem

#### Purpose

Add list item

#### Syntax

`addItem(element)`

#### Arguments

*element*

Element, cell array of elements, or string to be added. When a cell array of elements is added, they form different rows in the list.

#### Description

This method adds items to the list.

### setType

#### Purpose

Specify list type

#### Syntax

`setType(listType)`

## Arguments

### *listType*

String specifying type of list, either numbered or bulleted.

## Description

This method specifies the type of list created.

## Example

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));

topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1',{'keyword','bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2',{'keyword','bold'}), subList]);
```

## See Also

ModelAdvisor.Image, ModelAdvisor.LineBreak,  
ModelAdvisor.Paragraph, ModelAdvisor.Table, ModelAdvisor.Text

# ModelAdvisor.ListViewParameter

---

## Purpose

Add list view parameters to custom checks

## Class Description

The Model Advisor uses list view parameters to populate the Model Advisor Result Explorer. Users access the information provided in list views by clicking the **Explore Result** button in the Model Advisor window.

## Syntax

```
lv_param = ModelAdvisor.ListViewParameter
```

## Arguments

*lv\_param*

A variable representing the list view parameter object you create.

## Example

---

**Note** The following example is a fragment of code from the `sl_customization.m` file for the demo model `slvnvdemo_mdldadv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object');
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

## See Also

- `ModelAdvisor.Check`
- “Batch-Fixing Warnings or Failures” in the Simulink User’s Guide.

- “Defining Check List Views” on page 6-31 in the Simulink® Verification and Validation™ User’s Guide on page 1
- “getListViewParameters” in the Simulink User’s Guide.
- “setListViewParameters” in the Simulink User’s Guide.

# ModelAdvisor.Paragraph

---

**Purpose** Create and format paragraph

**Syntax** `para_obj = ModelAdvisor.Paragraph`

**Arguments** `para_obj`  
A variable representing the paragraph object created.

**Description** Creates and formats a paragraph.

## Method Summary

Name	Description
“setAlign” on page 8-54	Specify paragraph alignment
“addItem” on page 8-55	Add paragraph element

## Methods

### setAlign

#### Purpose

Specify paragraph alignment

#### Syntax

`setAlign(alignment)`

#### Arguments

*alignment*

A string that specifies the alignment of the text. Possible alignments include:

left

Align left

right

Align right

center

Align center

#### Description

Specifies the paragraph alignment. The default is left.

## **addItem**

### **Purpose**

Add paragraph element

### **Syntax**

```
addItem(element)
```

### **Arguments**

*element*

A string, element, or cell array of elements to add to the paragraph.

### **Description**

Adds an element to the paragraph.

## **Example**

```
report_paragraph = ModelAdvisor.Paragraph;
report_paragraph.setAlign('center');

report_text = ModelAdvisor.Text('Magic Square', {'bold'});
report_text.setItalic(true);

report_image = ModelAdvisor.Image;
report_image = ModelAdvisor.Image; report_image.setImageSource(...
    'http://www.mathworks.com/access/helpdesk/help/techdoc/learn_matlab/p09.gif');

line_break = ModelAdvisor.LineBreak;

report_paragraph.addItem([report_text line_break line_break report_image]);
```

## **See Also**

ModelAdvisor.Image, ModelAdvisor.LineBreak, ModelAdvisor.List,  
ModelAdvisor.Table, ModelAdvisor.Text

# ModelAdvisor.Root

---

**Purpose** Identify root node

**Class Description** The ModelAdvisor.Root class returns the root object.

## Syntax

*MAobj* = ModelAdvisor.Root

## Arguments

*MAobj*  
A variable representing the root object.

## Method Summary

Name	Description
“register” on page 8-57	Register object in Model Advisor root
“publish” on page 8-58	Publish object in Model Advisor root



## Methods

### register

#### Purpose

Register object in Model Advisor root

#### Syntax

```
MAobj.register(obj)
```

#### Arguments

*MAobj*

A variable representing the root object.

*obj*

A variable representing the object to register.

#### Description

The register method registers `ModelAdvisor.Check`, `ModelAdvisor.Task`, `ModelAdvisor.Group`, and `ModelAdvisor.FactoryGroup` objects in the Model Advisor memory.

Use the register method to place objects in the Model Advisor memory that you use in other functions. The register method does not place object in the Model Advisor tree.

#### See Also

- `ModelAdvisor.Check`
- `ModelAdvisor.FactoryGroup`
- `ModelAdvisor.Group`
- `ModelAdvisor.Task`

# ModelAdvisor.Root

---

## **publish**

### **Purpose**

Publish object in Model Advisor root

### **Syntax**

```
MAobj.publish(check_obj, location)
```

```
MAobj.publish(group_obj)
```

```
MAobj.publish(fg_obj)
```

### **Arguments**

*MAobj*

A variable representing the root object.

*check\_obj*

A variable representing the ModelAdvisor.Check object to publish.

*location*

A string identifying where the Model Advisor places the check in the Model Advisor tree. The *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'.

*group\_obj*

A variable representing the ModelAdvisor.Group object to publish as a folder in the **Model Advisor Task Manager** folder.

*fg\_obj*

A variable representing the ModelAdvisor.FactoryGroup object to publish as a subfolder in the **By Task** folder.

### **Description**

The publish method places objects in the Model Advisor tree. The location in the tree depends on the type of object. When you publish an object, you do not have to register it.

## See Also

- “Defining Where Custom Checks Appear” on page 6-22 in the Simulink® Verification and Validation™ User’s Guide on page 1
- “Defining Where Tasks Appear” on page 6-40 in the Simulink® Verification and Validation™ User’s Guide on page 1
- “Defining Where Custom Groups Appear” on page 6-41 in the Simulink® Verification and Validation™ User’s Guide on page 1
- ModelAdvisor.Check
- ModelAdvisor.FactoryGroup
- ModelAdvisor.Group

## Example

---

**Note** The following example is a fragment of code from the `sl_customization.m` file for the demo model `slvnvdemo_mdadv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

```
function defineTaskAdvisor
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);
```

# ModelAdvisor.Root

---

```
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
MAT3.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT3);

MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.DisplayName='My Group';
MAG.addTask(MAT1);
MAG.addTask(MAT2);
MAG.addTask(MAT3);
mdladvRoot.publish(MAG); % publish under Model Advisor Task Manager
```

## See Also

- “Registering Custom Checks, Tasks, and Groups” on page 6-7 in the Simulink® Verification and Validation™ User’s Guide on page 1
- ModelAdvisor.Check
- ModelAdvisor.FactoryGroup
- ModelAdvisor.Group
- ModelAdvisor.Task

**Purpose** Create table class

**Syntax** `table = ModelAdvisor.Table(row, column)`

**Arguments**

*table*  
A variable representing the table object created.

*row*  
An integer specifying the number of rows the table contains.

*column*  
An integer specifying the number of columns the table contains.

**Description** Use instances of this class to create and format a table. Specify the number of rows and columns in a table, excluding the table title and table heading row.

## Method Summary

Name	Description
“getEntry” on page 8-62	Get cell contents
“setColHeading” on page 8-62	Specify table column title
“setColHeadingAlign” on page 8-63	Specify column title alignment
“setColWidth” on page 8-63	Specify column widths
“setEntry” on page 8-64	Add cell to table
“setEntryAlign” on page 8-64	Specify cell alignment
“setHeading” on page 8-65	Specify table title
“setHeadingAlign” on page 8-66	Specify table title alignment
“setRowHeading” on page 8-66	Specify table row title
“setRowHeadingAlign” on page 8-67	Specify row title alignment

# ModelAdvisor.Table

---

## Methods

### **getEntry**

#### **Purpose**

Get cell contents

#### **Syntax**

```
content = getEntry(row, column)
```

#### **Arguments**

*row*

An integer specifying the row.

*column*

An integer specifying the column.

*content*

An element object or object array specifying the content of the table entries.

#### **Description**

Gets the contents of a specified cell.

### **setColHeading**

#### **Purpose**

Specify table column title

#### **Syntax**

```
setColHeading(column, heading)
```

#### **Arguments**

*column*

An integer specifying column number.

*heading*

A string, element object, or object array specifying the table column title.

#### **Description**

Specifies the table column title.

## **setColHeadingAlign**

### **Purpose**

Specify column title alignment

### **Syntax**

```
setColHeadingAlign(column, alignment)
```

### **Arguments**

*column*

An integer specifying column number.

*alignment*

A string specifying the cell alignment. Possible values are:

left

Align left

right

Align right

center

Align center

### **Description**

Specifies the alignment of the column headings.

## **setColWidth**

### **Purpose**

Specify column widths

### **Syntax**

```
setColWidth(column, width)
```

### **Arguments**

*column*

An integer specifying column number.

*width*

An integer or array of integers specifying the column widths, relative to the entire table width.

## **Description**

Specifies the table column widths relative to the entire table width. If column widths are [1 2 3], the second column is twice the width of the first column, and the third column is three times the width of the first column. Unspecified columns have a default width of 1. For example:

```
setColWidth(1, 1);  
setColWidth(3, 2);
```

specifies [1 1 2] column widths.

## **setEntry**

### **Purpose**

Add cell to table

### **Syntax**

```
setEntry(row, column, string)
```

```
setEntry(row, column, content)
```

### **Arguments**

*row*

An integer specifying the row.

*column*

An integer specifying the column.

*string*

A string representing the contents of the entry

*content*

An element object or object array specifying the content of the table entries.

## **Description**

Add a cell entry to a table.

## **setEntryAlign**

### **Purpose**

Specify cell alignment



**Syntax**

`setEntryAlign(row, column, alignment)`

**Arguments**

*row*

An integer specifying row number.

*column*

An integer specifying column number.

*alignment*

A string specifying the cell alignment. Possible values are:

`left`

Align left

`right`

Align right

`center`

Align center

**Description**

Specifies the alignment of the table cells.

**setHeading****Purpose**

Specify table title

**Syntax**

`setHeading(title)`

**Arguments**

*title*

A string, element object, or object array, specifying the table title.

**Description**

Specifies the table title, which is the first line of the table.

## **setHeadingAlign**

### **Purpose**

Specify table title alignment

### **Syntax**

`setHeadingAlign(alignment)`

### **Arguments**

*alignment*

A string specifying the cell alignment. Possible values are:

`left`

Align left

`right`

Align right

`center`

Align center

### **Description**

Specifies the alignment of table titles.

## **setRowHeading**

### **Purpose**

Specify table row title

### **Syntax**

`setRowHeading(row, heading)`

### **Arguments**

*row*

An integer specifying row number.

*heading*

A string, element object, or object array specifying the table row title.

### **Description**

Specifies the table row title.

## **setRowHeadingAlign**

### **Purpose**

Specify row title alignment

### **Syntax**

```
setRowHeadingAlign(row, alignment)
```

### **Arguments**

*row*

An integer specifying row number.

*alignment*

A string specifying the cell alignment. Possible values are:

left

Align left

right

Align right

center

Align center

### **Description**

Specifies the alignment of row titles.

## **Example**

```
table1 = ModelAdvisor.Table(1,1);  
table2 = ModelAdvisor.Table(2,3);  
table2.setHeading('Table 2');  
table2.setHeadingAlign('center');  
table2.setColHeading(1, 'Header 1');  
table2.setColHeading(2, 'Header 2');  
table2.setColHeading(3, 'Header 3');  
table1.setHeading('Table 1');  
table1.setEntry(1,1,table2);
```

## **See Also**

ModelAdvisor.Image, ModelAdvisor.LineBreak, ModelAdvisor.List,  
ModelAdvisor.Paragraph, ModelAdvisor.Text

# ModelAdvisor.Task

---

**Purpose** Define custom tasks

**Class Description** The ModelAdvisor.Task class is a wrapper for a check so 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 in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

## Syntax

```
task_obj = ModelAdvisor.Task(task_ID)
```

## Arguments

*task\_obj*  
A variable representing the task object you create.

*task\_ID*  
A string that uniquely identifies the task. The value of *task\_ID* must remain constant. If you do not specify a *task\_ID*, the Model Advisor assigns a random *task\_ID* to the task object.

## Method Summary

Name	Description
“setCheck” on page 8-69	Specify check used in task

## Methods

### setCheck

#### Purpose

Specify check used in task

#### Syntax

```
task_obj.setCheck(check_ID)
```

#### Arguments

*task\_obj*

A variable representing the task object.

*check\_ID*

A unique string that identifies the check to use in the task.

#### Description

The setCheck method specifies the check to use in the task.

#### See Also

ModelAdvisor.Check

## Example

---

**Note** The following example is a fragment of code from the `sl_customization.m` file for the demo model `slvndemo_md1adv`. The example does not execute as shown without the additional content found in the `sl_customization.m` file.

---

See “Demo and Code Example” on page 6-6 in the Simulink® Verification and Validation™ User’s Guide on page 1 for more information.

```
function defineTaskAdvisor
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');
```

# ModelAdvisor.Task

---

```
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);
```

## See Also

- `ModelAdvisor.Check`
- “Defining Custom Tasks” on page 6-37 in the Simulink® Verification and Validation™ User’s Guide on page 1

**Purpose**

Create Model Advisor text output

**Syntax**

```
text = ModelAdvisor.Text(content, {attribute})
```

**Arguments**

*text*

A variable representing the text object created.

*content*

A string specifying the content.

*attribute*

A string specifying the formatting of the content. If no attribute is specified, the output text has default coloring with no formatting implemented. Possible formatting options include:

**bold**

Text is bold.

*italic*

Text is italic.

underlined

Text is underlined.

normal

Text is default color.

pass

Text is green.

warn

Text is yellow.

fail

Text is red.

keyword

Text is blue.

subscript

Text is subscripted.

# ModelAdvisor.Text

---

superscript

Text is superscripted.

retainspacereturn

Text retains spacing and returns.

## Description

Use instances of this constructor to create formatted text for Model Advisor outputs. You can implement the `ModelAdvisor.Text` constructor with or without an *attribute* value. If *content* is empty, empty text is output.

## Method Summary

Name	Description
“setBold” on page 8-72	Bold text
“setColor” on page 8-73	Color text
“setHyperlink” on page 8-73	Hyperlink text
“setItalic” on page 8-74	Italic text
“setRetainSpaceReturn” on page 8-74	Retain spacing and returns in text
“setSubscript” on page 8-74	Subscripted text
“setSuperscript” on page 8-75	Superscripted text
“setUnderlined” on page 8-75	Underlined text

## Methods

### setBold

#### Purpose

Bold text

#### Syntax

`setBold(mode)`

#### Arguments

*mode*

A Boolean value indicating bold formatting of text, either on (true) or off (false).



**Description**

This method makes text bold.

**setColor****Purpose**

Color text

**Syntax**

```
setColor(color)
```

**Arguments**

*color*

An enumerated string specifying the color of the text. Possible formatting options include:

normal

Text is default color.

pass

Text is green.

warn

Text is yellow.

fail

Text is red.

keyword

Text is blue.

**Description**

This method colors text.

**setHyperlink****Purpose**

Hyperlink text

**Syntax**

```
setHyperlink(url)
```

## **Arguments**

*url*

A string that specifies the location of the link.

## **Description**

This method hyperlinks text to the specified URL.

## **setItalic**

### **Purpose**

Italic text

### **Syntax**

`setItalic(mode)`

### **Arguments**

*mode*

A Boolean value indicating italicized formatting of text, either on (true) or off (false).

## **Description**

This method italicizes text.

## **setRetainSpaceReturn**

### **Purpose**

Retain spacing and returns

### **Syntax**

`setRetainSpaceReturn(mode)`

### **Arguments**

*mode*

A Boolean value indicating whether to preserve space and return formatting of text, either on (true) or off (false).

## **Description**

This method retains spaces and carriage returns in text.

## **setSubscript**

### **Purpose**

Subscript text

**Syntax**

`setSubscript(mode)`

**Arguments**

*mode*

A Boolean value indicating subscripted formatting of text, either on (true) or off (false).

**Description**

This method subscripts text.

**setSuperscript****Purpose**

Superscript text

**Syntax**

`setSuperscript(mode)`

**Arguments**

*mode*

A Boolean value indicating superscripted formatting of text, either on (true) or off (false).

**Description**

This method superscripts text.

**setUnderlined****Purpose**

Underline text

**Syntax**

`setUnderlined(mode)`

**Arguments**

*mode*

A Boolean value indicating underlined formatting of text, either on (true) or off (false).

**Description**

This method underlines text.

# ModelAdvisor.Text

---

## Example

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' to ensure uniform appearance of model.');
```

```
result = [t1, t2, t3, t4, t5];
```

## See Also

ModelAdvisor.Image, ModelAdvisor.LineBreak, ModelAdvisor.List,  
ModelAdvisor.Paragraph, ModelAdvisor.Table

**Purpose**

Requirements Management Interface API

**Syntax**

```
rmi setup
rmi register linktypename
rmi unregister linktypename
rmi linktypelist
reqlinks = rmi('createempty')
reqlinks = rmi('get', object)
reqlinks = rmi('get', object, group)
rmi('set', object, reqlinks)
rmi('set', object, reqlinks, group)
rmi('cat', object, reqlinks)
cnt = rmi('count', object)
rmi('clearall', object)
cmdstr = rmi('navcmd', object)
[cmdstr, titlestr] = rmi('navcmd', object)
guidstr = rmi('guidget', object)
object = rmi('guidlookup', model, guidstr)
rmi('highlightModel', object)
rmi('unhighlightModel', object)
rmi('view', object, index)
dialog = rmi('edit', object)
rmi('copyObj', object)
```

**Description**

Use the `rmi` command to interact programmatically with the Requirements Management Interface (RMI).

- “RMI Setup” on page 8-77
- “Requirement Link Management” on page 8-78
- “Navigation and Display Options” on page 8-79

**RMI Setup**

`rmi setup` configures the RMI for use with your computer and installs the interface for use with the Telelogic DOORS software, if needed. See “Configuring the Requirements Management Interface” on page 2-3 for more information about using this command to set up the RMI.

`rmi register linktypename` registers the custom link type specified by the M-file function `linktypename`. See “Linking to Custom Types of Requirements Documents” on page 2-28 for more information.

`rmi unregister linktypename` removes the custom link type specified by the M-file function `linktypename`.

`rmi linktypelist` displays a list of the currently registered link types. The list indicates whether each link type is built-in or custom and provides the path to the M-file function used for its registration.

## Requirement Link Management

`reqlinks = rmi('createempty')` creates an empty instance of the requirement links data structure. See “Requirement Links Data Structure” on page 8-80 for more information.

`reqlinks = rmi('get', object)` returns the requirement links data structure for `object`. `object` is the name or handle of a Simulink or Stateflow object with which requirements can be associated.

`reqlinks = rmi('get', object, group)` returns the requirement links data structure for the Signal Builder group specified by the index `group`. In this case, `object` is the name or handle of a Signal Builder block whose signal groups are associated with requirements.

`rmi('set', object, reqlinks)` sets the requirement links data structure `reqlinks` to `object`.

`rmi('set', object, reqlinks, group)` sets the requirement links data structure `reqlinks` to the Signal Builder group specified by the index `group`. In this case, `object` is the name or handle of a Signal Builder block whose signal groups you want to associate with requirements.

`rmi('cat', object, reqlinks)` appends the requirement links data structure `reqlinks` to the end of the existing structure associated with `object`. If no structure exists, the RMI sets `reqlinks` to `object`.

`cnt = rmi('count', object)` returns the number of requirement links associated with `object`.

`rmi('clearall', object)` removes the requirement links data structure associated with `object`, deleting its requirements.

### Navigation and Display Options

`cmdstr = rmi('navcmd', object)` returns the MATLAB command string used to navigate to `object`. `object` is the name or handle of a Simulink or Stateflow object with which requirements can be associated. See “Navigating to Simulink Models from External Documents” on page 2-42 for more information.

`[cmdstr, titlestr] = rmi('navcmd', object)` returns the MATLAB command string `cmdstr` and the title string `titlestr` that provides descriptive text for `object`.

`guidstr = rmi('gidget', object)` returns the globally unique identifier for `object`. A globally unique identifier is created for `object` if it lacks one. See “Providing Unique Object Identifiers” on page 2-42 for more information.

`object = rmi('guidlookup', model, guidstr)` returns the object name in `model` that has the globally unique identifier specified by `guidstr`.

`rmi('highlightModel', object)` highlights all of the objects in the parent model of `object` that have requirement links.

`rmi('unhighlightModel', object)` removes highlighting of objects in the parent model of `object` that have requirement links.

`rmi('view', object, index)` accesses the requirement numbered `index` in the requirements document associated with `object`. `index` is an integer that represents the *n*th requirement linked to `object`.

`dialog = rmi('edit', object)` displays the Requirements dialog box for `object` and returns the handle of the dialog box.

`rmi('copyObj', object)` resets the globally unique identifier for `object`, preserving its requirement links.

## Requirement Links Data Structure

Requirement links are represented using a MATLAB structure array with the following fields:

- `doc` — a string identifying the requirements document, equivalent to the **Document** field of the Requirements dialog box.
- `id` — a string defining a particular location in the requirements document. The first character in the string specifies the type of identifier that follows. Valid characters that can appear at the beginning of the string are

Character	Identifier	Example
?	Search text, the first occurrence of which is located in the requirements document	'?Requirement 1'
@	Named item, such as a bookmark in a Microsoft Word document or an anchor in an HTML document	'@my_req'
#	Page or item number	'#21'
>	Line number	'>3156'
\$	Worksheet range in a spreadsheet	'\$A2:C5'

- `linked` — a Boolean value specifying whether the requirement link is accessible for report generation and highlighting. The default value is 1 (true), specifying that the RMI can highlight the model object and include its requirement link in generated reports.
- `description` — a string describing the requirement, equivalent to the **Description** field of the Requirements dialog box.
- `keywords` — an optional string supplementing `description`, equivalent to the **User tag** field of the Requirements dialog box.
- `reqsys` — a string identifying the link type registration name. This field displays 'other' for built-in link types.



<b>Purpose</b>	Start Requirements Management Interface
<b>Syntax</b>	<code>rminav</code>
<b>Description</b>	<p><code>rminav</code> starts the Requirements Management Interface Navigator window.</p> <p>If you specified <code>reqsys = 'OTHERS'</code> in the MATLAB M-file <code>reqmgropts.m</code>, the standard version of the Requirements Management Interface Navigator window opens. You can associate requirements documents written in HTML or Microsoft Word and Excel software with Simulink models, Stateflow charts, and MATLAB M-files.</p> <p>If you specified <code>reqsys = 'DOORS'</code> in <code>reqmgropts.m</code>, the DOORS software version of the Requirements Management Interface Navigator window opens. You can associate requirements in the Telelogic DOORS software with Simulink models, Stateflow charts, and MATLAB M-files.</p> <p>To associate requirements in the DOORS software with MATLAB objects, you must start the MATLAB software with the <code>/automation</code> option.</p>

# sigrangeinfo

**Purpose** Display signal range coverage information for model object

**Syntax**  
`[min, max] = sigrangeinfo(cvdo, object)`  
`[min, max] = sigrangeinfo(cvdo, object, portID)`

**Description** `[min, max] = sigrangeinfo(cvdo, object)` returns the minimum and maximum signal values output by the model component object within the cvdata object cvdo. See “Specifying a Model Object” on page 8-82 for more information about the object argument. If object outputs a vector, min and max are also vectors.

`[min, max] = sigrangeinfo(cvdo, object, portID)` returns the minimum and maximum signal values associated with the output port portID of the Simulink block object.

## Specifying a Model Object

The object argument specifies an object in the Simulink model or Stateflow diagram that received decision coverage. Valid values for object include the following:

Object Specification	Description
BlockPath	Full path to a Simulink model or block
BlockHandle	Handle to a Simulink model or block
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	Description
{BlockPath, sfObj}	Cell array with the path to a Stateflow block and a Stateflow object API handle contained in that chart
[BlockHandle, sfID]	Array with a Stateflow block handle and the ID of an object contained in that chart

## Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable signal range coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';
open_system(mdl)
testObj = cvtest(mdl)
testObj.settings.sigrange = 1;
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the signal range for the `Product` block.

```
blk_handle = get_param([mdl, '/Product'], 'Handle');
[minVal, maxVal] = sigrangeinfo(data, blk_handle)
```

# tableinfo

---

## Purpose

Display lookup table coverage information for model object

## Syntax

```
coverage = tableinfo(cvdo, object)
coverage = tableinfo(cvdo, object, ignore_descendants)
[coverage, exeCounts] = tableinfo(cvdo, object)
[coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)
```

## Description

`coverage = tableinfo(cvdo, object)` returns lookup table coverage results from the cvdata object `cvdo` for the model component specified by `object`. `object` is the full path or handle to a Simulink lookup table block or a model containing such a block. The value of `coverage` is a two-element vector of form `[covered_intervals total_intervals]`, the elements of which are defined as follows:

- `covered_intervals` — the number of interpolation/extrapolation intervals satisfied for `object`
- `total_intervals` — the total number of interpolation/extrapolation intervals for `object`

---

**Note** `coverage` is empty if `cvdo` does not contain lookup table coverage results for `object`.

---

`coverage = tableinfo(cvdo, object, ignore_descendants)` returns lookup table coverage results for `object`, ignoring the coverage of its descendent objects if `ignore_descendants` is true (i.e., 1).

`[coverage, exeCounts] = tableinfo(cvdo, object)` returns lookup table coverage results and the execution count for each interpolation/extrapolation interval in the lookup table block specified by `object`. `exeCounts` is an array having the same dimensionality as the lookup table block; however, its size has been extended to allow for the lookup table extrapolation intervals.

`[coverage, exeCounts, brkEquality] = tableinfo(cvdo, object)` returns lookup table coverage results, the execution count for

each interpolation/extrapolation interval, and the execution counts for breakpoint equality. `brkEquality` is a cell array containing vectors that identify the number of times in a simulation the lookup table block input was equivalent to a breakpoint value. Each vector represents the breakpoints along a different lookup table dimension.

## Example

The following commands open the `slvndemo_cv_small_controller` demo model, create the test specification object `testObj`, enable lookup table coverage for `testObj`, and execute `testObj`.

```
mdl = 'slvndemo_cv_small_controller';  
open_system(mdl)  
testObj = cvtest(mdl)  
testObj.settings.tableExec = 1;  
data = cvsim(testObj)
```

Afterward, issue the following commands to retrieve the lookup table coverage results for the Gain Table block (in the Gain subsystem) and determine its percentage of interpolation/extrapolation intervals covered.

```
blk_handle = get_param([mdl, '/Gain/Gain Table'], 'Handle');  
cov = tableinfo(data, blk_handle)  
percent_cov = 100 * cov(1) / cov(2)
```

# tableinfo

---

# Block Reference

---

# System Requirements

**Purpose** List system requirements in Simulink diagrams

**Library** Simulink Verification and Validation

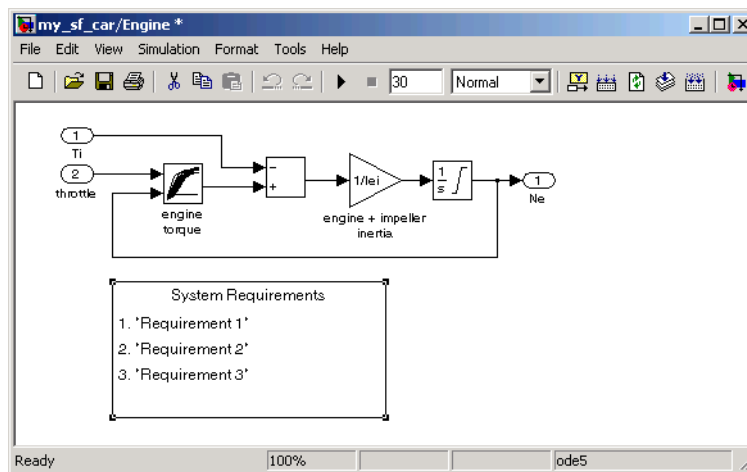
**Description**



The System Requirements block lists all the system requirements associated with the model or subsystem depicted in the current diagram. It does not list requirements associated with individual blocks in the diagram.

You can place this block anywhere in a diagram. It is not connected to other Simulink blocks. You can only have one System Requirements block in a diagram.

When you drag the System Requirements block from the Library Browser into your Simulink diagram, it is automatically populated with the system requirements, as shown.



Each of the listed requirements is an active link to the actual requirements document. When you double-click on a requirement name, the associated requirements document opens in its editor window, scrolled to the target location.

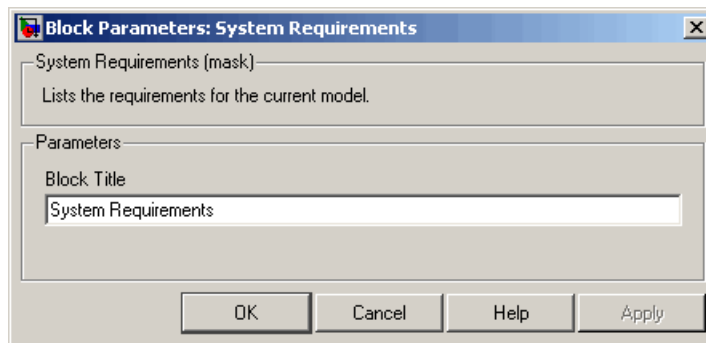


If the System Requirements block exists in a diagram, it automatically updates the requirements listing as you add, modify, or delete requirements for the model or subsystem.

For more information on using the System Requirements block, see “Displaying the System Requirements in a Diagram” on page 2-50.

## Dialog Box and Parameters

To access the Block Parameters dialog box for the System Requirements block, right-click on the System Requirements block and, from the resulting pop-up menu, select **Mask Parameters**. The Block Parameters dialog box opens, as shown.



The Block Parameters dialog box for the System Requirements block contains one parameter.

### Block Title

The title of the system requirements list in the diagram. The default title is **System Requirements**. You can type a customized title, for example, **Engine Requirements**.

# System Requirements

---

# Model Advisor Checks

---

- “Simulink® Verification and Validation Checks” on page 10-2
- “DO-178B Checks” on page 10-3
- “IEC 61508 Checks” on page 10-57
- “MathWorks Automotive Advisory Board Checks” on page 10-73
- “Requirements Consistency Checks” on page 10-140

## **Simulink Verification and Validation Checks**

## DO-178B Checks

<b>In this section...</b>
“Check safety-related optimization settings” on page 10-4
“Check safety-related diagnostic settings for solvers” on page 10-8
“Check safety-related diagnostic settings for sample time” on page 10-11
“Check safety-related diagnostic settings for signal data” on page 10-14
“Check safety-related diagnostic settings for parameters” on page 10-17
“Check safety-related diagnostic settings for data used for debugging” on page 10-20
“Check safety-related diagnostic settings for data store memory” on page 10-22
“Check safety-related diagnostic settings for type conversions” on page 10-24
“Check safety-related diagnostic settings for signal connectivity” on page 10-26
“Check safety-related diagnostic settings for bus connectivity” on page 10-28
“Check safety-related diagnostic settings that apply to function-call connectivity” on page 10-30
“Check safety-related diagnostic settings for compatibility” on page 10-32
“Check safety-related diagnostic settings for model referencing” on page 10-34
“Check safety-related model referencing settings” on page 10-38
“Check safety-related code generation settings” on page 10-40
“Check safety-related diagnostic settings for saving” on page 10-47
“Check for proper usage of For Iterator blocks” on page 10-49
“Check for proper usage of While Iterator blocks” on page 10-50
“Display model version information” on page 10-52
“Check for proper usage of blocks that compute absolute values” on page 10-53
“Check for proper usage of Relational Operator blocks” on page 10-55

## Check safety-related optimization settings

Check model configuration for optimization settings that can impact safety.

### Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

### Results and Recommended Actions

Condition	Recommended Action
Block reduction optimization is on. This optimization can remove blocks from generated code, resulting in requirements with no associated code and violations for traceability requirements. (See DO-178B, Section 6.3.4e—Source code is traceable to low-level requirements.)	Clear the <b>Block reduction</b> check box on the <b>Optimization</b> pane of the Configuration Parameters dialog box or set the parameter <code>BlockReduction</code> to off.
Conditional input branch execution is on. Because the model coverage tool does not account for this optimization, the optimization can result in the tool reporting 100% model coverage while coverage for the code using the same test cases can be less than 100%. (See DO-178B, Section 6.4.4.2—Test coverage of software structure is achieved.)	Clear the <b>Conditional input branch execution</b> check box on the <b>Optimization</b> pane of the Configuration Parameters dialog box or set the parameter <code>ConditionallyExecuteInputs</code> to off.
Implementation of logic signals as Boolean data is off. Strong data typing is recommended for safety-related code. (See DO-178B, Section 6.3.1e—High-level requirements conform to standards, DO-178B, Section 6.3.2e—Low-level requirements conform to standards, and MISRA C 2004, Rule 12.6.)	Select <b>Implement logic signals as boolean data (vs. double)</b> on the <b>Optimization</b> 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 <b>Application lifespan (days)</b> on the <b>Optimization</b> 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 on. This optimization can remove blocks that typecast data from generated code, resulting in requirements with no associated code and violations for traceability requirements. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 10.1.)</p>	<p>Clear the <b>Ignore integer downcasts in folded expressions</b> check box on the <b>Optimization</b> pane of the Configuration Parameters dialog box or set the parameter <code>EnforceIntegerDowncast</code> to <code>off</code>.</p>
<p>The optimization that suppresses the generation of initialization code for root-level inports and outports that are set to zero is on. For safety-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>Clear the <b>Remove root level I/O zero initialization</b> check box on the <b>Optimization</b> pane of the Configuration Parameters dialog box or set the parameter <code>ZeroExternalMemoryAtStartup</code> to <code>on</code>. Alternatively, integrate external, hand-written code that initializes all I/O variables to zero explicitly.</p>

Condition	Recommended Action
<p>The optimization that suppresses the generation of initialization code for internal work structures, such as block states and block outputs that are set to zero, is on. For safety-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>Clear the <b>Remove internal data zero initialization</b> check box on the <b>Optimization</b> pane of the Configuration Parameters dialog box or set the parameter <code>ZeroInternalMemoryAtStartup</code> to on. Alternatively, integrate external, hand-written code that initializes all state variables to zero explicitly.</p>
<p>The optimization that suppresses generation of code resulting from floating-point to integer conversions that wrap out-of-range values is off. You must avoid overflows for safety-related code. When this optimization is off and your model includes blocks that disable the <b>Saturate on overflow</b> 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>Select <b>Remove code from floating-point to integer conversions that wraps out-of-range values</b> on the <b>Optimization</b> pane of the Configuration Parameters dialog box or set the parameter <code>EfficientFloat2IntCast</code> to on.</p>
<p>The optimization that specifies whether to generate code that guards against division by zero for fixed-point data is on. You must avoid division-by-zero exceptions in safety-related code. (See DO-178B, Section 6.3.1g—Algorithms are accurate, DO-178B, Section 6.3.2g—Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>Clear the <b>Remove code that protects against division arithmetic exceptions</b> check box on the <b>Optimization</b> 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 <b>Algebraic loop</b> on the <b>Diagnostics &gt; Solver</b> 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 <b>Minimize algebraic loop</b> on the <b>Diagnostics &gt; Solver</b> 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>

<b>Condition</b>	<b>Recommended Action</b>
<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 <b>Block priority violation</b> on the <b>Diagnostics &gt; Solver</b> 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 <b>Unspecified inheritability of sample times</b> on the <b>Diagnostics &gt; Solver</b> 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 <b>Automatic solver parameter selection</b> on the <b>Diagnostics &gt; Solver</b> 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 <b>State name clash</b> on the <b>Diagnostics &gt; Solver</b> 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 <b>Source block specifies -1 sample time</b> on the <b>Diagnostics &gt; Sample Time</b> 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 <b>Discrete used as continuous</b> on the <b>Diagnostics &gt; Sample Time</b> 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 <b>Multitask rate transition</b> on the <b>Diagnostics &gt; Sample Time</b> 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 <b>Multitask conditionally executed subsystem</b> on the <b>Diagnostics &gt; Sample Time</b> 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 <b>Enforce sample times specified by Signal Specification blocks</b> on the <b>Diagnostics &gt; Sample Time</b> 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 <b>Signal resolution</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Division by singular matrix</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Underspecified data types</b> on the <b>Diagnostics &gt; Data Validity</b> 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 incorrect and unsafe application behavior. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>Set <b>Detect overflow</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Inf or NaN block output</b> on the <b>Diagnostics &gt; Data Validity</b> 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 "<b>rt</b>" <b>prefix for identifiers</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Simulation range checking</b> on the <b>Diagnostics &gt; Data Validity</b> 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 incorrect and unsafe behavior. (See DO-178B, Section 6.3.1g – Algorithms are accurate, DO-178B, Section 6.3.2g – Algorithms are accurate, and MISRA C 2004, Rule 21.1.)</p>	<p>Set <b>Detect downcast</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Detect underflow</b> on the <b>Diagnostics &gt; Data Validity</b> pane of the Configuration Parameters dialog box or set the parameter ParameterUnderflowMsg to error.</p>

<b>Condition</b>	<b>Recommended Action</b>
<p>The diagnostic that detects when a parameter overflow occurs is set to none or warning. Numeric overflows can result in incorrect and unsafe application behavior and should be detected and corrected in safety-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 <b>Detect overflow</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Detect precision loss</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Detect loss of tunability</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Model Verification block enabling</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Detect read before write</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Detect write after read</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Detect write after write</b> on the <b>Diagnostics &gt; Data Validity</b> 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 <b>Multitask data store</b> on the <b>Diagnostics &gt; Data Validity</b> 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 warning can result in requirements with no corresponding code. The removal of such blocks need to be detected so model developers can remove the unnecessary blocks explicitly. (See DO-178B, Section 6.3.1g – Algorithms are accurate and DO-178B, Section 6.3.2g – Algorithms are accurate.)</p>	<p>Set <b>Unnecessary type conversions</b> on the <b>Diagnostics &gt; Type Conversion</b> 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 <b>Vector/matrix block input conversion</b> on the <b>Diagnostics &gt; Type Conversion</b> 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 <b>32-bit integer to single precision float conversion</b> on the <b>Diagnostics &gt; Type Conversion</b> 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 <b>Signal label mismatch</b> on the <b>Diagnostics &gt; Connectivity</b> 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 <b>Unconnected block input ports</b> on the <b>Diagnostics &gt; Connectivity</b> 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 <b>Unconnected block output ports</b> on the <b>Diagnostics &gt; Connectivity</b> 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 <b>Unconnected line</b> on the <b>Diagnostics &gt; Connectivity</b> 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 Outport block is connected to a bus but does not specify a bus object is set to none or warning. For a bus signal to cross a model boundary, the signal must be defined as a bus object to ensure compatibility with higher level models that use a model as a reference model.</p>	<p>Set <b>Unspecified bus object at root Outport block</b> on the <b>Diagnostics &gt; Connectivity</b> pane of the Configuration Parameters dialog box or set the parameter <code>RootOutportRequireBusObject</code> to error.</p>

Condition	Recommended Action
<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 <b>Element name mismatch</b> on the <b>Diagnostics &gt; Connectivity</b> 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 an unintended operation or unpredictable behavior to occur.</p>	<p>Set <b>Mux blocks used to create bus signals</b> on the <b>Diagnostics &gt; Connectivity</b> pane of the Configuration Parameters dialog box or set the parameter <code>StrictBusMsg</code> to error. 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 <b>Invalid function-call connection</b> on the <b>Diagnostics &gt; Connectivity</b> 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 a function-call subsystem’s inputs 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 subsystem’s inputs.	Set <b>Context-dependent inputs</b> on the <b>Diagnostics &gt; Connectivity</b> 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 can impact safety.

### Description

This check verifies that model diagnostic configuration parameters pertaining to compatibility are set optimally for generating code for a safety-related application.

See DO-178B, Section 6.3.3b – Software architecture is consistent and MISRA C 2004, Rule 9.1.

### Results and Recommended Actions

Condition	Recommended Action
<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 <b>S-function upgrades needed</b> on the <b>Diagnostics &gt; Compatibility</b> pane of the Configuration Parameters dialog box or set the parameter <code>SFcnCompatibilityMsg</code> to error.</p>
<p>The <b>Check undefined subsystem initial output</b> diagnostic is off. This diagnostic specifies whether the Simulink software displays a warning if the model contains a conditionally executed subsystem in which a block with a specified initial condition drives an Outport block with an undefined initial condition. A conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.</p>	<p>Set <b>Check undefined subsystem initial output</b> on the <b>Diagnostics &gt; Compatibility</b> pane of the Configuration Parameters dialog box or set the parameter <code>CheckSSInitialOutputMsg</code> to on.</p>

Condition	Recommended Action
The diagnostic that detects potential initial output differences from earlier releases is set to off. An output of a conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.	Set <b>Check preactivation output of execution context</b> on the <b>Diagnostics &gt; Compatibility</b> pane of the Configuration Parameters dialog box or set the parameter <code>CheckExecutionContextPreStartOutputMsg</code> to on.
The diagnostic that detects potential output differences from earlier releases is set to off. An output of a conditionally executed subsystem could have an output that is not initialized and feeds into a block with a tunable parameter. If undetected, this condition can cause the behavior of the downstream block to be nondeterministic.	Set <b>Check runtime output of execution context</b> on the <b>Diagnostics &gt; Compatibility</b> pane of the Configuration Parameters dialog box or set the parameter <code>CheckExecutionContextRuntimeOutputMsg</code> to on.

### Action Results

Clicking **Modify Settings** configures model diagnostic settings that affect compatibility and that can impact safety.

### See Also

- Diagnosing Simulation Errors in the Simulink documentation
- Diagnostics Pane: Compatibility in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

## Check safety-related diagnostic settings for model referencing

Check model configuration for diagnostic settings that apply to model referencing and that can impact safety.

### Description

This check verifies that model diagnostic configuration parameters pertaining to model referencing are set optimally for generating code for a safety-related application.

### Results and Recommended Actions

Condition	Recommended Action
<p>The diagnostic that detects a mismatch between the version of the model that creates or refreshes a Model block and the referenced model's current version is set to none or warning. The detection occurs during load and update operations. Get the latest version of the referenced model from the software configuration management system, rather than using an older version. Using an older version can produce incorrect simulation results and mismatches between simulation and target code operation. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set <b>Model block version mismatch</b> on the <b>Diagnostics &gt; Model Referencing</b> pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceVersionMismatchMessage</code> to error.</p>
<p>The diagnostic that detects port and parameter mismatches during model loading and updating is set to none or warning. If undetected, such mismatches can lead to incorrect simulation results because the parent and referenced models have different interfaces. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set <b>Port and parameter mismatch</b> on the <b>Diagnostics &gt; Model Referencing</b> pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceIOMismatchMessage</code> to error.</p>

<b>Condition</b>	<b>Recommended Action</b>
<p>The <b>Model configuration mismatch</b> 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 <b>Model configuration mismatch</b> on the <b>Diagnostics &gt; Model Referencing</b> pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceCSMismatchMessage</code> to warning.</p>

Condition	Recommended Action
<p>The diagnostic that detects invalid internal connections to the current model's root-level Inport and Outport blocks is set to none or warning. When this condition is detected, the Simulink software might automatically insert hidden blocks into the model to correct the condition. The hidden blocks can result in generated code that has no traceable requirements. Setting the diagnostic to error forces model developers to correct the referenced models manually. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)</p>	<p>Set <b>Invalid root Inport/Outport block connection</b> on the <b>Diagnostics &gt; Model Referencing</b> pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceIOMessage</code> to error.</p>
<p>The diagnostic that detects whether To Workspace or Scope blocks are logging data in a referenced model is set to none or warning. Because To Workspace and Scope blocks are for debugging and not for embedded code, it is best to detect the condition so model developers can correct it. (See DO-178B, Section 6.3.1d – High-level requirements are verifiable and DO-178B, Section 6.3.2d – Low-level requirements are verifiable.)</p>	<p>Set <b>Unsupported data logging</b> on the <b>Diagnostics &gt; Model Referencing</b> pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceDataLoggingMessage</code> to error.</p>

### Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to model referencing and taht can impact safety.

### See Also

- Diagnosing Simulation Errors in the Simulink documentation
- Diagnostics Pane: Model Referencing in the Simulink graphical user interface documentation

- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

## Check safety-related model referencing settings

Check model configuration for model referencing settings that can impact safety.

### Description

This check verifies that model configuration parameters for model referencing are set optimally for generating code for a safety-related application.

### Results and Recommended Actions

Condition	Recommended Action
<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 <b>Rebuild options</b> on the <b>Model Referencing</b> 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 <b>Rebuild options</b> 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 <b>Never rebuild targets diagnostic</b> on the <b>Model Referencing</b> 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
The ability to pass scalar root input by value is on. This capability should be off because scalar values can change during a time step and result in unpredictable data. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set <b>Pass scalar root inputs by value</b> on the <b>Model Referencing</b> pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferencePassRootInputsByReference</code> to off.
The model is configured to minimize algebraic loop occurrences. This configuration is incompatible with the recommended setting of <b>Single output/update function</b> for embedded systems code. (See DO-178B, Section 6.3.3b – Software architecture is consistent.)	Set <b>Minimize algebraic loop occurrences</b> on the <b>Model Referencing</b> pane of the Configuration Parameters dialog box or set the parameter <code>ModelReferenceMinAlgLoopOccurrences</code> to off.

### Action Results

Clicking **Modify Settings** configures model referencing settings that can impact safety.

### See Also

- Model Dependencies in the Simulink documentation
- Model Referencing Pane in the Simulink graphical user interface documentation
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178B, Software Considerations in Airborne Systems and Equipment Certification standard

## Check safety-related code generation settings

Check model configuration for code generation settings that can impact safety.

### Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-related application.

### Results and Recommended Actions

Condition	Recommended Action
<p>The option to include comments in the generated code is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set <b>Include comments</b> on the <b>Real-Time Workshop &gt; Comments</b> &gt; pane of the Configuration Parameters dialog box or set the parameter <code>GenerateComments</code> to on.</p>
<p>The option to include comments that describe the code for blocks is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set <b>Simulink block / Stateflow object comments</b> on the <b>Real-Time Workshop &gt; Comments</b> pane of the Configuration Parameters dialog box or set the parameter <code>SimulinkBlockComments</code> to on.</p>
<p>The option to include comments that describe the code for blocks eliminated from a model is off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set <b>Show eliminated blocks</b> on the <b>Real-Time Workshop &gt; Comments</b> pane of the Configuration Parameters dialog box or set the parameter <code>ShowEliminatedStatement</code> to on.</p>

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 off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set <b>Verbose comments for SimulinkGlobal storage class</b> on the <b>Real-Time Workshop &gt; Comments</b> 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 off. Comments are necessary for good traceability between the code and the model. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set <b>Requirements in block comments</b> on the <b>Real-Time Workshop &gt; Comments</b> 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 on. Support for nonfinite numbers is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>Support: non-finite numbers</b> on the <b>Real-Time Workshop &gt; Interface</b> 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 on. Support for absolute time is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>Support: absolute time</b> on the <b>Real-Time Workshop &gt; Interface</b> pane of the Configuration Parameters dialog box or set the parameter <code>SupportAbsoluteTime</code> to off.</p>

<b>Condition</b>	<b>Recommended Action</b>
<p>The option to generate code for blocks that use continuous time is on. Support for continuous time is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>Support: continuous time</b> on the <b>Real-Time Workshop &gt; Interface</b> 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 on. This option requires support of nonfinite numbers, which is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>Support: non-inlined S-functions</b> on the <b>Real-Time Workshop &gt; Interface</b> 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 on. This option is inappropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>GRT compatible call interface</b> on the <b>Real-Time Workshop &gt; Interface</b> 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 off. Having a single call to the output and update functions simplifies the interface to the real-time operating system (RTOS) and simplifies verification of the generated code. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>Single output/update function</b> on the <b>Real-Time Workshop &gt; Interface</b> 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 on. This function deallocates dynamic memory, which is not appropriate for real-time safety-related systems. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>Terminate function required</b> on the <b>Real-Time Workshop &gt; Interface</b> 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 off. If you do not select this option, the Real-Time Workshop product generates extra code that might not be reachable for testing. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>Suppress error status in real-time model data structure</b> on the <b>Real-Time Workshop &gt; Interface</b> pane of the Configuration Parameters dialog box or set the parameter <code>SuppressErrorStatus</code> to on.</p>

<b>Condition</b>	<b>Recommended Action</b>
<p>MAT-file logging is enabled. This option adds extra code for logging test points to a MAT-file, which is not supported by embedded targets. Use this option only in test harnesses. (See DO-178B, Section 6.3.1c – High-level requirements are compatible with target computer and DO-178B, Section 6.3.2c – Low-level requirements are compatible with target computer.)</p>	<p>Set <b>MAT-file logging</b> on the <b>Real-Time Workshop &gt; Interface</b> pane of the Configuration Parameters dialog box or set the parameter <code>MatFileLogging</code> to on.</p>
<p>The option that specifies the style for parenthesis usage is set to Minimum (Rely on C/C++ operators precedence) or to Nominal (Optimize for readability). For safety-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 <b>Parenthesis level</b> on the <b>Real-Time Workshop &gt; Code</b> pane of the Configuration Parameters dialog box or set the parameter <code>ParenthesisStyle</code> to Maximum (Specify precedence with parentheses).</p>
<p>The option that specifies whether to preserve operand order is off. This option increases the traceability of the generated code. (See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set <b>Preserve operand order in expression</b> on the <b>Real-Time Workshop &gt; Code</b> 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 off. This option increases the traceability of the generated code. ( See DO-178B, Section 6.3.4e – Source code is traceable to low-level requirements.)</p>	<p>Set <b>Preserve condition expression in if statement</b> on the <b>Real-Time Workshop &gt; Code</b> 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 <b>Minimum mangle length</b> on the <b>Real-Time Workshop &gt; Symbols</b> pane of the Configuration Parameters dialog box or set 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.

### 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.	Set <b>Block diagram contains disabled library links</b> on the <b>Diagnostics &gt; Saving</b> > 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.	Set <b>Block diagram contains parameterized library links</b> on the <b>Diagnostics &gt; Saving</b> > 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 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"> <li>• Set the <b>Iteration limit source</b> parameter of the For Iterator block to <code>internal</code>.</li> <li>• If the <b>Iteration limit source</b> parameter of the For Iterator block must be <code>external</code>, use a Constant, Probe, or Width block as the source.</li> <li>• Avoid selecting the <b>Set next i (iteration variable) externally</b> parameter of the For Iterator block.</li> </ul>

### 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"> <li>• Set the <b>Maximum number of iterations</b> parameter of the While Iterator block to a positive integer value.</li> <li>• Consider selecting the <b>Show iteration number port</b> 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.</li> </ul>

### 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"> <li>• 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.</li> <li>• Computes an absolute value of a signed integer and <b>Saturate on integer overflow</b> is not selected for that block. Taking the absolute value of full scale negative integer value results in an overflow.</li> </ul>	<ul style="list-style-type: none"> <li>• To avoid unreachable code, change the input to the Absolute Value block to a signed input type.</li> <li>• To avoid overflows, select the <b>Saturate on integer overflow</b> check box of the Absolute Value block.</li> </ul>

### **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

<b>In this section...</b>
“Display model metrics and complexity” on page 10-58
“Check for unconnected objects” on page 10-59
“Check for fully defined interface” on page 10-60
“Check for questionable blocks” on page 10-62
“Check usage of Stateflow” on page 10-64
“Display configuration management data” on page 10-67
“Check usage of Simulink” on page 10-68

## Display model metrics and complexity

Display model metrics and complexity information.

### 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"><li>• Double-click an element in the list of unconnected items to locate the item in the model diagram.</li><li>• Properly connect the objects identified in the results.</li></ul>

### See Also

“Working with Signals” in the Simulink documentation

## Check for fully defined interface

Identify root model Inport blocks that do not have fully defined attributes.

### Description

Using root model Inport blocks that do not have fully define dimensions, sample time, or data type can lead to undesired simulation results. Simulink back-propagates dimensions, sample times, and data types from downstream blocks unless you explicitly assign these values.

See IEC 61508-3, Table B.9 (5) – Fully defined interface.

### Results and Recommended Actions

Condition	Recommended Action
The model has root-level Inport blocks that have undefined attributes, such as an inherited sample time, data type, or port dimension.	Explicitly define all root-level Inport block attributes identified in the results. Double-click an element from the list of underspecified items to locate the condition.

### Tip

The following configurations pass this check:

- Inport blocks with inherited port dimensions in conjunction with the usage of bus objects
- Inport blocks with automatically inherited data types in conjunction with bus objects
- Inport blocks with inherited sample times in conjunction with the **Periodic sample time constraint** menu set to **Ensure sample time independent**

### See Also

- Working with Data Types in the Simulink documentation
- Determining Output Signal Dimensions in the Simulink documentation

- Specifying Sample Time in the Simulink documentation

## Check for questionable blocks

Identify blocks not supported by code generation or not recommended for deployment.

### Description

This check partially identifies model constructs that are not suited for code generation or not recommended for production code generation as identified in the Simulink Block Support tables for Real-Time Workshop and Real-Time Workshop Embedded Coder. If you are using blocks with support notes for code generation, review the information and follow the given advice.

See IEC 61508-3, Table A.3 (3) – Language subset.

### Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for code generation.	Consider replacing the blocks listed in the results. Double-click an element from the list of questionable items to locate condition.
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Double-click an element from the list of questionable items to locate condition.
The model or subsystem contains Gain blocks whose value equals 1.	If you are using Gain blocks as buffers, consider replacing them with Signal Conversion blocks. Double-click an element from the list of questionable items to locate condition.

### Limitation

This check might not identify all instances of noncompliance with the Real-Time Workshop and Real-Time Workshop Embedded Coder Simulink Block Support tables.



**See Also**

- “Simulink Block Support” tables in the Real-Time Workshop documentation for Real-Time Workshop and Real-Time Workshop Embedded Coder
- “Requirements and Restrictions for ERT-Based Simulink Models” in the Real-Time Workshop Embedded Coder documentation

## Check usage of Stateflow

Identify usage of Stateflow that can impact safety.

### Description

This check identifies instances of Stateflow software being used in a way that can impact an application's safety, including

- Use of strong data typing
- Port name mismatches
- Scope of data objects and events
- Formatting of state action statements

See

- IEC 61508-3, Table A.3 (2) – Strongly typed programming language
- IEC 61508-3, Table A.3 (3) – Language subset
- IEC 61508-3, Table B.9 (2) – Information hiding/encapsulation
- MISRA C:2004, Rule 10.1
- MISRA C:2004, Rule 10.2
- MISRA C:2004, Rule 10.3
- MISRA C:2004, Rule 10.4
- MAAB Control Algorithm Modeling Guidelines, db\_0122: Stateflow and Simulink interface signals and parameters
- MAAB Control Algorithm Modeling Guidelines, db\_0123: Stateflow port names
- MAAB Control Algorithm Modeling Guidelines, db\_0125: Scope of internal signals and local auxiliary variables
- MAAB Control Algorithm Modeling Guidelines, db\_0126: Scope of Events
- MAAB Control Algorithm Modeling Guidelines, jc\_0501: Format of entries in a state block

## Results and Recommended Actions

Condition	Recommended Action
A Stateflow chart is not configured for strong data typing on boundaries between a Simulink model and theStateflow chart.	Enable the option <b>Use Strong Data Typing with Simulink I/O</b> 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"> <li>• Check whether the ports are connected properly and, if not, correct the connections.</li> <li>• Change the names of the signals or the Stateflow ports so that the names match.</li> </ul>
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"> <li>• An entry (en), during (du), or exit (ex) statement</li> <li>• The semicolon (;) at the end of an assignment statement</li> </ul>	Add missing new lines.

### See Also

See the following topics in the Stateflow documentation

- “Strong Data Typing with Simulink I/O”
- “Property Fields”
- “Defining Events”
- “Defining Data”
- “Labeling States”

## Display configuration management data

Display model configuration and checksum information.

### Description

This informer check displays the following information for the current model:

- Model version number
- Model author
- Date
- Model checksum

See IEC 61508-3, Table A.8 (5) – Software configuration management.

### Results and Recommended Actions

Condition	Recommended Action
Could not retrieve model version and checksum information.	This summary is provided for your information. No action is required.

### See Also

- “How Simulink Helps You Manage Model Versions” in the Simulink documentation
- Model Change Log in the Simulink® Report Generator™ documentation
- Simulink.BlockDiagram.getChecksum in the Simulink documentation
- Simulink.SubSystem.getChecksum in the Simulink documentation

## Check usage of Simulink

Identify usage of Simulink blocks that can 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"> <li>• Change the input of the Abs block to a signed input type.</li> <li>• Remove the Abs from the model.</li> </ul>
<p>The model or subsystem contains an Abs block that is operating on a signed integer value, and the <b>Saturate on integer overflow</b> 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 <b>Saturate on integer overflow</b> 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 <b>Output data type</b> to boolean in the <b>Block Parameters &gt; Signal Attributes</b> pane for the specified blocks.</p>

<b>Condition</b>	<b>Recommended Action</b>
The model or subsystem contains a block computing a relational operator that uses the == or ~= operator to compare floating-point signals. The use of these operators on floating-point signals is unreliable and unpredictable because of floating-point precision issues, and can lead to unpredictable results in the generated code.	For the specified blocks, do one of the following: <ul style="list-style-type: none"><li>• Change the signal data type.</li><li>• Rework the model to eliminate the need to use == or ~= operators on floating-point signals.</li></ul>



Condition	Recommended Action
<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"> <li>• Set the <b>Maximum number of iterations (-1 for unlimited)</b> parameter to a positive integer value.</li> <li>• Consider selecting the <b>Show iteration number port</b> check box and observe the iteration value during simulation.</li> </ul>
<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"> <li>• Set the <b>Iteration limit source</b> parameter to <code>internal</code>.</li> <li>• If the <b>Iteration limit source</b> parameter must be <code>external</code>, use a Constant, Probe, or Width block as the source.</li> <li>• Clear the <b>Set next i (iteration variable) externally</b> check box.</li> <li>• Consider selecting the <b>Show iteration variable</b> check box and observe the iteration value during simulation.</li> </ul>

### 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...

- “Check for difference in font and font sizes” on page 10-76
- “Check transition orientations in flow charts” on page 10-78
- “Check for display of nondefault block attributes” on page 10-79
- “Check for proper labeling on signal lines” on page 10-80
- “Check for propagated labels on signal lines” on page 10-82
- “Check default transition placement in Stateflow charts” on page 10-84
- “Check setting Stateflow graphical function return value” on page 10-85
- “Check for blocks not using one-based indexing” on page 10-86
- “Check for invalid file names” on page 10-88
- “Check for invalid model directory names” on page 10-90
- “Check for blocks that are not discrete ” on page 10-91
- “Check for prohibited sink blocks” on page 10-92
- “Check for invalid port positioning and configuration” on page 10-93
- “Check for mismatches between names of ports and corresponding signals” on page 10-95
- “Check whether block names do not appear below blocks” on page 10-96
- “Check for systems that mix primitive blocks and subsystems” on page 10-97
- “Check whether model has unconnected block input ports, output ports, or signal lines” on page 10-99
- “Check for improperly positioned Trigger and Enable blocks” on page 10-100
- “Check whether annotations have drop shadows” on page 10-101
- “Check whether tunable parameters specify expressions, data type conversions, or indexing operations” on page 10-102
- “Check whether Stateflow events are defined at the chart level or below” on page 10-104

**In this section...**

“Check whether Stateflow data objects with local scope are defined at the chart level or below” on page 10-105

“Check interface signals and parameters” on page 10-106

“Check for exclusive states, default states, and substate validity” on page 10-107

“Check optimization parameters for Boolean data types” on page 10-109

“Check model diagnostic settings” on page 10-110

“Check the display attributes of block names” on page 10-114

“Check icon display attributes for port blocks” on page 10-115

“Check whether subsystem block names include invalid characters” on page 10-116

“Check whether Inport and Outport block names include invalid characters” on page 10-118

“Check whether signal line names include invalid characters” on page 10-120

“Check whether block names include invalid characters” on page 10-122

“Check Trigger and Enable block port names” on page 10-124

“Check for Simulink diagrams that have nonstandard appearance attributes” on page 10-125

“Check visibility of port block names” on page 10-128

“Check for direction of subsystem blocks” on page 10-130

“Check for proper position of constants used in Relational Operator blocks” on page 10-131

“Check for entry format in state blocks” on page 10-132

“Check for use of tunable parameters in Stateflow” on page 10-134

“Check for proper use of Switch blocks” on page 10-135

“Check for proper use of signal buses and Mux block usage” on page 10-136

**In this section...**

“Check for mismatches between Stateflow ports and associated signal names” on page 10-138

“Check for proper scope of From and Goto blocks” on page 10-139

## 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 <b>Modify all Fonts</b> , or manually change the fonts and font sizes of text elements in the model such that they are consistent and portable.

### Action Results

Clicking **Modify all Fonts** changes the font and font size of all text elements in the model according to the values you specify for the font parameters.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check transition orientations in flow charts

Check transition orientations in flow charts.

### Description

The following rules apply to transitions in flow charts:

- Draw transition conditions horizontally.
- Draw transitions with a condition action vertically.

Loop constructs are exceptions to these rules.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db\_0132: Transitions in flowcharts.

### Results and Recommended Actions

Condition	Recommended Action
The model includes a transition with a condition that is not drawn horizontally or a transition action that is not drawn vertically.	Modify the model.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*



## Check for display of nondefault block attributes

Check for display of nondefault block attributes.

### Description

Model diagrams should display block parameters that have values other than default values. One way of displaying this information is by using the **Block Annotation** tab in the Block Properties dialog box.

This guideline facilitates

- Readability
- Verification and validation

See MAAB guideline db\_0140: Display of basic block attributes.

### 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 <b>Block Annotation</b> tab in the Block Properties dialog to add block parameter annotations.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for proper labeling on signal lines

Check for proper labeling on signal lines.

### Description

You should use a label to identify:

- Signals originating from the following blocks (the block icon exception noted below applies to all blocks listed except Inport, Bus Selector, Demux, and Selector):

- Bus Selector block (tool forces labeling)
- Chart block (Stateflow)
- Constant block
- Data Store Read block
- Demux block
- From block
- Inport block
- Selector block
- Subsystem block

---

**Block Icon Exception** If a signal label is visible in the display of the icon for the originating block, you do not have to display a label for the connected signal unless the signal label is needed elsewhere due to a rule for signal destinations.

---

- Signals connected to one of the following destination blocks (directly or indirectly with a basic block that performs an operation that is not transformative):

- Bus Creator block
- Chart block (Stateflow)
- Data Store Write block
- Goto block
- Mux block
- Outport block
- Subsystem block

- Any signal of interest.

This guideline facilitates

- Readability
- Workflow
- Verification and validation
- Code generation

See MAAB guideline na\_0008: Proper labeling of signal lines.

## Results and Recommended Actions

Condition	Recommended Action
Signals coming from Bus Selector, Chart, Constant, Data Store Read, Demux, From, Inport, or Selector blocks are not labeled.	Double-click the line that represents the signal. After the text cursor appears, enter a name and click anywhere outside the label to exit label editing mode.

## See Also

- Signal Labels in the Simulink documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for propagated labels on signal lines

Check for propagated labels on signal lines.

### Description

You should propagate a signal label from its source rather than enter the signal label explicitly (manually) if the signal originates from:

- An Inport block in a nested subsystem. However, if the nested subsystem is a library subsystem, you can explicitly label the signal coming from the Inport block to accommodate reuse of the library block.
- A basic block that performs a nontransformative operation.
- A Subsystem or Stateflow Chart block. However, if the connection originates from the output of an instance of the library block, you can explicitly label the signal to accommodate reuse of the library block.

This guideline facilitates

- Readability
- Workflow
- Verification and validation
- Code generation

See MAAB guideline na\_0009: Entry versus propagation of signal labels.

### Results and Recommended Actions

Condition	Recommended Action
The model includes signal labels that were entered explicitly, but should be propagated.	Use the open angle bracket (<) character to mark signal labels that should be propagated and remove the labels that were entered explicitly.

**See Also**

- Signal Labels in the Simulink documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check default transition placement in Stateflow charts

Check default transition placement in Stateflow charts.

### Description

In a Stateflow chart, you should connect the default transition at the top of the state and place the destination state of the default transition above other states in the hierarchy.

Properly position the default transition and its destination state for:

- Readability

See MAAB guideline jc\_0531: Placement of default transition.

### Results and Recommended Actions

Condition	Recommended Action
The default transition for a Stateflow chart is not connected at the top of the state.	Move the default transition to the top of the state chart.
The destination state of a Stateflow chart's default transition is lower than other states in the same hierarchy.	Adjust the position of the default transition's destination state such that the state is above other states in the same hierarchy.

### See Also

- “Defining Transitions Between States” in the Stateflow documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check setting Stateflow graphical function return value

Check setting Stateflow graphic function return value.

### Description

The return value from a Stateflow graphical function must be set in only one place.

This guideline facilitates

- Workflow
- Code generation

See MAAB guideline jc\_0511: Setting the return value from a graphical function.

### Results and Recommended Actions

Condition	Recommended Action
The return value from a Stateflow graphical function is set in multiple places.	Modify the function such that its return value is set in one place.

### See Also

- “Graphical Functions” in the Stateflow documentation
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for blocks not using one-based indexing

Check for blocks that do not use one-based indexing.

### Description

One-based indexing ([1, 2, 3,...]) is used for the following:

Product	Items
MATLAB	<ul style="list-style-type: none"> <li>• Workspace variables and structures</li> <li>• Local variables of MATLAB functions</li> <li>• Global variables</li> </ul>
Simulink	<ul style="list-style-type: none"> <li>• Signal vectors and matrices</li> <li>• Parameter vectors and matrices</li> <li>• S-function input and output signal vectors and matrices in M-code</li> <li>• S-function parameter vectors and matrices in M-code</li> <li>• S-function local variables in M-code</li> </ul>
Stateflow	<ul style="list-style-type: none"> <li>• Input and output signal vectors and matrices</li> <li>• Parameter vectors and matrices</li> <li>• Local variables</li> </ul>

Zero-based indexing ([0, 1, 2, ...]) is used for the following:



Product	Items
Simulink	<ul style="list-style-type: none"> <li>• Signal vectors and matrices</li> <li>• S-function input and output signal vectors and matrices in C code</li> <li>• S-function input parameters in C code</li> <li>• S-function parameter vectors and matrices in C code</li> <li>• S-function local variables in C code</li> </ul>
Stateflow	<ul style="list-style-type: none"> <li>• Variables and structures in custom C code</li> </ul>
C code	<ul style="list-style-type: none"> <li>• Local variables and structures</li> <li>• Global variables</li> </ul>

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline db\_0112: Indexing.

## Results and Recommended Actions

Condition	Recommended Action
Blocks in your model are not configured for one-based indexing.	Using block parameters, configure all blocks for one-based indexing.

## See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for invalid file names

Check for files residing in the same directory as the model that have illegal file names.

### Description

This guideline facilitates

- Readability
- Workflow

See MAAB guideline 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**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for invalid model directory names

Checks model directory and subdirectory names for invalid characters.

### Description

This guideline facilitates

- Readability
- Workflow

See MAAB guideline 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

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for blocks that are not discrete

Check for blocks that are not discrete.

### Description

You cannot include continuous blocks in controller models.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jm\_0001: Prohibited Simulink standard blocks inside controllers.

### Results and Recommended Actions

Condition	Recommended Action
Continuous blocks — Derivative, Integrator, State-Space, Transfer Fcn, Transfer Delay, Variable Time Delay, Variable Transport Delay, and Zero-Pole — are not permitted in models representing discrete controllers.	Replace continuous blocks with the equivalent blocks discretized in the s-domain by using the Discretizing library, as explain in “How to Discretize Blocks from the Simulink Model” in the Simulink documentation.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for prohibited sink blocks

Check for prohibited Simulink sink blocks.

### Description

You must design controller models from discrete blocks. Sink blocks, such as the Scope block, are not allowed.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline hd\_0001: Prohibited Simulink Sink blocks.

### Results and Recommended Actions

Condition	Recommended Action
Sink blocks are not permitted in discrete controllers.	Remove sink blocks from the model.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for invalid port positioning and configuration

Check whether the model contains ports with invalid position and configuration.

### Description

In models, ports must comply with the following rules:

- Place Inport blocks on the left side of the diagram. Move the Inport block right only to prevent signal crossings.
- Place Outport blocks on the right side of the diagram. Move the Outport block left only to prevent signal crossings.
- Avoid using duplicate Inport blocks at the subsystem level if possible.
- Do not use duplicate Inport blocks at the root level.

This guideline facilitates

- Readability

See MAAB guideline db\_0042: Port block in Simulink models.

### Results and Recommended Actions

Condition	Recommended Action
Inport blocks are too far to the right and result in left-flowing signals.	Move the specified Inport blocks to the left.
Outport blocks are too far to the left and result in right-flowing signals.	Move the specified Output blocks to the right.

<b>Condition</b>	<b>Recommended Action</b>
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><li data-bbox="872 586 1317 647">• If the duplicate Inport blocks are at the root level, remove them.</li></ul>

**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*



## Check for mismatches between names of ports and corresponding signals

Check for mismatches between names of ports and corresponding signals.

### Description

Use matching names for ports and their corresponding signals.

This guideline facilitates

- Readability
- Workflow
- Simulation

See MAAB guideline jm\_0010: Port block names in Simulink models.

### Results and Recommended Actions

Condition	Recommended Action
Ports have names that differ from their corresponding signals.	Change the port name or the signal name to match the correct name for the signal.

### Limitations

Prerequisite MAAB guidelines for this check are:

- db\_0042: Ports in Simulink models
- na\_0005: Port block name visibility in Simulink models

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether block names do not appear below blocks

Check whether block names do not appear below blocks.

### Description

If shown, the name of all blocks should appear below the blocks.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline db\_0142: Position of block names.

### Results and Recommended Actions

Condition	Recommended Action
Blocks have names that do not appear below the blocks.	Set the name of the block to appear below the blocks.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for systems that mix primitive blocks and subsystems

Check for systems that mix primitive blocks and subsystems.

### Description

You must design every level of a model with building blocks of the same type, for example, only subsystems or only primitive (basic) blocks.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db\_0143: Similar block types on the model levels.

### Results and Recommended Actions

Condition	Recommended Action
A level in the model includes both subsystem blocks and primitive blocks.	<ul style="list-style-type: none"> <li>• Move nonvirtual blocks into the subsystem.</li> <li>• 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.</li> </ul>

### **See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether model has unconnected block input ports, output ports, or signal lines

Check whether model has unconnected input ports, output ports, or signal lines.

### Description

All unconnected inputs should be connected to ground blocks. All unconnected outputs should be connected to terminator blocks. Respecting the guideline eliminates error messages.

See MAAB guideline db\_0081: Unconnected signals, block inputs, and block outputs.

### Results and Recommended Actions

Condition	Recommended Action
Blocks have unconnected inputs or outputs.	Connect unconnected lines to blocks specified by the design or to Ground or Terminator blocks.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for improperly positioned Trigger and Enable blocks

Check for improperly positioned Trigger and Enable blocks.

### Description

Locate blocks that define subsystems as conditional or iterative at the top of the subsystem diagram.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db\_0146: Triggered, enabled, conditional Subsystems.

### Results and Recommended Actions

Condition	Recommended Action
Trigger , Enable, and Action Port blocks are not centered in the upper third of the model diagram.	Move the Trigger, Enable, and Action Port blocks to the correct area of the model diagram.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether annotations have drop shadows

Check whether annotations have drop shadows.

### Description

Annotations should not have a drop shadow for readability.

This guideline facilitates

- Readability

See MAAB guideline jm\_0013: Annotations.

### Results and Recommended Actions

Condition	Recommended Action
Annotations display drop shadows.	Clear the <b>Format &gt; Show Drop Shadow</b> menu option.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether tunable parameters specify expressions, data type conversions, or indexing operations

Check whether tunable parameters specify expressions, data type conversions, or indexing operations.

### Description

To ensure that a parameter is tunable, you must enter the basic block without the use of MATLAB calculations or scripting. For example, omit

- Expressions
- Data type conversions
- Selections of rows or columns

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline db\_0110: Tunable parameters in basic blocks.

### Results and Recommended Actions

Condition	Recommended Action
Blocks have a tunable parameter that specifies an expression, data type conversion, or indexing operation.	In each case, move the calculation outside of the block, for example, by performing the calculation with a series of Simulink blocks, or precompute the value in the base workspace as a new variable.



**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether Stateflow events are defined at the chart level or below

Check whether Stateflow events are defined at the chart level or below.

### Description

All events of a Stateflow chart must be defined at the chart level or lower. Events cannot be at the machine level; that is, charts cannot interact with local events.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db\_0126: Scope of events.

### Results and Recommended Actions

Condition	Recommended Action
An event in a chart is not defined at the chart level or below.	Define the event at the chart level or below.

### See Also

- “Defining Events” in the Stateflow documentation.
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether Stateflow data objects with local scope are defined at the chart level or below

Check whether Stateflow data objects with local scope are defined at the chart level or below.

### Description

You must define all local data of a Stateflow block on the chart level or below in the object hierarchy. You cannot define local variables on the machine level; however, parameters and constants are allowed at the machine level.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db\_0125: Scope of internal signals and local auxiliary variables.

### Results and Recommended Actions

Condition	Recommended Action
Local data is not defined in the Stateflow hierarchy at the chart level or below.	Define local data at the chart level or below.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check interface signals and parameters

Check whether labeled Stateflow and Simulink input and output signals are strongly typed.

### Description

Strong data typing between Stateflow and Simulink input and output signals is required.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db\_0122: Stateflow and Simulink interface signals and parameters.

### Results and Recommended Actions

Condition	Recommended Action
A Stateflow chart does not use strong data typing with Simulink.	Select the <b>Use Strong Data Typing with Simulink I/O</b> check box for the specified block.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for exclusive states, default states, and substate validity

Check states in state machines.

### Description

In state machines:

- There must be at least two exclusive states.
- A state cannot have only one substate.
- The initial state of a hierarchical level with exclusive states is clearly defined by a default transition.

This guideline facilitates

- Readability
- Workflow
- Verification and validation

See MAAB guideline db\_0137: States in state machines.

### Prerequisite

A prerequisite MAAB guideline for this check is db\_0149: Flowchart patterns for conditional 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.

<b>Condition</b>	<b>Recommended Action</b>
Chart does not have a default state defined.	Define a default state.
Chart has multiple default states defined.	Define only one default state. Make the others nondefault.
State has only one exclusive (OR) substate.	Make the state a parallel state, or add another exclusive (OR) state.
State does not have a default substate defined.	Define a default substate.
State has multiple default substates defined.	Define only one default substate, make the others nondefault.

**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check optimization parameters for Boolean data types

Check the optimization parameter for Boolean data types.

### Description

Optimization for Boolean data types is required

This guideline facilitates

- Workflow
- Code generation

See MAAB guideline jc\_0011: Optimization parameters for Boolean data types.

### Results and Recommended Actions

Condition	Recommended Action
Configuration setting for <b>Implement logic signals as boolean data (vs. double)</b> is incorrect.	Select the <b>Implement logic signals as boolean data (vs. double)</b> check box in the Configuration Parameters dialog box <b>Optimization</b> pane.

### Prerequisite

A prerequisite MAAB guideline for this check is na\_0002: Appropriate implementation of fundamental logic and numerical operations.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check model diagnostic settings

Check the model diagnostics configuration parameter settings.

### Description

You should enable the following diagnostics:

**Algebraic loop**

**Minimize algebraic loop**

**Inf or NaN block output**

**Duplicate data store names**

**Unconnected block input ports**

**Unconnected block output ports**

**Unconnected line**

**Unspecified bus object at root Output 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
<p><b>Algebraic loop</b> is set to none.</p>	<p>Set <b>Algebraic loop</b> on the <b>Diagnostics &gt; Solver</b> 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.</p>
<p><b>Minimize algebraic loop</b> is set to none.</p>	<p>Set <b>Minimize algebraic loop</b> on the <b>Diagnostics &gt; Solver</b> 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.</p>
<p><b>Inf or NaN block output</b> is set to none, which can result in numerical exceptions in the generated code.</p>	<p>Set <b>Inf or NaN block output</b> on the <b>Diagnostics &gt; Data Validity &gt; Signals</b> pane of the Configuration Parameters dialog box to error or warning.</p>
<p><b>Duplicate data store names</b> is set to none, which can result in nonunique variable naming in the generated code.</p>	<p>Set <b>Duplicate data store names</b> on the <b>Diagnostics &gt; Data Validity &gt; Signals</b> pane of the Configuration Parameters dialog box to error or warning.</p>
<p><b>Unconnected block input ports</b> is set to none, which prevents code generation.</p>	<p>Set <b>Unconnected block input ports</b> on the <b>Diagnostics &gt; Data Validity &gt; Signals</b> pane of the Configuration Parameters dialog box to error or warning.</p>

Condition	Recommended Action
<p><b>Unconnected block output ports</b> is set to none, which can lead to dead code.</p>	<p>Set <b>Unconnected block output ports</b> on the <b>Diagnostics &gt; Data Validity &gt; Signals</b> pane of the Configuration Parameters dialog box to error or warning.</p>
<p><b>Unconnected line</b> is set to none, which prevents code generation.</p>	<p>Set <b>Unconnected line</b> on the <b>Diagnostics &gt; Connectivity &gt; Signals</b> pane of the Configuration Parameters dialog box to error or warning.</p>
<p><b>Unspecified bus object at root Output block</b> is set to none, which can lead to an unspecified interface if the model is referenced from another model.</p>	<p>Set <b>Unspecified bus object at root Output block</b> on the <b>Diagnostics &gt; Connectivity &gt; Buses</b> pane of the Configuration Parameters dialog box to error or warning.</p>
<p><b>Mux blocks used to create bus signals</b> is set to none, which can lead to an unintended bus being created in the model.</p>	<p>Set <b>Mux blocks used to create bus signals</b> on the <b>Diagnostics &gt; Connectivity &gt; Buses</b> pane of the Configuration Parameters dialog box to error or warning.</p>
<p><b>Element name mismatch</b> is set to none, which can lead to an incorrect interface in the generated code.</p>	<p>Set <b>Element name mismatch</b> on the <b>Diagnostics &gt; Connectivity &gt; Buses</b> pane of the Configuration Parameters dialog box to error or warning.</p>
<p><b>Invalid function-call connection</b> is set to none, which can lead to an error in the operation of the generated code.</p>	<p>Set <b>Invalid function-call connection</b> on the <b>Diagnostics &gt; Connectivity &gt; Function Calls</b> 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>

**Tip****See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check the display attributes of block names

Check the display attributes of block names.

### Description

Block names should be displayed when providing descriptive information. Block names should not be displayed if the block function is known from its appearance.

This guideline facilitates

- Readability

See MAAB guideline jc\_0061: Display of block names.

### Results and Recommended Actions

Condition	Recommended Action
Block name is not descriptive.	These block names should be modified to be more descriptive or not be shown.
Block name is not displayed.	These block names should be shown since they appear to have a descriptive name.
Block name is obvious.	These block names should not be displayed.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check icon display attributes for port blocks

Check the **Icon display** setting for Inport and Outport blocks.

### Description

The **Icon display** setting is required.

This guideline facilitates

- Readability

See MAAB guideline jc\_0081: Icon display for port block.

### Results and Recommended Actions

Condition	Recommended Action
The <b>Icon display</b> setting is incorrect.	Set the <b>Icon display</b> to Port number for the specified Inport and Outport blocks.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether subsystem block names include invalid characters

Check whether subsystem block names include invalid characters.

### Description

The names of all subsystem blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc\_0201: Usable characters for Subsystem name.

### Results and Recommended Actions

Condition	Recommended Action
The subsystem name contains illegal characters.	Rename the subsystem. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The subsystem name starts with a number.	Rename the subsystem.
The subsystem name starts with an underscore ("_").	Rename the subsystem.
The subsystem name ends with an underscore ("_").	Rename the subsystem.
The subsystem name contains consecutive underscores.	Rename the subsystem.
The subsystem name has consecutive underscores.	Rename the subsystem.
The subsystem name has blank spaces.	Rename the subsystem.

**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

**Tip**

Use underscores to separate parts of a subsystem name instead of spaces.

## Check whether Inport and Outport block names include invalid characters

Check whether Inport and Outport block names include invalid characters.

### Description

The names of all Inport and Outport blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc\_0211: Usable character for Inport block and Outport block.

### Results and Recommended Actions

Condition	Recommended Action
The block name contains illegal characters.	Rename the block. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The block name starts with a number.	Rename the block.
The block name starts with an underscore ("_").	Rename the block.
The block name ends with an underscore ("_").	Rename the block.
The block name contains consecutive underscores.	Rename the block.
The block name has consecutive underscores.	Rename the block.
The block name has blank spaces.	Rename the block.



**Tips**

Use underscores to separate parts of a block name instead of spaces.

**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether signal line names include invalid characters

Check whether signal line names include invalid characters.

### Description

The names of all signal lines are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc\_0221: Usable characters for signal line name.

### 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.

<b>Condition</b>	<b>Recommended Action</b>
The signal line name has blank spaces.	Rename the signal line.
The signal line name has control characters.	Rename the signal line.

**Tip**

Use underscores to separate parts of a signal line name instead of spaces.

**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check whether block names include invalid characters

Check whether block names include invalid characters.

### Description

The names of all blocks are required.

This guideline facilitates

- Readability
- Workflow
- Code generation

This guideline does not apply to subsystem blocks.

See MAAB guideline jc\_0231: Usable characters for signal line name.

### Results and Recommended Actions

Condition	Recommended Action
The block name contains illegal characters.	Rename the block. Allowed characters include a–z, A–Z, 0–9, underscore (_), and period (.).
The block name starts with a number.	Rename the block.
The block name has blank spaces.	Rename the block.
The block name has double byte characters.	Rename the block.

### Prerequisite

A prerequisite MAAB guideline for this check is jc\_0201: Usable characters for Subsystem names.

**Tip**

Carriage returns are allowed in block names.

**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check Trigger and Enable block port names

Check Trigger and Enable block port names.

### Description

Block port names should match the name of the signal triggering the subsystem.

This guideline facilitates

- Readability

See MAAB guideline jc\_0281: Naming of Trigger block and Enable block.

### Results and Recommended Actions

Condition	Recommended Action
Trigger block does not match the name of the signal to which it is connected.	Match Trigger block names to the connecting signal.
Enable block does not match the name of the signal to which it is connected.	Match Enable block names to the connecting signal.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for Simulink diagrams that have nonstandard appearance attributes

Check model appearance setting attributes.

### Description

Model appearance settings are required to conform to the guidelines when the model is released.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline na\_0004: Simulink model appearance.

### Results and Recommended Actions

Condition	Recommended Action
Diagrams do not have white backgrounds.	Select <b>Format &gt; Screen Color &gt; Automatic</b> .
Diagrams do not have zoom factor set to 100%.	Select <b>View &gt; Normal (100%)</b> .
The toolbar is not visible.	Select <b>View &gt; Toolbar</b> .
The status bar is not visible.	Select <b>View &gt; Status Bar</b> .
Block backgrounds are not white.	Blocks should have black foregrounds with white backgrounds. Click the specified block and select <b>Format &gt; Foreground Color &gt; Black</b> and <b>Format &gt; Background Color &gt; White</b> .
<b>Wide Nonscalar Lines</b> is cleared.	Select <b>Format &gt; Port/Signal Displays &gt; Wide Nonscalar Lines</b> .

<b>Condition</b>	<b>Recommended Action</b>
Viewer Indicators is cleared.	Select <b>Format &gt; Port/Signal Displays &gt; Viewer Indicators</b> .
Testpoint Indicators is cleared.	Select <b>Format &gt; Port/Signal Displays &gt; Testpoint Indicators</b> .
Port Data Types is selected.	Clear <b>Format &gt; Port/Signal Displays &gt; Port Data Types</b> .
Storage Class is selected.	Clear <b>Format &gt; Port/Signal Displays &gt; Storage Class</b> .
Signal Dimensions is selected.	Clear <b>Format &gt; Port/Signal Displays &gt; Signal Dimensions</b> .
Model Browser is selected.	Clear <b>View &gt; Model Browser Options &gt; Model Browser</b> .
Sorted Order is selected.	Clear <b>Format &gt; Block Displays &gt; Sorted Order</b> .
Model Block Version is selected.	Clear <b>Format &gt; Block Displays &gt; Model Block Version</b> .
Model Block I/O Mismatch is selected.	Clear <b>Format &gt; Block Displays &gt; Model Block I/O Mismatch</b> .
Execution Context Indicator is selected.	Clear <b>Format &gt; Block Displays &gt; Execution Context Indicator</b> .
Sample Time Colors is selected.	Clear <b>Format &gt; Port/Signal Displays &gt; Sample Time Colors</b> .
Library Link Display is set to User or All.	Select <b>Format &gt; Library Link Display &gt; None</b> .
Linearization Indicators is cleared.	Select <b>Format &gt; Port/Signal Displays &gt; Linearization Indicators</b> .



**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check visibility of port block names

Check the visibility of port block names.

### Description

An organization applying the MAAB guidelines must select one of the following alternatives to enforce:

- The name of port blocks are not hidden.
- The name of port blocks must be hidden.

This guideline facilitates

- Readability

---

**Note** This check does not look in masked subsystems.

---

See MAAB guideline na\_0005: Port block name visibility in Simulink models.

### Input Parameters

#### All Port names should be shown (Format/Show Name)

Select this check box if all ports should show the name, including subsystems.

### Results and Recommended Actions

Condition	Recommended Action
Blocks do not show their name and the <b>All Port names should be shown (Format/Show Name)</b> check box is selected.	Change the format of the specified blocks to show names according to the input requirement.

<b>Condition</b>	<b>Recommended Action</b>
Blocks show their name and the <b>All Port names should be shown (Format/Show Name)</b> 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 <b>Show port labels</b> to a value other than none.
Subsystem blocks show their port names.	Set the subsystem parameter <b>Show port labels</b> to none.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for direction of subsystem blocks

Check the orientation of subsystem blocks.

### Description

Subsystem inputs must be located on the left side of the block, and outputs must be located on the right side of the block.

This guideline facilitates

- Readability

See MAAB guideline jc\_0111: Direction of Subsystem.

### Results and Recommended Actions

Condition	Recommended Action
Subsystem blocks are not in the correct orientation.	Change the subsystem blocks to have the correct orientation, with inports on the left and outports on the right.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for proper position of constants used in Relational Operator blocks

Check the position of Constant blocks used in Relational Operator blocks.

### Description

When the relational operator is used to compare a signal to a constant value, the constant input should be the second, lower input.

This guideline facilitates

- Readability
- Code generation

See MAAB guideline jc\_0131: Use of Relational Operator block.

### Results and Recommended Actions

Condition	Recommended Action
Relational Operator blocks have a Constant block on the first, upper input.	Move the Constant block to the second, lower input.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for entry format in state blocks

Check the format of entries in state blocks.

### Description

A new line should be started after the entry, during, and exit action statements and after the completion of an assignment statement “;”.

This guideline facilitates

- Readability

See MAAB guideline jc\_0501: Format of entries in a state block.

### Results and Recommended Actions

Condition	Recommended Action
An entry in a state block is not formatted correctly.	Validate that the intended design is properly represented in the Stateflow diagram.
An entry action statement is not by itself.	Add a new line.
Multiple entry action statements found on one line.	Add a new line between entry action statements.
An during action statement is not by itself.	Add a new line.
Multiple during action statements found on one line.	Add a new line between during action statements.
An exit action statement is not by itself.	Add a new line.
Multiple exit action statements found on one line.	Add a new line between exit action statements.

<b>Condition</b>	<b>Recommended Action</b>
Multiple action statements found on one line.	Add a new line between action statements.
Potential misuse of semicolon (;) on a line.	Correct the use of the semicolon where specified.

**See Also**

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for use of tunable parameters in Stateflow

Check for use of tunable parameters in Stateflow charts.

### Description

Include tunable parameters in a Stateflow chart as inputs from the Simulink model.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline jc\_0541: Use of tunable parameters in Stateflow.

### Results and Recommended Actions

Condition	Recommended Action
Stateflow charts reference Simulink data objects, which should be used as inputs from the Simulink model.	Make the Simulink data objects inputs from the Simulink model to the specified Stateflow chart.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*



## Check for proper use of Switch blocks

Check for proper use of Switch blocks.

### Description

This check verifies that the Switch block's control input (the second input) is a Boolean value and that the block is configured to pass the first input when the control input is nonzero.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline jc\_0141: Use of the Switch block.

### Results and Recommended Actions

Condition	Recommended Action
The Switch block's control input (second input) is not a Boolean value.	Change the data type of the control input to Boolean.
The Switch block is not configured to pass the first input when the control input is nonzero.	Set the block parameter <b>Criteria for passing first input</b> to <code>u2 ~=0</code> .

### See Also

- See the description of the Switch block in the Simulink documentation.
- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for proper use of signal buses and Mux block usage

Check for proper use of signal busses and Mux block usage.

### Description

This check verifies whether a model is using signal buses and Mux blocks properly.

This guideline facilitates

- Readability
- Workflow

See MAAB guideline na\_0010: Grouping data flows into signals.

### Results and Recommended Actions

Condition	Recommended Action
The individual scalar input signals for a Mux block do not have common functionality, data types, dimensions, and units.	Modify the scalar input signals such that the specifications match.
The output of a Mux block is not a vector.	Change the output of the Mux block to a vector.
All inputs to a Mux block are not scalars.	Make sure that all input signals to Mux blocks are scalars.
The input for a Bus Selector block is not a bus signal.	Make sure that the input for all Bus Selector blocks is a bus signal.

### See Also

- Using Composite Signals in the Simulink documentation.

- The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for mismatches between Stateflow ports and associated signal names

Check for mismatches between Stateflow ports and associated signal names.

### Description

The name of Stateflow input and output should be the same as the corresponding signal. This guideline is required for:

- Readability
- Workflow

See MAAB guideline db\_0123: Stateflow port names.

### Results and Recommended Actions

Condition	Recommended Action
Signals have names that differ from those of their corresponding Stateflow ports.	Change the names of either the signals or the Stateflow ports.

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Check for proper scope of From and Goto blocks

Check the scope of From and Goto blocks.

### Description

You can use global scope for controlling flow. However, From and Goto blocks must use local scope for signal flows.

This guideline facilitates

- Readability
- Workflow
- Code generation

See MAAB guideline na\_0011: Scope of Goto and From blocks.

### Results and Recommended Actions

Condition	Recommended Action
From and Goto blocks are not configured with local scope.	<ul style="list-style-type: none"> <li>• Make sure the ports are connected correctly.</li> <li>• Change the scope of the specified blocks to local.</li> </ul>

### See Also

The MathWorks Automotive Advisory Board, which lists downloads for the latest version of *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*

## Requirements Consistency Checks

In this section...
“Identify requirement links with missing documents” on page 10-141
“Identify requirement links that specify invalid locations within documents” on page 10-142
“Identify selection-based links having descriptions that do not match their requirements document text” on page 10-143
“Identify requirement links with inconsistent path types and preferences” on page 10-144

## Identify requirement links with missing documents

Ensure that requirements link to existing documents.

### Description

You used the Requirements Management Interface (RMI) to associate a design requirements document with a part of your model design and the interface cannot find the specified document.

### Results and Recommended Actions

Condition	Recommended Action
The requirements document associated with a part of your model design is not accessible at the specified location.	Open the Requirements dialog box and correct the path name of the requirements document or move the document to the specified location.

### See Also

“Adding and Viewing Requirement Links” on page 2-4

## **Identify requirement links that specify invalid locations within documents**

Ensure that requirements link to valid locations (e.g., bookmarks, line numbers, anchors) within documents.

### **Description**

You used the Requirements Management Interface (RMI) to associate a location in a design requirements document (a bookmark, line number, or anchor) with a part of your model design and the interface cannot find the specified location in the specified document.

### **Results and Recommended Actions**

<b>Condition</b>	<b>Recommended Action</b>
The location in the requirements document associated with a part of your model design is not accessible.	Open the Requirements dialog box and correct the location reference within the requirements document.

### **See Also**

“Adding and Viewing Requirement Links” on page 2-4



## Identify selection-based links having descriptions that do not match their requirements document text

Ensure that descriptions of selection-based links use the same text found in their requirements documents.

### Description

You used selection-based linking of the Requirements Management Interface (RMI) to label requirements in the model's **Requirements** menu with text that appears in the corresponding requirements document. This check helps you manage traceability by identifying requirement descriptions in the menu that are not synchronized with text in the documents.

### Results and Recommended Actions

Condition	Recommended Action
Selection-based links have descriptions that differ from their corresponding selections in the requirements documents.	If the difference reflects a change in the requirements document, click the link in the Model Advisor results to replace the current description in the selection-based link with the text from the requirements document (the external description). Alternatively, you can right click the object in the model window, select <b>Edit/Add Links</b> from the <b>Requirements</b> menu, and use the Requirements dialog box that appears to synchronize the text.

### See Also

“Selection-Based Linking” on page 2-22

## 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 <b>Selection-based linking</b> 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"> <li>• Click <b>Fix</b> to change the current path to the valid path.</li> <li>• Update the preference in the <b>Selection-based linking</b> dialog box. In the model window, select <b>Tools &gt; Requirements &gt; Link settings</b> and change the value for the <b>Document file reference</b> option.</li> </ul>

### See Also

“Selection-Based Linking” on page 2-22

# Examples

---

Use this list to find examples in the documentation.

## **Requirements Management Interface**

- “Adding Requirement Links to an Object” on page 2-7
- “Viewing Requirements Documents” on page 2-14
- “Making Selection-Based Links” on page 2-23
- “Creating a Custom Link Requirement Type” on page 2-32
- “Viewing Objects with Requirement Links” on page 2-45
- “Generating a Requirements Report” on page 2-48
- “Displaying the System Requirements in a Diagram” on page 2-50
- “Including Requirements with Generated Code” on page 2-56

## **Requirements Management Interface (DOORS Version)**

- “Linking Objects to DOORS Requirements” on page 3-9
- “Synchronizing a Model with the DOORS Software” on page 3-16
- “Navigating from a Simulink Model to DOORS Requirements” on page 3-28
- “Navigating from a DOORS Requirements to the Simulink Model” on page 3-30

## **Verification Manager**

- “Opening the Verification Manager” on page 4-7
- “Enabling and Disabling Model Verification Blocks with the Verification Manager” on page 4-15
- “Using Enabling and Disabling Tools in the Verification Manager” on page 4-20
- “Managing Verification Requirements” on page 4-24

## **Model Coverage**

- “Details Report Section” on page 5-28
- “Decisions Analyzed Table” on page 5-30
- “Conditions Analyzed Table” on page 5-31

- “MC/DC Analysis Table” on page 5-31
- “N-Dimensional Lookup Table Report” on page 5-36
- “Signal Range Analysis Report” on page 5-43
- “Displaying Model Coverage with Model Coloring” on page 5-48
- “Creating a Model with Embedded MATLAB Function Block Decisions”  
on page 5-65
- “Understanding Embedded MATLAB Function Block Model Coverage”  
on page 5-69



## A

- adding links to requirements 2-7
- adding requirements 2-4
- Assertion block appearance 4-19

## C

- categorical lists of functions 7-1
- changing links to requirements 2-12
- closing Signal Builder Requirements pane 4-13
- colored diagram model coverage display 5-47
  - enabling 5-47
- condition coverage
  - Embedded MATLAB Function blocks 5-79
  - statements in Embedded MATLAB Function block 5-65
- configuring MATLAB
  - for DOORS version 3-6
- customizing Model Advisor 6-1
- cv.cvdatagroup function
  - reference 7-3 8-5
- cv.cvtestgroup function
  - reference 7-3 8-7
- cvexit function
  - reference 7-3 8-9
- cvhtml function
  - model coverage 5-55
  - reference 7-3 8-10
- cvload function
  - model coverage 5-56
  - reference 7-3 8-12
- cvsave function
  - model coverage 5-56
  - reference 7-3 8-14
- cvsim function
  - model coverage 5-54
  - reference 7-3 8-15
- cvsimref function
  - reference 7-3 8-17

- cvtest function
  - model coverage 5-52
  - reference 7-3 8-19

## D

- decision coverage
  - Embedded MATLAB Function blocks 5-78
  - statements in Embedded MATLAB Function blocks 5-64
- defining Model Advisor checks 6-17
- defining Model Advisor tasks 6-41
- demos
  - Model Advisor customization demo 6-6
  - simcovdemo model coverage demo 5-7
- disabling Model Verification blocks across test groups 4-20
- DOORS
  - additional installation for 3-3
  - starting 3-6
- DOORS Requirements Management Interface
  - block type descriptions 3-18
  - definition for object 3-15
  - from Simulink to DOORS 3-28
  - hierarchical numbers 3-18
  - naming of surrogate exported module 3-18
  - object identifiers 3-18
  - opening the object in Simulink, Stateflow, or MATLAB 3-32
  - overview 3-2
  - saving formal modules 3-21
  - starting MATLAB for 3-6
  - synchronizing models with DOORS 3-16
  - synchronizing objects with DOORS formal module 3-16
  - viewing model elements with requirements 3-26
  - viewing requirements 3-26

**E**

- Embedded MATLAB Function blocks
  - condition coverage 5-79
  - condition coverage statements 5-65
  - decision coverage 5-78
  - decision coverage statements 5-64
  - MCDC coverage 5-79
  - MCDC coverage statements 5-65
  - model coverage 5-64
  - model coverage example 5-65
  - types of model coverage 5-64
- enabling Model Verification blocks across test groups 4-20

**F**

- functions
  - categories 7-1
  - cv.cvdatagroup 7-3 8-5
  - cv.cvtestgroup 7-3 8-7
  - cvexit 7-3 8-9
  - cvhtml 7-3 8-10
  - cvload 7-3 8-12
  - cvsave 7-3 8-14
  - cvsim 7-3 8-15
  - cvsimref 7-3 8-17
  - cvtest 7-3 8-19
  - Model Advisor customization API 7-4
  - Model Advisor formatting API 7-5
  - model coverage 7-3
  - rminav 7-2 8-81
  - start old Requirements Management Interface 7-2

**I**

- icons for Model Verification blocks in Verification Manager 4-16
- IEC 61508
  - Model Advisor checks 10-57

- installing DOORS 3-3

**L**

- linking model objects to requirements 2-7
- Lookup Table block in model coverage report 5-36
- Lookup Table model coverage
  - n-dimensional 5-42
  - three-dimensional example 5-39
  - two-dimensional example 5-36

**M**

- MathWorks Automotive Advisory Board
  - Model Advisor checks 10-73
- MCDC coverage
  - Embedded MATLAB Function blocks 5-79
  - statements in Embedded MATLAB Function blocks 5-65
- MCDC table
  - condition cases 5-32
- Model Advisor checks
  - IEC 61508 10-57
  - MathWorks Automotive Advisory Board 10-73
  - requirements consistency 10-140
- Model Advisor customization API functions 7-4
- Model Advisor customizations
  - creating check callback functions 6-10
  - defining custom checks 6-17
  - defining custom tasks 6-41
  - defining process callback functions 6-45
  - formatting Model Advisor outputs 6-48
  - registering custom checks and tasks 6-7
  - slvnvdemo\_mdadv demo 6-6
  - workflow overview 6-3
- Model Advisor formatting API functions 7-5



- model coverage
    - colored Simulink diagram display 5-47
    - colored Simulink diagram example 5-48
    - commands in MATLAB 5-52
    - Conditions analyzed table 5-31
    - Decisions analyzed table 5-30
    - Details report section 5-28
    - Embedded MATLAB Function blocks 5-64
    - enabling colored diagram display 5-47
    - enabling colored Simulink diagram display 5-47
    - HTML settings 5-19
    - introduction 5-2
    - Lookup Table block report 5-36
    - MCDC table 5-32
    - n-dimensional Lookup Table 5-42
    - settings in dialog 5-11
    - signal range analysis report 5-43
    - Summary report section 5-27
    - three-dimensional Lookup Table example 5-39
    - two-dimensional Lookup Table 5-36
    - understanding report 5-25
    - workflow 5-7
  - model coverage demo
    - simcovdemo 5-7
  - model coverage functions 7-3
    - cvhtml 5-55
    - cvload 5-56
    - cvsave 5-56
    - cvsim 5-54
    - cvtest 5-52
  - Model Verification blocks
    - block appearance 4-17
    - disabling for test groups 4-15
    - enabling for test groups 4-15
    - icons 4-16
    - parameter settings 4-3
    - using individually 4-2
  - models
    - running test cases 5-7
  - modifying requirements 2-4
- O**
- objects
    - linking model objects to requirements 2-7
    - viewing objects with requirements 2-45
  - old Requirements Management Interface 7-2
  - opening a Signal Builder block 4-9
  - operating system requirements 1-3
- P**
- parameters for Model Verification blocks 4-3
- R**
- report
    - model coverage HTML options 5-19
    - understanding model coverage report 5-25
  - requirements
    - adding 2-4
    - adding to test groups 4-25
    - for Model Verification block settings 4-24
    - for Requirements Management Interface for DOORS 3-2
    - in generated code 2-56
    - linking to model objects 2-7
    - modifying 2-4
    - viewing 2-4
    - viewing for test groups 4-27
    - viewing objects with 2-45
  - requirements consistency
    - Model Advisor checks 10-140
  - requirements documents
    - editing 2-14
    - viewing 2-14
  - requirements links
    - editing 2-12

- Requirements Management Interface
  - overview 2-2
- Requirements Management Interface for DOORS
  - block type descriptions 3-18
  - definition of object in DOORS 3-15
  - from Simulink to DOORS 3-28
  - hierarchical numbers 3-18
  - naming of surrogate exported modules 3-18
  - object identifiers 3-18
  - opening the object in Simulink or Stateflow 3-32
  - overview 3-2
  - saving formal modules 3-21
  - starting 3-6
  - starting MATLAB for 3-6
  - synchronizing models with DOORS 3-16
  - synchronizing objects with DOORS formal module 3-16
  - viewing model elements with requirements 3-26
  - viewing requirements 3-26
- Requirements pane for Verification Manager 4-24
- rminav function
  - reference 7-2 8-81
- 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 5-43
- simcovdemo
  - model coverage demo 5-7
- slvncdemo\_mdadv
  - Model Advisor customization demo 6-6
- starting DOORS 3-6
- starting MATLAB for DOORS 3-6
- starting Requirements Management Interface for DOORS 3-6
- Summary section of model coverage report 5-27
- synchronizing models with DOORS 3-16
- system requirements 1-3
  - MATLAB 1-3
  - Microsoft Excel 1-3
  - Microsoft Word 1-3
  - operating system 1-3
  - Simulink 1-3
  - Stateflow 1-3
  - Telelogic DOORS 1-3
- T**
- test case commands 5-7
- test groups
  - adding requirements 4-25
  - disabling Model Verification blocks 4-15
  - enabling Model Verification blocks 4-15
  - Model Verification blocks enabled across 4-20
- V**
- verification blocks
  - example of use 4-2
  - icons 4-16
  - requirements for test groups 4-24
  - stopping simulation 4-4

- Verification Manager
  - closing Requirements pane 4-13
  - disabling Model Verification blocks for test groups 4-15
  - enabled/disabled block appearance 4-17
  - enabling Model Verification blocks for test groups 4-15
  - flat display 4-15
  - hierarchical display 4-15
  - icons for Model Verification blocks 4-16
  - opening 4-7
  - Requirements pane 4-24
  - viewing objects with requirements 2-45
  - viewing requirements 2-4