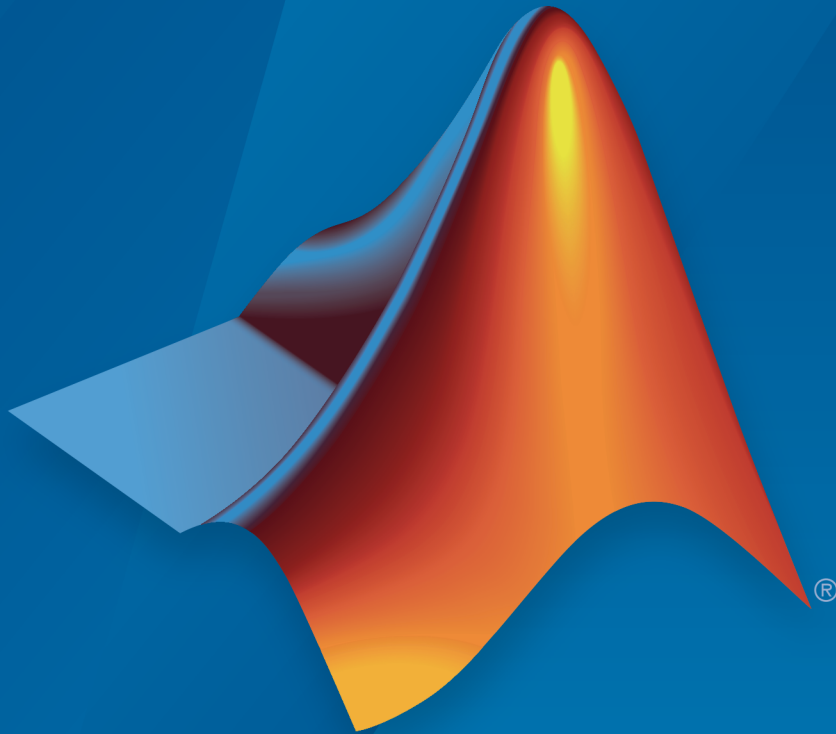


Simulink® Verification and Validation™

User's Guide



MATLAB® & SIMULINK®

R2016b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Simulink[®] Verification and Validation[™] User's Guide

© COPYRIGHT 2004–2016 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

June 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Online only	Revised for Version 1.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.0.2 (Release 14SP2)
April 2005	Second printing	Revised for Version 1.1 (Web release)
September 2005	Online only	Revised for Version 1.1.1 (Release 14SP3)
March 2006	Online only	Revised for Version 1.1.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.0 (Release 2006b)
March 2007	Online only	Revised for Version 2.1 (Release 2007a)
September 2007	Online only	Revised for Version 2.2 (Release 2007b)
March 2008	Online only	Revised for Version 2.3 (Release 2008a)
October 2008	Online only	Revised for Version 2.4 (Release 2008b)
March 2009	Online only	Revised for Version 2.5 (Release 2009a)
September 2009	Online only	Revised for Version 2.6 (Release 2009b)
March 2010	Online only	Revised for Version 2.7 (Release 2010a)
September 2010	Online only	Revised for Version 3.0 (Release 2010b)
April 2011	Online only	Revised for Version 3.1 (Release 2011a)
September 2011	Online only	Revised for Version 3.2 (Release 2011b)
March 2012	Online only	Revised for Version 3.3 (Release 2012a)
September 2012	Online only	Revised for Version 3.4 (Release 2012b)
March 2013	Online only	Revised for Version 3.5 (Release 2013a)
September 2013	Online only	Revised for Version 3.6 (Release 2013b)
March 2014	Online only	Revised for Version 3.7 (Release 2014a)
October 2014	Online only	Revised for Version 3.8 (Release 2014b)
March 2015	Online only	Revised for Version 3.9 (Release 2015a)
September 2015	Online only	Revised for Version 3.10 (Release 2015b)
October 2015	Online only	Rereleased for Version 3.9.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.11 (Release 2016a)
September 2016	Online only	Revised for Version 3.12 (Release 2016b)

Getting Started

1

Simulink Verification and Validation Product Description .	1-2
Key Features	1-2
System Requirements	1-3
Operating System Requirements	1-3
Product Requirements	1-3

Verification and Validation

2

Test Model Against Requirements and Report Results	2-2
Requirements Overview	2-2
Test a Cruise Control Safety Requirement	2-2
Analyze a Model for Standards Compliance and Design Errors	2-6
Standards and Analysis Overview	2-6
Check Model for Style Guideline Violations and Design Errors	2-6
Perform Functional Testing and Analyze Test Coverage ...	2-9
Functional Testing and Coverage Analysis Overview	2-9
Incrementally Increase Test Coverage Using Test Case Generation	2-9
Analyze Code and Test Software-in-the-Loop	2-16
Code Analysis and Testing Software-in-the-Loop Overview .	2-16
Analyze Code for Defects, Metrics, and MISRA C:2012	2-16

Module Verification and Testing Processor-in-the-Loop . .	2-25
Module Verification and Testing Processor-in-the-Loop	
Overview	2-25
Test a Model in Real Time	2-26
Real-Time Testing and Testing Production Models Overview	2-26

Requirements Traceability

Links Between Models and Requirements

3

Overview of the Requirements Management Interface (RMI)	3-3
Requirements Traceability Links	3-4
Requirements Link Storage	3-5
Supported Requirements Document Types	3-6
Supported Model Objects for Requirements Linking . .	3-9
Selection-Based Linking	3-10
Link to Requirements Document Using Selection-Based Linking	3-11
Link Test Cases to Requirements Documents	3-12
Establish Requirements Traceability for Testing	3-12
Configure RMI for IBM Rational DOORS or Microsoft ActiveX Navigation	3-16
Requirements Traceability Link Editor	3-17
Manage Requirements Traceability Links Using Link Editor	3-17

Requirements Tab	3-19
Document Index Tab	3-19
Requirements Settings	3-20
Selection Linking Tab	3-20
Link Model Objects	3-22
Link Objects in the Same Model	3-22
Link Objects in Different Models	3-22
Link from External Applications	3-24
Link Multiple Model Objects to a Requirements	
Document	3-25
Link Multiple Model Objects to a Requirement Document	
Using a Simulink DocBlock	3-28
Link to Requirements in Microsoft Word Documents .	3-30
Create Bookmarks in a Microsoft Word Requirements	
Document	3-30
Open the Example Model and Associated Requirements	
Document	3-32
Create a Link from a Model Object to a Microsoft Word	
Requirements Document	3-32
Link to Requirements in IBM Rational DOORS	
Databases	3-36
Link to Requirements in Microsoft Excel Workbooks .	3-38
Navigate from a Model Object to Requirements in a	
Microsoft Excel Workbook	3-38
Create Requirements Links to the Workbook	3-38
Link Multiple Model Objects to a Microsoft Excel	
Workbook	3-39
Change Requirements Links	3-40
Link to Requirements in MuPAD Notebooks	3-43
Create Requirements Traceability Report for Model .	3-46
If Your Model Has Library Reference Blocks	3-48
If Your Model Has Model Reference Blocks	3-48
Link to Requirements Modeled in Simulink	3-49

Requirements Linking with Simulink Annotations . . .	3-58
Link Signal Builder Blocks to Requirements Documents	3-59
Link Signal Builder Blocks to Model Objects	3-61
Link Requirements to Simulink Data Dictionary Entries	3-63

How Is Requirements Link Information Stored?

4

External Storage	4-2
Guidelines for External Storage of Requirements Links	4-3
Specify Storage for Requirements Links	4-4
Default storage mode for traceability data	4-4
Duplicating outgoing links when copying Simulink and Stateflow objects	4-4
Save Requirements Links in External Storage	4-5
Load Requirements Links from External Storage	4-6
Move Internally Stored Requirements Links to External Storage	4-7
Move Externally Stored Requirements Links to the Model File	4-8

5

Highlight Model Objects with Requirements	5-2
Highlight Model Objects with Requirements Using Model Editor	5-2
Highlight Model Objects with Requirements Using Model Explorer	5-3
Navigate to Requirements from Model	5-5
Navigate from Model Object	5-5
Navigate from System Requirements Block	5-5
Customize Requirements Traceability Report for Model	5-7
Create Default Requirements Report	5-7
Report for Requirements in Model Blocks	5-15
Customize Requirements Report	5-17
Generate Requirements Reports Using Simulink	5-23
Filter Requirements with User Tags	5-25
User Tags and Requirements Filtering	5-25
Apply a User Tag to a Requirement	5-25
Filter, Highlight, and Report with User Tags	5-27
Apply User Tags During Selection-Based Linking	5-29
Configure Requirements Filtering	5-31
Create Requirements Traceability Report for Simulink Project	5-33
View Requirements Details for a Selected Block	5-34
Requirements Details Workflow	5-34
Requirements Details Limitations	5-35

Requirements Links Maintenance

6

Validation of Requirements Links	6-2
When to Check Links in a Requirements Document	6-2

How the rmi Function Checks a Requirements Document	6-3
Validate Requirements Links in a Model	6-4
Check Requirements Links with the Model Advisor	6-4
Fix Invalid Requirements Links Detected by the Model Advisor	6-6
Validate Requirements Links in a Requirements Document	6-10
Check Links in a Requirements Document	6-10
When Multiple Objects Have Links to the Same Requirement	6-11
Fix Invalid Links in a Requirements Document	6-12
Document Path Storage	6-14
Relative (Partial) Path Example	6-14
Relative (No) Path Example	6-14
Absolute Path Example	6-15
Delete Requirements Links from Simulink Objects ..	6-16
Delete a Single Link from a Simulink Object	6-16
Delete All Links from a Simulink Object	6-16
Delete All Links from Multiple Simulink Objects	6-17
Requirements Links for Library Blocks and Reference Blocks	6-18
Introduction to Library Blocks and Reference Blocks ..	6-18
Library Blocks and Requirements	6-18
Copy Library Blocks with Requirements	6-19
Manage Requirements on Reference Blocks	6-19
Manage Requirements Inside Reference Blocks	6-20
Links from Requirements to Library Blocks	6-23

IBM Rational DOORS Surrogate Module Synchronization

7

Synchronization with DOORS Surrogate Modules	7-2
--	-----

Advantages of Synchronizing Your Model with a Surrogate Module	7-4
Synchronize a Simulink Model to Create a Surrogate Module	7-5
Create Links Between Surrogate Module and Formal Module in a DOORS Database	7-7
Customize DOORS Synchronization	7-8
DOORS Synchronization Settings	7-8
Resynchronize a Model with a Different Surrogate Module	7-10
Customize the Level of Detail in Synchronization	7-11
Resynchronize to Include All Simulink Objects	7-12
Resynchronize DOORS Surrogate Module to Reflect Model Changes	7-15
Navigate with the Surrogate Module	7-17
Navigate Between Requirements and the Surrogate Module in the DOORS Database	7-17
Navigate Between DOORS Requirements and the Simulink Module via the Surrogate Module	7-18

Navigation from Requirements Documents

8

Why Add Navigation Objects to DOORS Requirements?	8-2
Configure Requirements Management Interface for DOORS Software	8-3
Before You Begin	8-3
Manually Install Additional Files for DOORS Software	8-3
Enable Linking from DOORS Databases to Simulink Objects	8-5

Insert Navigation Objects into DOORS Requirements .	8-7
Insert Navigation Objects to Multiple Simulink Objects .	8-8
Customize DOORS Navigation Objects	8-9
Navigate Between DOORS Requirement and Model Object	8-10
Diagnose and Fix DXL Errors	8-12
Why Add Navigation Objects to Microsoft Office Requirements?	8-13
Enable Linking from Microsoft Office Documents to Simulink Objects	8-14
Insert Navigation Objects in Microsoft Office Requirements Documents	8-16
Insert Navigation Object That Links to Multiple Simulink Objects	8-16
Customize Microsoft Office Navigation Objects	8-18
Navigate Between Microsoft Word Requirement and Model	8-19
Navigate with Objects Created Using ActiveX in Microsoft Office 2007 and 2010	8-21
Save Requirements Documents to Microsoft Word 2007 or 2010 Format	8-21
Field Codes in Requirements Documents	8-22
ActiveX Control Does Not Link to Model Object	8-24
Delete an ActiveX Control from Microsoft Excel 2007 .	8-26
Delete an ActiveX Control from Microsoft Excel 2010 .	8-27

9

Link Between MATLAB Code Lines and Requirements	
Information in External Documents	9-2
Create Link Using Context Menu Shortcuts	9-2
Create Link Using Link Editor	9-2
Enable or Disable Highlighting of Traceability Links for MATLAB Code	
Enable Traceability Highlighting of MATLAB Code ...	9-4
Disable Traceability Highlighting of MATLAB Code ...	9-4
Remove Traceability Links from MATLAB Code Lines .	
Delete Links to Requirements from MATLAB Code Lines	9-5
Delete Link Targets in MATLAB Code Lines	9-5
Traceability for MATLAB Code Lines	
Traceability Link Targets	9-6
Storage of Traceability Links	9-6
Limitations of MATLAB Code Traceability	9-7
Associate Traceability Information with MATLAB Code Lines in Simulink	
	9-8

Custom Types of Requirements Documents

10

Why Create a Custom Link Type?	10-2
Implement Custom Link Types	10-3
Custom Link Type Functions	10-4
Links and Link Types	10-5
Link Type Properties	10-6

Custom Link Type Registration	10-10
Create a Custom Requirements Link Type	10-11
Create a Document Index	10-17
Custom Link Type Synchronization	10-19
Navigate to Simulink Objects from External Documents	10-21
Provide Unique Object Identifiers	10-21
Use the rmiobjnavigate Function	10-21
Determine the Navigation Command	10-21
Use the ActiveX Navigation Control	10-22
Typical Code Sequence for Establishing Navigation Controls	10-22

Requirements Information in Generated Code

11

How Requirements Information Is Included in Generated Code	11-2
Generate Code for Models with Requirements Links .	11-4

Model Component Testing

Overview of Component Verification

12

Component Verification	12-2
Component Verification Approaches	12-2
Simulink Verification and Validation Tools for Component Verification	12-2

Basic Approach to Component Verification	12-4
Workflow for Component Verification	12-4
Verify a Component Independently of the Container Model	12-6
Verify a Model Block in the Context of the Container Model	12-7
Functions for Component Verification	12-8

Verifying Generated Code for a Component

13

Verify Generated Code for a Component	13-2
About the Example Model	13-2
Prepare the Component for Verification	13-4
Create and Log Test Cases	13-6
Merge Test Case Data	13-7
Record Coverage for Component	13-7
Execute Component in Simulation Mode	13-8
Execute Component in SIL Mode	13-9

Signal Monitoring with Model Verification Blocks

Using Model Verification Blocks

14

Model Verification Blocks and the Verification Manager	14-2
Use Check Static Lower Bound Block to Check for Out- of-Bounds Signal	14-3
Linear System Modeling Blocks in Simulink Control Design	14-6

Constructing Simulation Tests Using the Verification Manager

15

What Is the Verification Manager?	15-2
Construct Simulation Tests Using the Verification Manager	15-3
View Model Verification Blocks	15-3
Enable and Disable Model Verification Blocks in a Model	15-9
Enable and Disable Model Verification Blocks in a Subsystem	15-12
Use Check Static Lower Bound Block to Check for Out-of-Bounds Signal	15-16
Link Test Cases to Requirements Documents Using the Verification Manager	15-19

Model Coverage Analysis

Model Coverage Definition

16

Model Coverage	16-2
Types of Model Coverage	16-3
Execution Coverage (EC)	16-3
Decision Coverage (DC)	16-3
Condition Coverage (CC)	16-3
Modified Condition/Decision Coverage (MCDC)	16-4
Cyclomatic Complexity	16-5
Lookup Table Coverage	16-5
Signal Range Coverage	16-6
Signal Size Coverage	16-6
Objectives and Constraints Coverage	16-7
Saturate on Integer Overflow Coverage	16-8
Relational Boundary Coverage	16-8

Simulink Optimizations and Model Coverage	16-11
Inlined parameters	16-11
Block reduction	16-11
Conditional input branch execution	16-12

17

Model Objects That Receive Model Coverage

Model Objects That Receive Coverage	17-2
Abs	17-8
Bias	17-9
Combinatorial Logic	17-9
Compare to Constant	17-9
Compare to Zero	17-10
Data Type Conversion	17-10
Dead Zone	17-10
Direct Lookup Table (n-D)	17-11
Discrete Filter	17-12
Discrete FIR Filter	17-12
Discrete-Time Integrator	17-12
Discrete Transfer Fcn	17-13
Dot Product	17-13
Enabled Subsystem	17-13
Enabled and Triggered Subsystem	17-14
Fcn	17-15
For Iterator, For Iterator Subsystem	17-16
Gain	17-16
If, If Action Subsystem	17-16
Interpolation Using Prelookup	17-17
Library-Linked Objects	17-18
Logical Operator	17-18
1-D Lookup Table	17-18
2-D Lookup Table	17-19
n-D Lookup Table	17-20
Math Function	17-20
MATLAB Function	17-20
MATLAB System	17-21
MinMax	17-21
Model	17-21
Multiport Switch	17-22
PID Controller, PID Controller (2 DOF)	17-22

Product	17-23
Proof Assumption	17-23
Proof Objective	17-23
Rate Limiter	17-23
Relational Operator	17-24
Relay	17-25
C/C++ S-Function	17-25
Saturation	17-27
Saturation Dynamic	17-27
Simulink Design Verifier Functions in MATLAB Function Blocks	17-28
Sqrt, Signed Sqrt, Reciprocal Sqrt	17-28
Sum, Add, Subtract, Sum of Elements	17-28
Switch	17-28
SwitchCase, SwitchCase Action Subsystem	17-29
Test Condition	17-29
Test Objective	17-30
Triggered Models	17-30
Triggered Subsystem	17-31
Truth Table	17-32
Unary Minus	17-32
Weighted Sample Time Math	17-32
While Iterator, While Iterator Subsystem	17-32
Model Objects That Do Not Receive Coverage	17-33

Setting Model Coverage Options

18

Specify Model Coverage Options	18-2
Coverage Pane	18-2
Results Pane	18-9
Access, Manage, and Accumulate Coverage Results	18-12
Accessing Coverage Data from the Results Explorer	18-12
Managing Coverage Data from the Results Explorer	18-19
Accumulating Coverage Data from the Results Explorer	18-19
Cumulative Coverage Data	18-22

Types of Code Coverage	19-2
Statement Coverage for Code Coverage	19-2
Condition Coverage for Code Coverage	19-3
Decision Coverage for Code Coverage	19-3
Modified Condition/Decision Coverage (MCDC) for Code Coverage	19-4
Cyclomatic Complexity for Code Coverage	19-4
Relational Boundary for Code Coverage	19-5
Code Coverage for Models in Software-in-the-Loop (SIL)	
Mode and Processor-in-the-Loop (PIL) Mode	19-6
Requirements to Enable SIL or PIL Code Coverage for a Model	19-6
Conditions for Simulink Verification and Validation Code Coverage Measurement	19-7
Reviewing the Coverage Results for Models in SIL or PIL Mode	19-7
Limitations	19-9

Coverage Collection During Simulation

Model Coverage Collection Workflow	20-2
Create and Run Test Cases	20-3
Modified Condition and Decision Coverage in Simulink Design Verifier	20-4
MCDC Definitions for Simulink Verification and Validation and Simulink Design Verifier	20-4
Cascaded Logic Blocks in Simulink Verification and Validation and Simulink Design Verifier	20-6
View Coverage Results in a Model	20-7
Overview of Model Coverage Highlighting	20-7
Enable Coverage Highlighting	20-8

View Results in Coverage Display Window	20-11
Model Coverage for Multiple Instances of a Referenced Model	20-13
About Coverage for Model Blocks	20-13
Record Coverage for Multiple Instances of a Referenced Model	20-13
Model Coverage for MATLAB Functions	20-23
About Model Coverage for MATLAB Functions	20-23
Types of Model Coverage for MATLAB Functions	20-23
How to Collect Coverage for MATLAB Functions	20-25
Examples: Model Coverage for MATLAB Functions	20-26
Model Coverage for C and C++ S-Functions	20-40
Make S-Function Compatible with Model Coverage	20-40
Generate Coverage Report for S-Function	20-41
View Coverage Results for C/C++ Code in S-Function Blocks	20-43
Model Coverage for Stateflow Charts	20-48
How Model Coverage Reports Work for Stateflow Charts	20-48
Specify Coverage Report Settings for Stateflow Charts	20-49
Cyclomatic Complexity for Stateflow Charts	20-49
Decision Coverage for Stateflow Charts	20-50
Condition Coverage for Stateflow Charts	20-53
MCDC Coverage for Stateflow Charts	20-54
Relational Boundary Coverage for Stateflow Charts	20-54
Simulink Design Verifier Coverage for Stateflow Charts	20-54
Model Coverage Reports for Stateflow Charts	20-56
Model Coverage for Stateflow State Transition Tables	20-65
Model Coverage for Stateflow Atomic Subcharts	20-66
Model Coverage for Stateflow Truth Tables	20-69
Colored Stateflow Chart Coverage Display	20-74

21

Types of Coverage Reports	21-2
Model Summary Report	21-3
Model Reference Coverage Report	21-4
External MATLAB File Coverage Report	21-5
Subsystem Coverage Report	21-9
Code Coverage Report	21-11
Top-Level Model Coverage Report	21-12
Coverage Summary	21-12
Details	21-14
Cyclomatic Complexity	21-22
Decisions Analyzed	21-24
Conditions Analyzed	21-25
MCDC Analysis	21-26
Cumulative Coverage	21-27
N-Dimensional Lookup Table	21-30
Block Reduction	21-36
Relational Boundary	21-37
Saturate on Integer Overflow Analysis	21-41
Signal Range Analysis	21-42
Signal Size Coverage for Variable-Dimension Signals	21-44
Simulink Design Verifier Coverage	21-45
Export Model Coverage Web View	21-47

Excluding Model Objects from Coverage

22

Coverage Filtering	22-2
When to Use Coverage Filtering	22-2
What Is Coverage Filtering?	22-2
Coverage Filter Rules and Files	22-4
What Is a Coverage Filter Rule?	22-4
What Is a Coverage Filter File?	22-4

Model Objects to Filter from Coverage	22-6
Create, Edit, and View Coverage Filter Rules	22-7
Create and Edit Coverage Filter Rules	22-7
Save Coverage Filter to File	22-9
Load Coverage Filter File	22-10
Update the Report with the Current Filter Settings ..	22-10
View Coverage Filter Rules in Your Model	22-10
View Coverage Filter Rules in Your Model	22-11
Coverage Filter Viewer	22-12

Automating Model Coverage Tasks

23

Commands for Automating Model Coverage Tasks ...	23-2
Create Tests with cvtest	23-3
Run Tests with cvsim	23-5
Retrieve Coverage Details from Results	23-7
Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs	23-8
Create HTML Reports with cvhtml	23-11
Save Test Runs to File with cvsave	23-12
Load Stored Coverage Test Results with cvload	23-13
cvload Special Considerations	23-13
Use Coverage Commands in a Script	23-14

Check for Compliance in Model and Subsystems	24-2
Check Your Model Interactively	24-2
Check Your Model While You Edit	24-3
Transform Model to Variant System	24-6
Example Model	24-7
Identify Blocks That Qualify for Variant Transformation	24-8
Perform Variant Transformation	24-10
Variant Transformation Limitations	24-12
Model Transformer Limitations	24-12
Enable Component Reuse with Clone Detection	24-14
Identify Subsystem Clones	24-14
Replace Subsystem Clones with Links to Library Blocks	24-16
Model Transformer Limitations	24-12
Limit Model Checks	24-19
What Is a Model Advisor Exclusion?	24-19
Save Model Advisor Exclusions in a Model File	24-20
Save Model Advisor Exclusions in Exclusion File	24-21
Create Model Advisor Exclusions	24-21
Review Model Advisor Exclusions	24-23
Manage Exclusions	24-24
Limit Model Checks By Excluding Gain and Output Blocks	24-27
Model Checks for DO-178C/DO-331 Standard Compliance	24-31
Model Checks for IEC 61508, IEC 62304, ISO 26262, and EN 50128 Standard Compliance	24-36

Model Checks for MathWorks Automotive Advisory Board (MAAB) Guideline Compliance	24-39
Model Checks for MISRA C:2012 Compliance	24-45
Model Checks for Requirements Links	24-46
Generate Model Advisor Reports in Adobe PDF and Microsoft Word Formats	24-47
Modify Default Template	24-47
Run Checks for Model Metrics	24-51

Check Systems Programmatically

25

Checking Systems Programmatically	25-2
Find Check IDs	25-3
Create a Function for Checking Multiple Systems	25-5
Check Multiple Systems in Parallel	25-7
Create a Function for Checking Multiple Systems in Parallel	25-8
Archive and View Results	25-10
Archive Results	25-10
View Results in Command Window	25-10
View Results in Model Advisor Command-Line Summary Report	25-11
View Results in Model Advisor GUI	25-12
View Model Advisor Report	25-12
Archive and View Model Advisor Run Results	25-14

26

Model Metrics	26-2
Available Model Metrics	26-2
Count Metrics	26-3
Complexity Metrics	26-9
Readability Metrics	26-10
Create Model Metrics by Using APIs	26-13
Metric Class Constructor	26-13
Metric Algorithm	26-13
Collect Model Metrics Programmatically	26-16
Create Model Metric for Nonvirtual Block Count ...	26-20

Customizing the Model Advisor

Overview of Customizing the Model Advisor

27

Model Advisor Customization	27-2
Requirements for Customizing the Model Advisor	27-2

Create Model Advisor Checks

28

Create Model Advisor Checks Workflow	28-2
Customization File Overview	28-3
Common Utilities for Creating Checks	28-6

Create and Add Custom Checks - Basic Examples	28-7
Add Custom Check to By Product Folder	28-7
Create Customized Pass/Fail Check	28-8
Create Customized Pass/Fail Check with Fix Action .	28-11
Create Check for Model Configuration Parameters .	28-17
Create Data File for Diagnostics Pane Configuration	
Parameter Check	28-17
Create Check for Diagnostics Pane Model Configuration	
Parameters	28-20
Data File for Configuration Parameter Check	28-22
Register Checks and Process Callbacks	28-29
Create sl_customization Function	28-29
Register Checks and Process Callbacks	28-29
Define Startup and Post-Execution Actions Using Process	
Callback Functions	28-30
Define Custom Checks	28-33
About Custom Checks	28-33
Contents of Check Definitions	28-33
Display and Enable Checks	28-34
Define Where Custom Checks Appear	28-35
Check Definition Function	28-36
Define Check Input Parameters	28-36
Define Model Advisor Result Explorer Views	28-38
Define Check Actions	28-39
Create Callback Functions and Results	28-41
About Callback Functions	28-41
Informational Check Callback Function	28-42
Simple Check Callback Function	28-43
Detailed Check Callback Function	28-44
Check Callback Function with Hyperlinked Results .	28-45
Action Callback Function	28-48
Check With Subchecks and Actions	28-49
Basic Check with Pass/Fail Status	28-51
Exclude Blocks From Custom Checks	28-54
Format Check Results	28-57
Format Results	28-57
Format Text	28-57

Format Lists	28-58
Format Tables	28-58
Format Paragraphs	28-60
Formatted Output	28-60
Format Linebreaks	28-61
Format Images	28-61

Create Custom Configurations by Organizing Checks and Folders

29

Create Custom Configurations	29-2
Create Configurations by Organizing Checks and Folders	29-3
Create Procedural-Based Configurations	29-4
Organize Checks and Folders Using the Model Advisor Configuration Editor	29-5
Overview of the Model Advisor Configuration Editor ..	29-5
Start the Model Advisor Configuration Editor	29-8
Organize Checks and Folders Using the Model Advisor Configuration Editor	29-9
Organize Customization File Checks and Folders ...	29-11
Customization File Overview	29-11
Register Tasks and Folders	29-12
Define Custom Tasks	29-13
Define Custom Folders	29-15
Customization Example	29-17
Verify and Use Custom Configurations	29-19
Update the Environment to Include Your sl_customization File	29-19
Verify Custom Configurations	29-19

Create Procedural-Based Model Advisor Configurations

30

Create Procedures	30-2
What Is a Procedure?	30-2
Create Procedures Using the Procedures API	30-2
Define Procedures	30-2
Create a Procedural-Based Configuration	30-5

Deploy Custom Configurations

31

Overview of Deploying Custom Configurations	31-2
About Deploying Custom Configurations	31-2
Deploying Custom Configurations Workflow	31-2
How to Deploy Custom Configurations	31-3
Manually Load and Set the Default Configuration . . .	31-4
Automatically Load and Set the Default Configuration	31-5

Getting Started

- “Simulink Verification and Validation Product Description” on page 1-2
- “System Requirements” on page 1-3

Simulink Verification and Validation Product Description

Verify models and generated code

Simulink Verification and Validation automates requirements tracing, modeling standards compliance checking, and measurement of coverage for models and generated code.

You can create detailed requirements traceability reports, author your own modeling style checks, and develop check configurations to share with engineering teams. Requirements documentation can be linked to models, test cases, and generated code. You can use coverage analysis to confirm that models and generated code have been thoroughly tested.

Simulink Verification and Validation provides modeling standards checks for the DO-178, ISO 26262, IEC 61508 and related industry standards.

Key Features

- Compliance checking for MAAB style guidelines and high-integrity system design guidelines (DO-178, ISO 26262, IEC-61508, and related industry standards)
- Model Advisor Configuration Editor, including custom check authoring
- Requirements Management Interface for traceability of model objects, code, and tests to requirements documents
- Model coverage analysis and generated code coverage analysis with software-in-the-loop (SIL)
- Programmable scripting interface for automating compliance checking, requirements traceability analysis, and component testing

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, Windows Vista™, and Windows 7
- UNIX® systems (Requirements linking to HTML and TXT documents only)

Product Requirements

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

- MATLAB®
- Simulink

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

- Stateflow

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

- Microsoft Word 2003 or later
- Microsoft Excel® 2003 or later
- IBM® Rational® DOORS® 6.0 or later
- Adobe® PDF

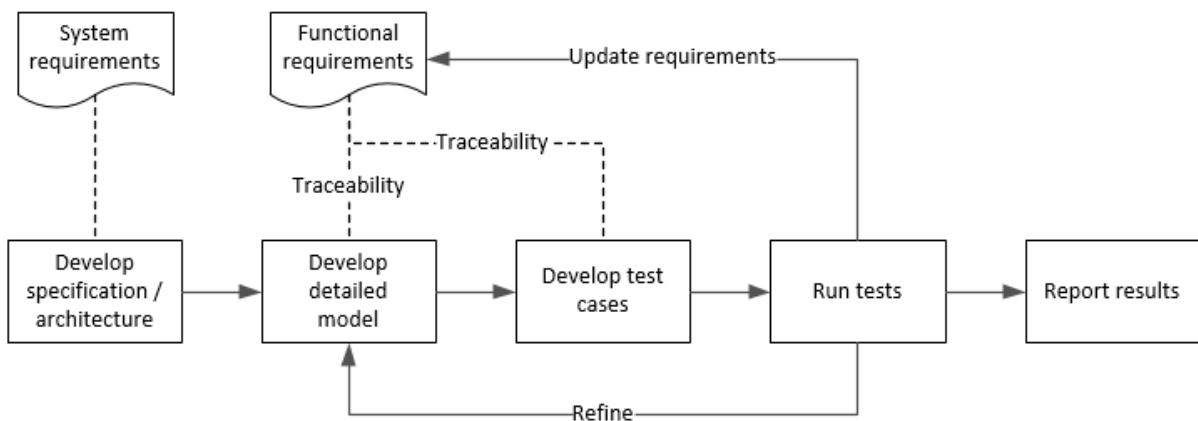
Verification and Validation

- “Test Model Against Requirements and Report Results” on page 2-2
- “Analyze a Model for Standards Compliance and Design Errors” on page 2-6
- “Perform Functional Testing and Analyze Test Coverage” on page 2-9
- “Analyze Code and Test Software-in-the-Loop” on page 2-16
- “Module Verification and Testing Processor-in-the-Loop” on page 2-25
- “Test a Model in Real Time” on page 2-26

Test Model Against Requirements and Report Results

Requirements Overview

Requirements are the basis for your system architecture, algorithm, and test plan. Traceability between requirements documents, model, code, and tests helps you document relationships, manage design changes, and interpret test results. Required model properties and test objectives enable targeted design analysis and test case generation for specific scenarios. You can evaluate your design and identify incomplete or missing requirements with ad-hoc testing, using simulated user interfaces for your model. Also, you can use rapid prototyping to validate requirements, and connect to physical or simulated environments to test your algorithm. Update the design, adding requirements and requirements links as necessary.



Test a Cruise Control Safety Requirement

This example shows a requirements-based testing workflow for a cruise control model. You start with a model that has traceability to an external requirements document. You add a test to evaluate two safety requirements, checking that the cruise control disengages when the system reaches certain conditions. You add traceability to this test, run the test, and report the results.

- 1 Create a copy of the project in a working folder. Enter
`s1VerificationCruiseStart`

- 2 Open the model and the test harness. On the command line, enter

```
open_system simulinkCruiseAddReqExample
slttest.harness.open('simulinkCruiseAddReqExample','SafetyTest_Harness1')
```

- 3 Open the Test Sequence block.

- The **BrakeTest** sequence tests that the system disengages when the brake pedal is pressed. It includes a **verify** statement

```
verify(engaged == false,...
      'verify:brake',...
      'system must disengage when brake applied')
```

- The **LimitTest** sequence tests that the system disengages when the speed exceeds a limit. It includes a **verify** statement

```
verify(engaged == false,...
      'verify:limit',...
      'system must disengage when limit exceeded')
```

- 4 Open the requirements document. In the Simulink Project window, expand the **documents** folder and open **simulinkCruiseChartReqs.docx**.
- 5 Add links between the test steps and the requirements document.

- 1 In the requirements document, highlight item 3.1, “Vehicle braking will transition system to disengaged (inactive) when engaged (active)”

- 2 With item 3.1 highlighted, in the test sequence, right-click the **BrakeTest** step. Select **Requirements traceability > Link to Selection in Word**.

- 3 In the requirements document, highlight item 3.4, “Transition to disengaged (inactive) when vehicle speed is outside the limits of 20 mph to 90 mph”

- 4 With item 3.4 highlighted, in the test sequence, right-click the **LimitTest** step. Select **Requirements traceability > Link to Selection in Word**.


- 5 Save the requirements document and the model.

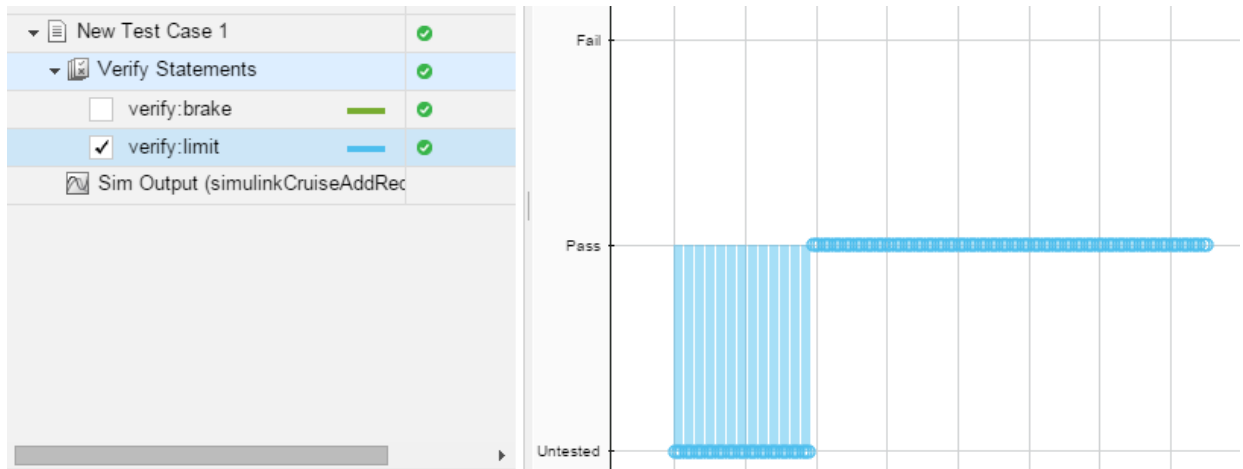
- 6 Create a test case in the Test Manager, and link the test case to the requirements section.

- 1 Open the Test Manager. In the Simulink menu, select **Analysis > Test Manager**.

- 2 In the Test Manager toolstrip, click **New > Test File**. Select the tests folder in the project, and enter a name for the test file. Click **Save**.



A new baseline test is created.

- 3 Under **System Under Test**, in the **Model** field, click the button  to use the current model. The field displays the model name.
- 4 Expand the **Test Harness** section. From the drop-down menu, select the test harness name.
- 5 In the requirements document, highlight section 3.1.
- 6 In the test case, expand the **Requirements** section. Click the arrow next to the **Add** button and select **Link to Selection in Word**.
- 7 Use the same process to link the test case to section 3.4 in the requirements document.
- 7 Highlight the test case. In the Test Manager toolstrip, click **Run**.
- 8 When the test finishes, expand the **Verify Statements** results. The results show that both assessments pass, and the plot shows the detailed results of each statement.



- 9 Create a report using a custom Microsoft Word template.
 - 1 In the Test Manager, right-click the test case name. Select **Results: > Create Report**.
 - 2 In the Create Test Result Report dialog box, set the options:
 - Title: **SafetyTest**
 - Results for: **All Tests**

- File Format: DOCX
 - For the other options, keep the default selections.
- 3 For the **Template File**, select the `ReportTemplate.dotx` file in the **documents** project folder.
 - 4 Enter a file name and select a location for the report.
 - 5 Click **Create**.
- 10 Review the report.
- 1 In the **Test Case Requirements** section, click the link to trace to the requirements document.
 - 2 The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

Name	Data Type	Units	Sample Time	Interp	Sync	Link to Plot
 Test Sequence/.../Verify:verify(engaged == false)	siTestResult			zoh	union	Link
 Test Sequence/.../VerifyHigh:verify(engaged == false)	siTestResult			zoh	union	Link

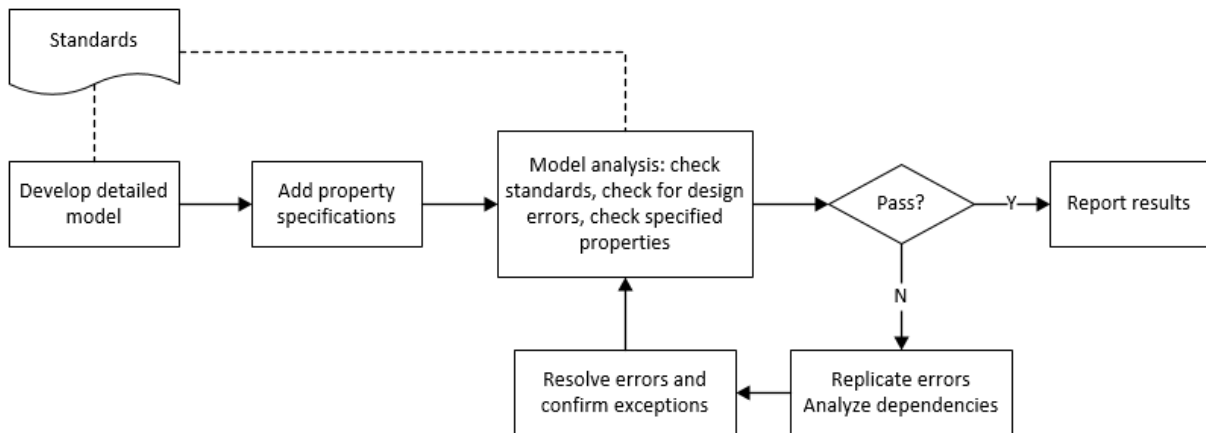
Related Examples

- “Link to Requirements Modeled in Simulink” on page 3-49
- “Link Tests to Requirements”
- “Validate Requirements Links in a Model” on page 6-4
- “Create Requirements Traceability Report for Model” on page 3-46

Analyze a Model for Standards Compliance and Design Errors

Standards and Analysis Overview

During model development, check and analyze your model to increase confidence in its quality. Check your model against standards such as MAAB style guidelines and high-integrity system design guidelines such as DO-178 and ISO 26262. Analyze your model for errors, dead logic, and conditions that violate required properties. Using the analysis results, update your model and document exceptions. Report the results using customizable templates.



Check Model for Style Guideline Violations and Design Errors

This example shows how to use the Model Advisor to check a cruise control model for MathWorks Automotive Advisory Board (MAAB) style guideline violations and design errors. Select checks and run the analysis on the model. Iteratively debug issues using the Model Advisor and rerun checks to verify that it is in compliance. After passing your selected checks, report results.

Check Model for MAAB Style Guideline Violations

In Model Advisor, you can check that your model complies with MAAB modeling guidelines.

- 1 Create a copy of the project in a working folder. On the command line, enter
`slVerificationCruiseStart`
- 2 Open the model. On the command line, enter
`open_system simulinkCruiseErrorAndStandardsExample`
- 3 In the model window, select **Analysis > Model Advisor > Model Advisor**.
- 4 Click OK to choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
- 5 Check your model for MAAB style guideline violations using Simulink Verification and Validation.
 - a In the left pane, in the **By Product > Simulink Verification and Validation > Modeling Standards > MathWorks Automotive Advisory Board Checks** folder, select:
 - **Check for indexing in blocks**
 - **Check for prohibited blocks in discrete controllers**
 - **Check model diagnostic parameters**
 - b Right-click the **MathWorks Automotive Advisory Board Checks** node, and then select **Run Selected Checks**.
 - c Click **Check model diagnostic parameters** to review the configuration parameter settings that violate MAAB style guidelines.
 - d In the right pane, click the parameter links to update the values in the Configuration Parameters dialog box.
 - e To verify that your model passes, rerun the check. Repeat steps c and d, if necessary, to reach compliance.
 - f To generate a results report of the Simulink Verification and Validation checks, select the **MathWorks Automotive Advisory Board Checks** node, and then, in the right pane click **Generate Report...**

Check Model for Design Errors

While in Model Advisor, you can also check your model for hidden design errors using Simulink Design Verifier.

- 1 In the left pane, in the **By Product > Simulink Design Verifier** folder, select **Design Error Detection**.

- 2 In the right pane, click **Run Selected Checks**.
- 3 After the analysis is complete, expand the **Design Error Detection** folder, then select checks to review warnings or errors.
- 4 In the right pane, click **Simulink Design Verifier Results Summary**. The dialog box provides tools to help you diagnose errors and warnings in your model.
 - a Review the results on the model. Click **Highlight analysis results on model**. Click the **Compute target speed** subsystem, outlined in red. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.
 - b Review the harness model. The Simulink Design Verifier Results Inspector window displays information that an overflow error occurred. To see the test cases that demonstrate the errors, click **View test case**.
 - c Review the analysis report. In the Simulink Design Verifier Results Inspector window, click **Back to summary**. To see a detailed analysis report, click **HTML** or **PDF**.

Related Examples

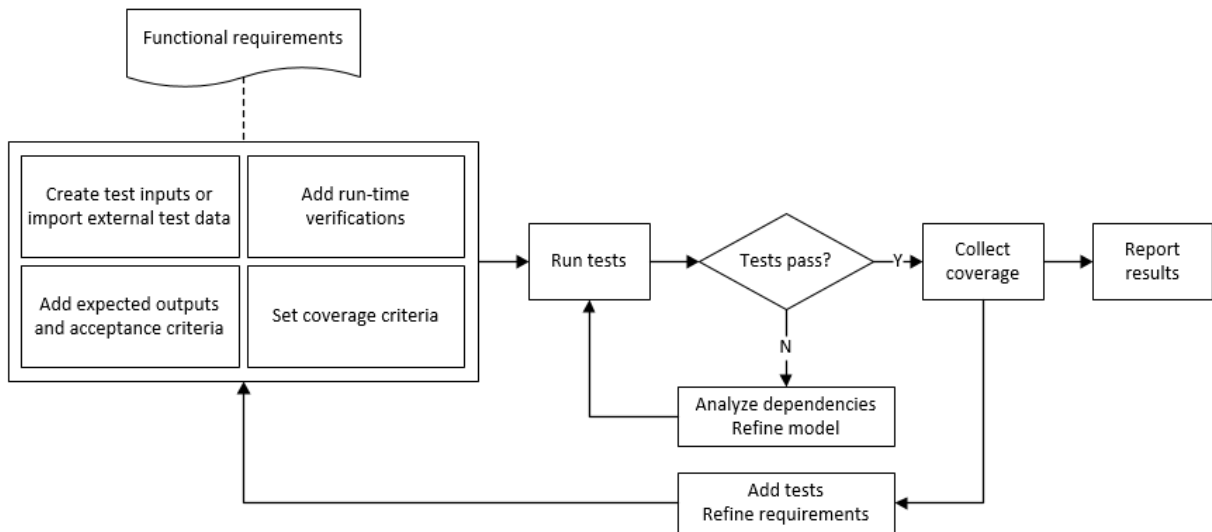
- “Check for Compliance in Model and Subsystems” on page 24-2
- “Run Checks for Model Metrics” on page 24-51
- “Run a Design Error Detection Analysis”
- “Prove Properties in a Model”

Perform Functional Testing and Analyze Test Coverage

Functional Testing and Coverage Analysis Overview

Functional testing starts with building test cases based on requirements. These tests can cover key aspects of your design and verify that individual model components meet requirements. Test cases include inputs, expected outputs, and acceptance criteria.

By collecting individual test cases within test suites, you can run functional tests systematically. To check for regression, add baseline criteria to the test cases and test the model regularly. Coverage measurement reflects the extent to which these tests have fully exercised the model. Coverage measurement also helps you to add tests and requirements to meet coverage targets.



Incrementally Increase Test Coverage Using Test Case Generation

This example shows a functional testing-based testing workflow for a cruise control model. You start with a model that has tests linked to an external requirements document, analyze the model for coverage in Simulink Verification and Validation, incrementally increase coverage with Simulink Design Verifier, and report the results.

Explore the Test Harness and the Model

- 1 Create a copy of the project in a working folder. At the command line, enter:

```
slVerificationCruiseStart
```

- 2 Open the model and the test harness. At the command line, enter:

```
open_system simulinkCruiseAddReqExample  
sltest.harness.open('simulinkCruiseAddReqExample','SafetyTest_Harness1')
```

- 3 Load the test suite from “Test Model Against Requirements and Report Results”. At the command line, enter:

```
open slReqTests.mldatx
```

- 4 Open the test sequence block. The sequence tests:

- That the system disengages when the brake pedal is pressed
- That the system disengages when the speed exceeds a limit

Some test sequence steps are linked to a requirements document `simulinkCruiseChartReqs.docx`.

Measure Model Coverage and Save Coverage Results

- 1 In the test manager, enable coverage collection for the test case.
 - a Open the test manager. In the Simulink menu, click **Analysis > Test Manager**.
 - b In the **Test Browser**, click the `slReqTests` test file.
 - c Expand **Coverage Settings**.
 - d Under **COVERAGE TO COLLECT**, select **Record coverage for referenced models**.
 - e Under **COVERAGE METRICS**, select **Decision**, **Condition**, and **MCDC**.

▼ COVERAGE SETTINGS

▼ COVERAGE TO COLLECT

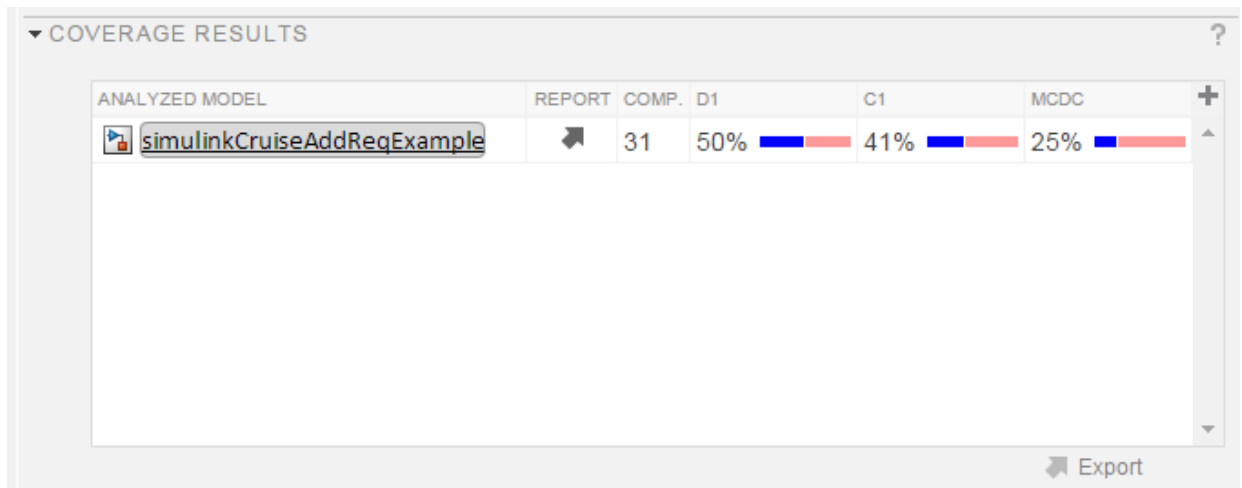
Record coverage for system under test

Record coverage for referenced models

COVERAGE METRICS

<input checked="" type="checkbox"/> Decision	<input checked="" type="checkbox"/> Condition
<input checked="" type="checkbox"/> MCDC	<input type="checkbox"/> Lookup Table
<input type="checkbox"/> Signal Range	<input type="checkbox"/> Signal Size
<input type="checkbox"/> Simulink Design Verifier	<input type="checkbox"/> Saturation on integer overflow
<input type="checkbox"/> Relational Boundary	

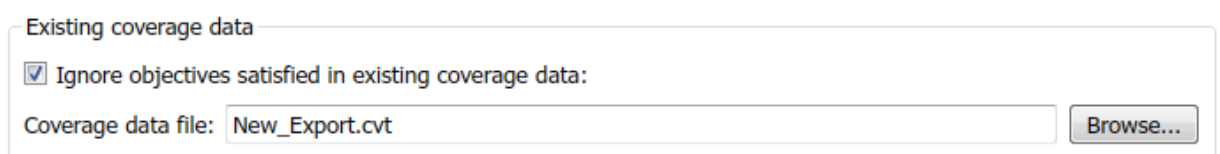
- 2 Run the test. On the test manager toolstrip, click **Run**.
- 3 When the test finishes, in the Test Manager, navigate to the test case. The example model achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.



- 4 In the coverage results, right-click the `simulinkCruiseAddReqExample` row and click **Export**
- 5 Select **Export to CVT-file** and export the coverage results to a folder in your working folder.

Generate Tests to Increase Model Coverage

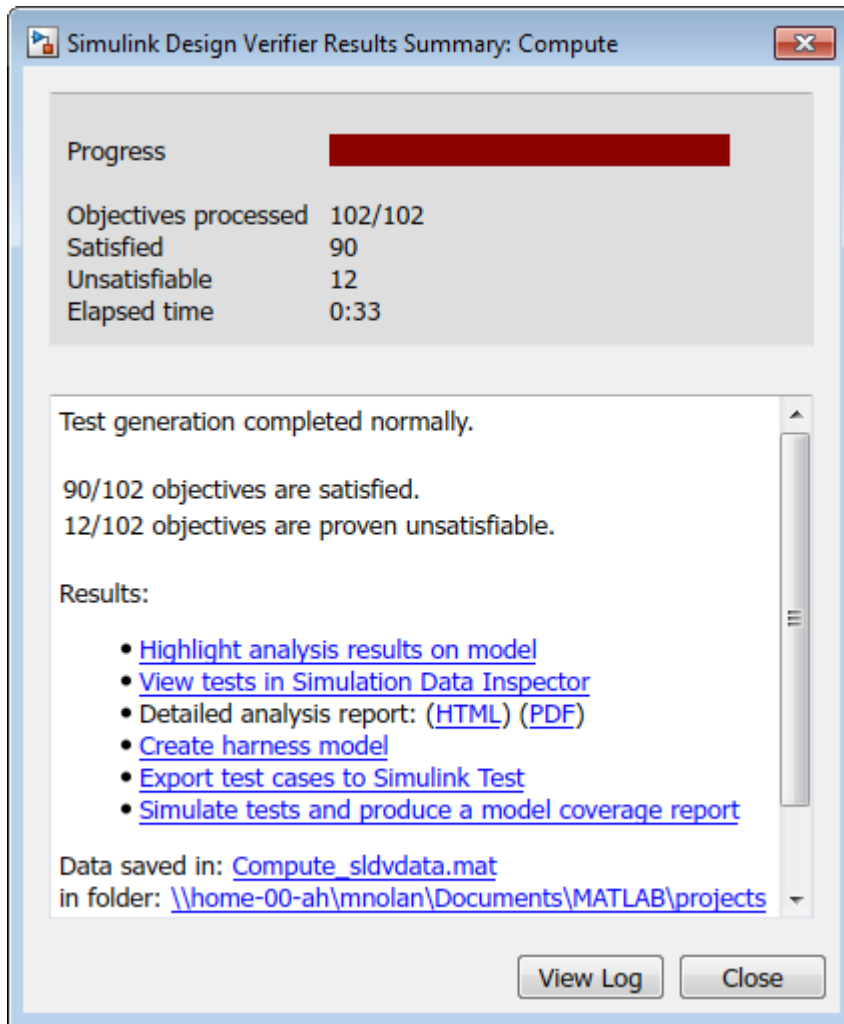
- 1 Generate additional tests for the referenced model. Specify that the analysis ignore satisfied coverage data from the coverage results recorded in the previous steps.
 - a Open the `simulinkCruiseAddReqExample` model Configuration Parameters
 - b In the **Design Verifier > Test Generation** pane, select **Ignore objectives satisfied in existing coverage data**
 - c Click **Browse**, select the Coverage data file that you exported in the previous section, and click **Open**.



- 2 To apply the settings and close the Configuration Parameters dialog box, click **OK**.

- 3 In the `simulinkCruiseAddReqExample` model, right-click the `Compute` target speed subsystem and click **Design Verifier > Generate Tests for Subsystem**. The test case generation is executed.
















The Simulink Design Verifier Results Summary window opens with the test case generation report.



- 4 In the Simulink Design Verifier Results Summary window, click **Simulate tests and produce a model coverage report**.

The example model now achieves 89% decision coverage, 88% condition coverage, and 88% MCDC coverage.

Summary

Model Hierarchy/Complexity	Test 1		
	Decision	Condition	MCDC
1. Compute	31 89% 	88% 	88% 
2. . . . Compute target speed	30 89% 	88% 	88% 
3. SF: Compute target speed	29 89% 	88% 	88% 
4. SF: CRUISE	26 88% 	88% 	88% 
5. SF: ON	15 80% 	75% 	75% 

Export the Test Cases to the Test Manager

To export the test cases that Simulink Design Verifier generates into the Test Manager, first close the existing Simulink Test test harness.

- 1 Close the `SafetyTest_Harness1` test harness.
- 2 In the Simulink Design Verifier Results Summary window, click **Export test cases to Simulink Test**.
- 3 Select the `simulinkCruiseAddReqExample_sldvharness` test harness and click **OK**.
- 4 In the test manager, a new test file `simulinkCruiseAddReqExample_test` appears. The test file contains a new test case that uses the inputs generated by Simulink Design Verifier for model coverage.
- 5 Right-click the new test case in the `simulinkCruiseAddReqExample_test` test file and select **Copy**.
- 6 In the original `s1ReqTests` test file, right-click the test suite and select **Paste**.

The test suite in the `slReqTests` test file now contains the original test case and the test cases generated by Simulink Design Verifier.

- 7 Run the test suite again. Highlight the `slReqTests Test Suite 1` test suite. On the test manager toolstrip, click **Run**.

The test results include coverage for the combined test case inputs.

- 8 Close the test harness and the model.

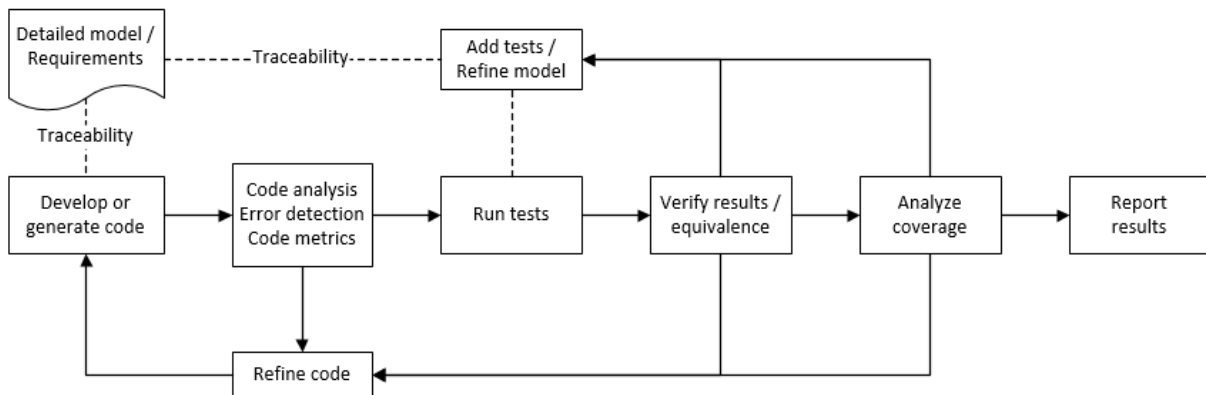
Related Examples

- “Link Tests to Requirements”
- “Assess Simulation Using Logical Statements”
- “Test Model Output Against a Baseline”
- “Highlight Functional Dependencies”
- “Generate Test Cases for Model Decision Coverage”
- “Extend Model Coverage of a Test Suite”

Analyze Code and Test Software-in-the-Loop

Code Analysis and Testing Software-in-the-Loop Overview

Analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. Typically for handwritten code, you check for run-time errors with static code analysis and run test cases that evaluate the code against requirements and evaluate code coverage. Based on the results, refine the code and add tests. For generated code, demonstrate that code execution produces equivalent results to the model by using the same test cases and baseline results. Compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.



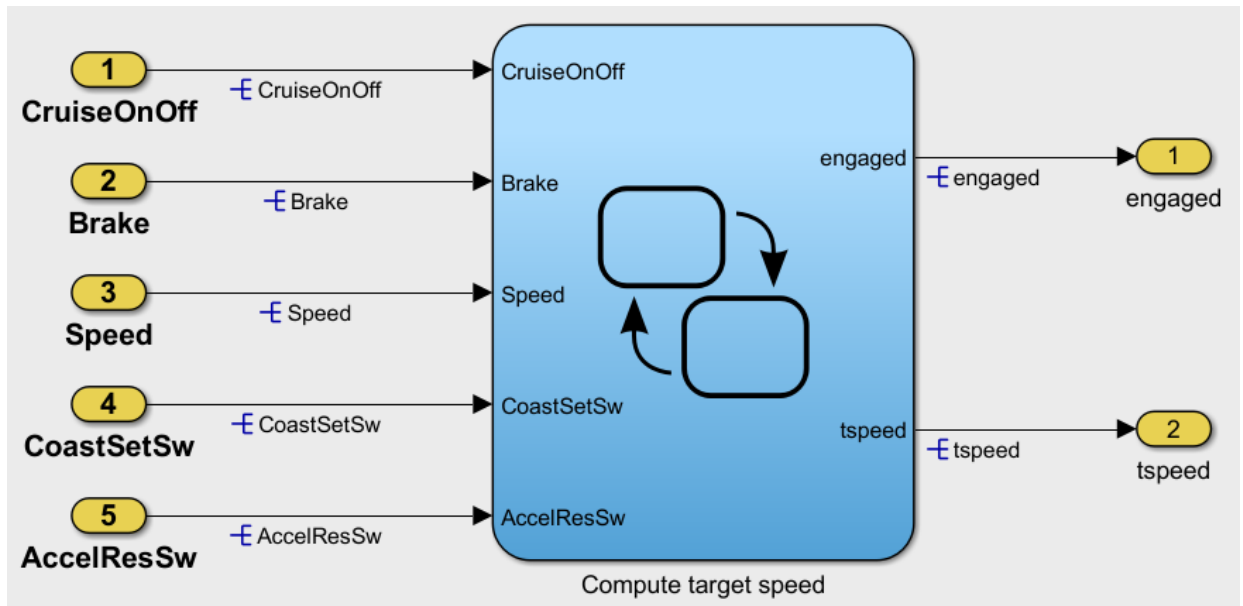
Analyze Code for Defects, Metrics, and MISRA C:2012

This workflow describes how to check if your model produces MISRA[®] C:2012 compliant code and how to check your generated code for code metrics, code defects, and MISRA compliance. To produce more MISRA compliant code from your model, you use the code generation and model advisors. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

- 1 Open the Simulink project:

slVerificationCruiseStart

- From the Simulink project, open the model `simulinkCruiseErrorAndStandardsExample`.

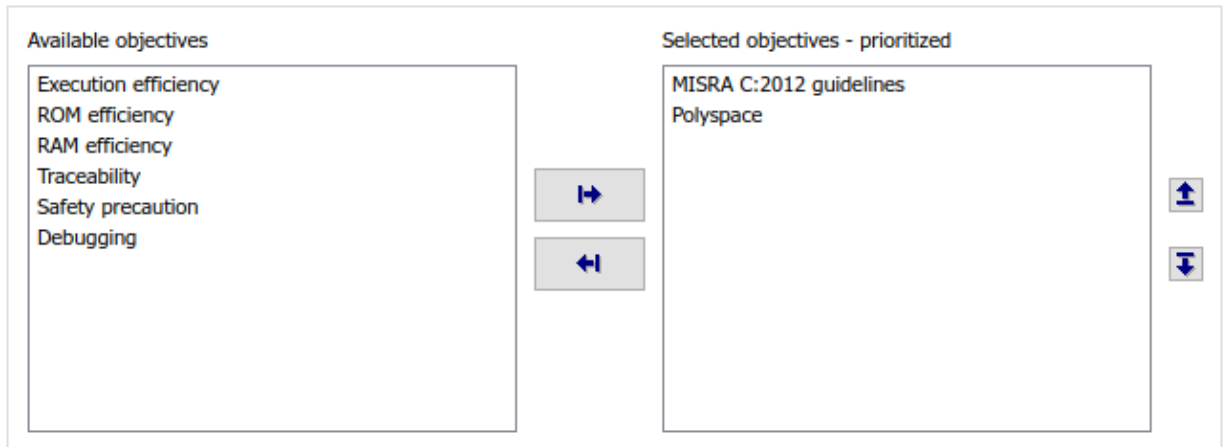


Run Code Generator Checks

Before you generate code from your model, there are steps that you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows how to use the Code Generation Advisor to check your model before generating code.

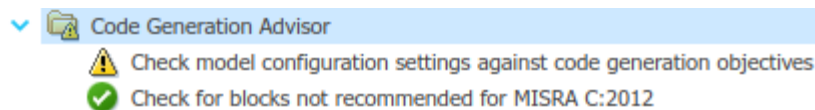
- Right-click `Compute target speed` and select `C/C++ > Code Generation Advisor`.
- Select the Code Generation Advisor folder. Add the `Polyspace` objective. The `MISRA C:2012 guidelines` objective is already selected.

Code Generation Objectives (System target file: ert.tlc)



3 Click **Run Selected Checks**.

The Code Generation Advisor checks whether there are any blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this mode, the check for incompatible blocks passes, but there are some configuration settings that are incompatible with MISRA compliance and Polyspace checking.



4 Click on check that was not passed. Accept the parameter changes by selecting **Modify Parameters**.

5 Rerun the check by selecting **Run This Check**.

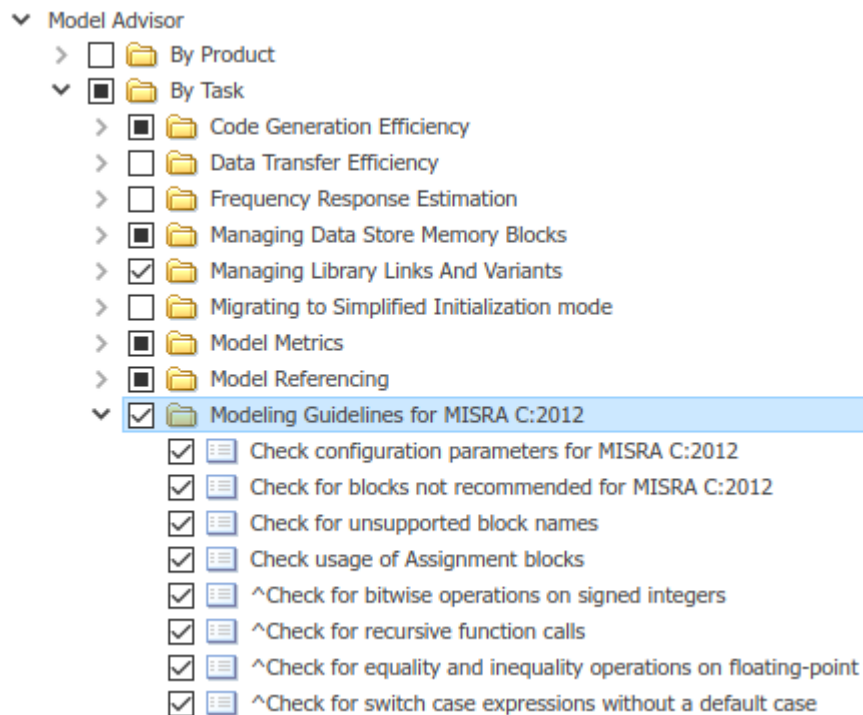
For your own model, you might not want to use all the recommended configuration settings. Using nonrecommended settings can generate less MISRA compliant code.

Run Model Advisor Checks

Before you generate code from your model, there are steps you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows you how to use the Model Advisor to check your model further before generating code.

For more checking before generating code, you can also run the Modeling Guidelines for MISRA C:2012.

- 1 At the bottom of the Code Generation Advisor window, select **Model Advisor**.
- 2 Under the **By Task** folder, select the **Modeling Guidelines for MISRA C:2012** advisor checks.



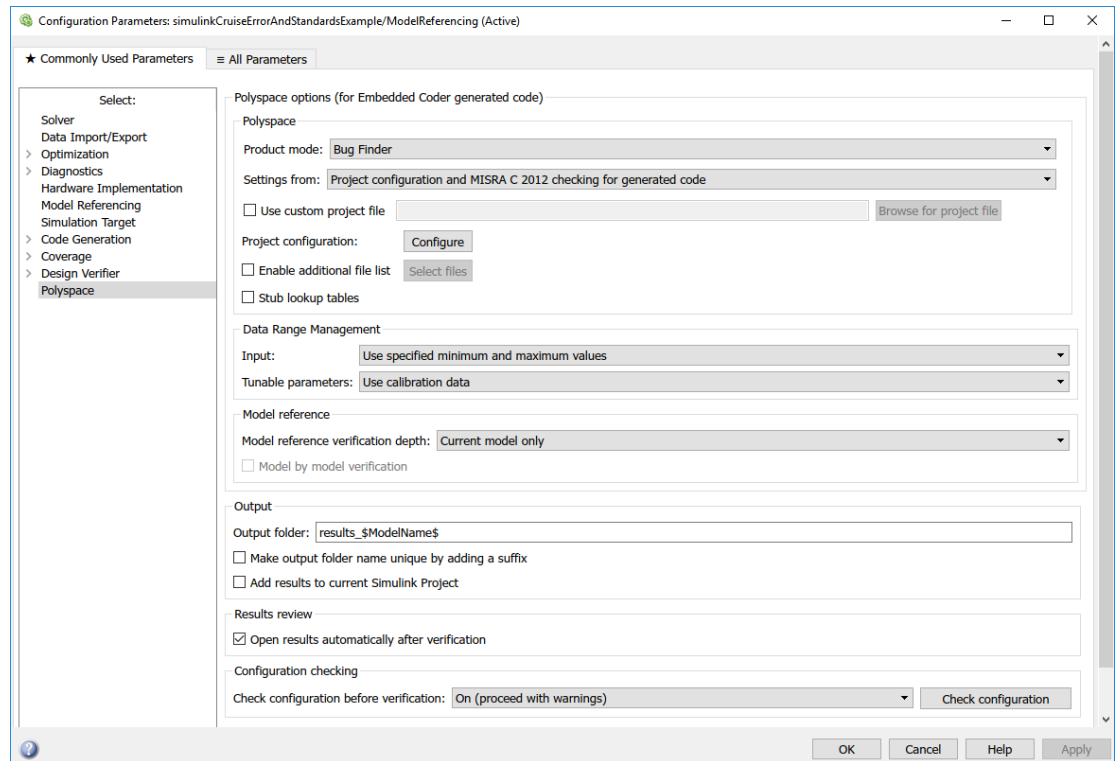
- 3 Click **Run Selected Checks** and review the results.
- 4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

For your own model, you might not want to use all the recommendations. Using nonrecommended settings or blocks can generate less MISRA compliant code.

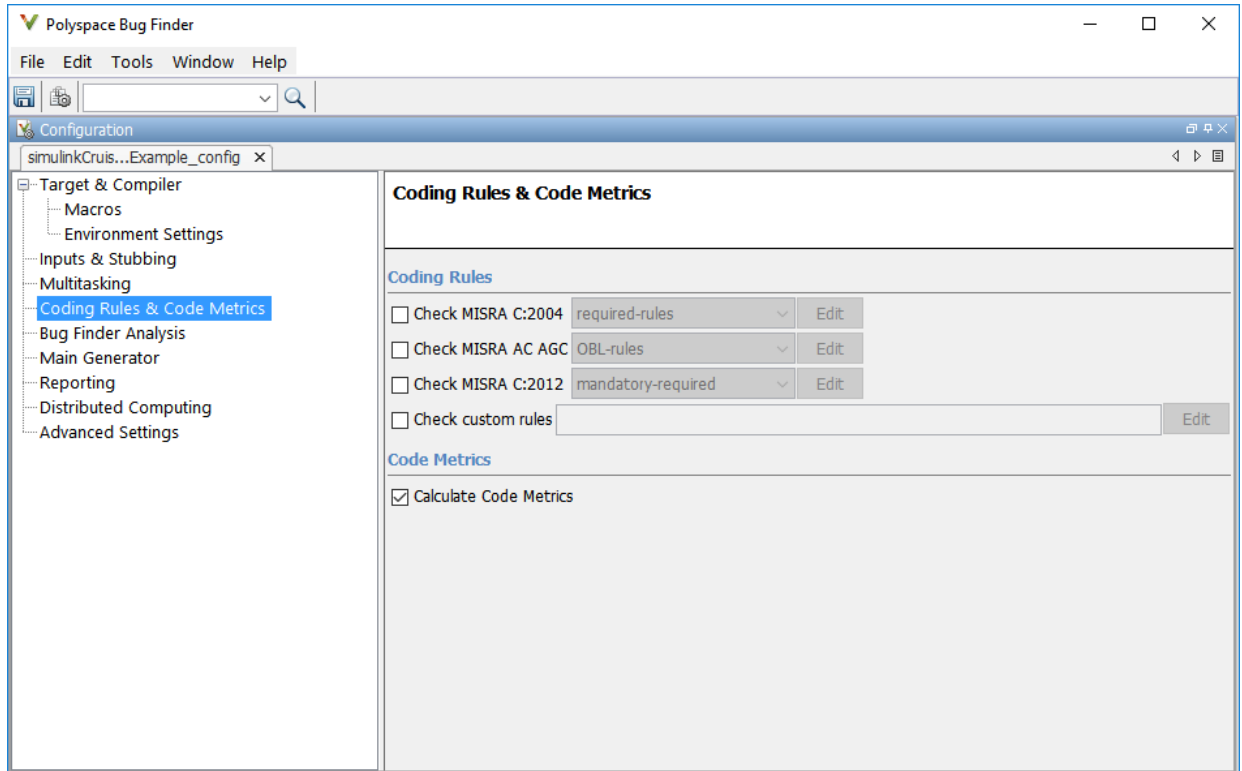
Generate and Analyze Code

After you have done the model compliance checking, you can now generate code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

- 1 In the Simulink editor, right-click Compute target speed and select **C/C++ > Build This Subsystem**.
- 2 Use the default settings for the tunable parameters and select **Build**.
- 3 After the code is generated, right-click Compute target speed and select **Polyspace > Options**.



- Click the **Configure** button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.



- On the same pane, select **Calculate Code Metrics**. This option turns on code metric calculations for your generated code.
- Save and close the Polyspace configuration window.
- From your model, right-click Compute target speed and select **Polyspace > Verify Code Generated For > Selected Subsystem**.

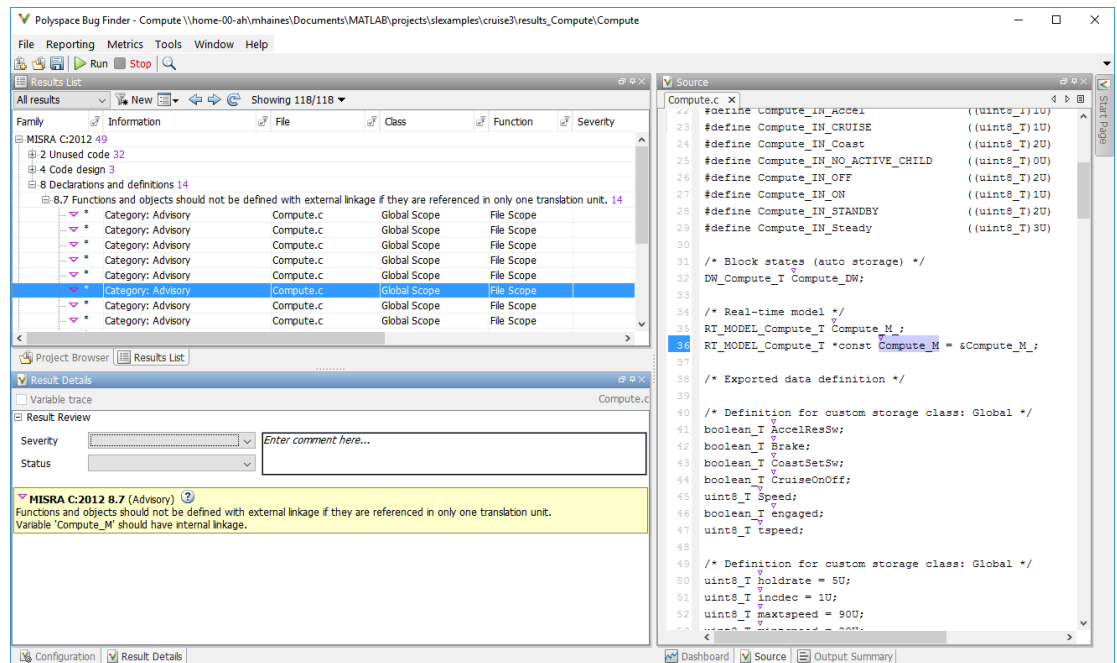
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

Review Results

After you run a Polyspace analysis of your generated code, the Polyspace environment shows you the results of the static code analysis. There are 50 MISRA C:2012 coding rule violations in your generated code.

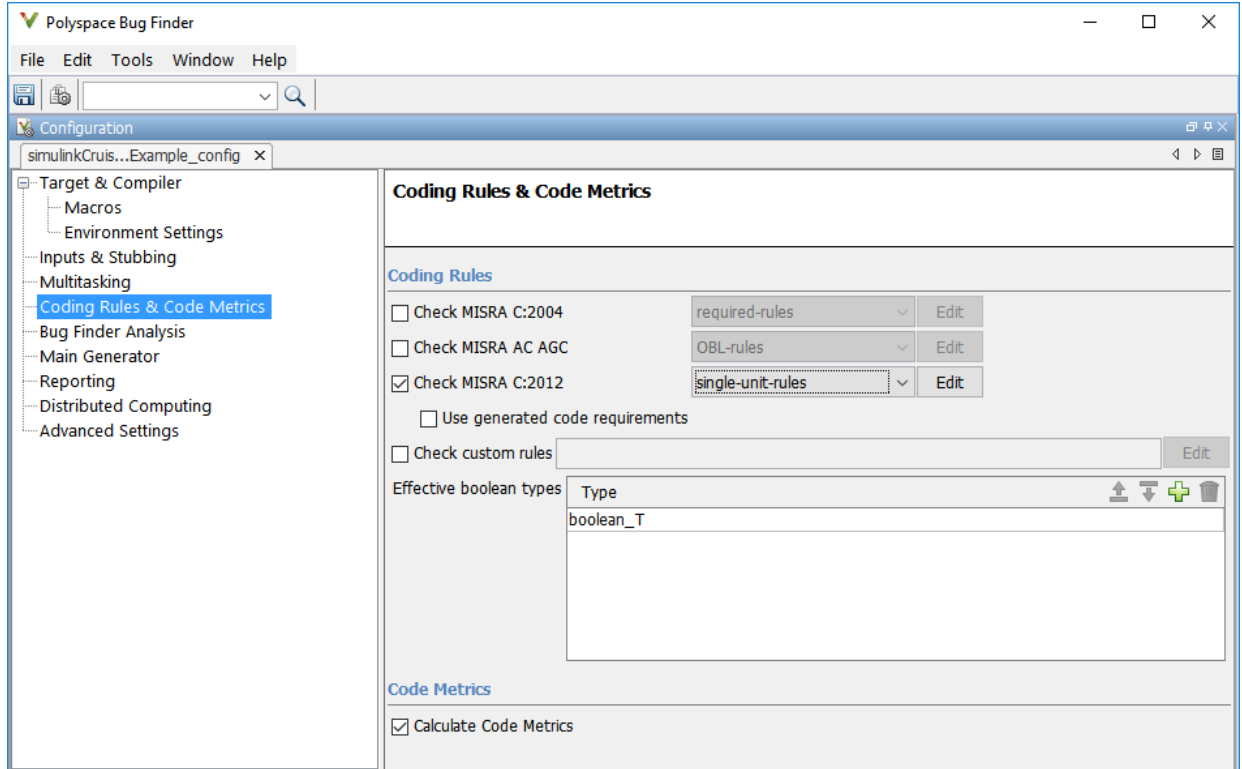
- 1 Expand the tree for rule 8.7 and click through the different results.

Rule 8.7 states that functions and objects should not be global if the function or object is local. As you click through the 8.7 violations, you can see that these results refer to variables that other components also use, such as `CruiseOnOff`. You can annotate your code or your model to justify every result. But, because this model is a unit in a larger program, you can also change the configuration of the analysis to check only a subset of MISRA rules.



- 2 In your model, right-click Compute target speed and select **Polyspace > Options**.
- 3 Set the **Settings from** option to **Project configuration**. This option allow you to choose a subset of MISRA rules in the Polyspace configuration.
- 4 Click the **Configure** button.

- 5 In the Polyspace Configuration window, on the **Coding Rules & Code Metrics** pane, select the check box **Check MISRA C:2012** and from the drop-down list, select **single-unit-rules**. Now, Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.



- 6 Save and close the Polyspace configuration window.
- 7 Rerun the analysis with the new configuration.

When the Polyspace environment reopens, there are no MISRA results, only code metric results. The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, no violations were found.



When this model is integrated with its parent model, you can add the rest of the MISRA C:2012 rules.

Generate Report

To demonstrate compliance with MISRA C:2012 and report on your generated code metrics, you must export your results. This section shows you how to generate a report after the analysis. If you want to generate a report every time you run an analysis, see Generate report.

- 1 If they are not open already, open your results in the Polyspace environment.
- 2 From the toolbar, select **Reporting > Run Report**.
- 3 Select **BugFinderSummary** as your report type.
- 4 Click **Run Report**.

The report is saved in the same folder as your results.

- 5 To open the report, select **Reporting > Open Report**.

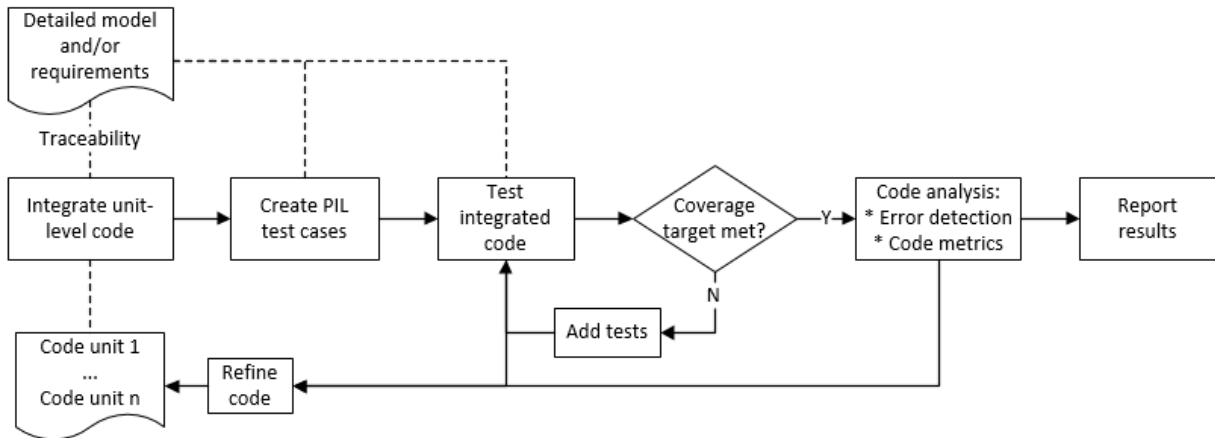
Related Examples

- “Generate and Analyze Code”
- “Test Two Simulations for Equivalence”
- “Export Test Results and Generate Reports”

Module Verification and Testing Processor-in-the-Loop

Module Verification and Testing Processor-in-the-Loop Overview

Module verification involves testing and analyzing code at a system level, integrating generated code from your model, handwritten code, and legacy code. Module verification often includes generating code that executes on a target object, rather than the desktop environment. Analyze the code to resolve errors and evaluate key metrics. Test the integrated system using new requirements-based tests and system-level tests from your model. Collect coverage on these tests and add tests to meet coverage targets.



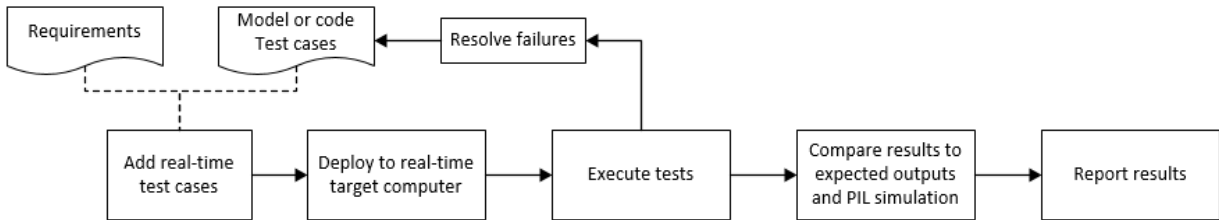
Related Examples

- “Test Two Simulations for Equivalence”
- “Generate and Analyze Code”

Test a Model in Real Time

Real-Time Testing and Testing Production Models Overview

Real-time testing assesses the system while including the effects of timers, physical signals, and target hardware. Sweep through parameter values on the target, verify system operation during execution, and verify expected results in the desktop environment. Systems that have been verified on target hardware often exist in a change-controlled state. You can test these systems without modifying them by using isolated simulation and analysis environments.



Related Examples

- “Create and Run Real-Time Application from Simulink Model”
- “Test Models in Real Time”
- “Assess Simulation Using Logical Statements”

Requirements Traceability

Links Between Models and Requirements

- “Overview of the Requirements Management Interface (RMI)” on page 3-3
- “Requirements Traceability Links” on page 3-4
- “Requirements Link Storage” on page 3-5
- “Supported Requirements Document Types” on page 3-6
- “Supported Model Objects for Requirements Linking” on page 3-9
- “Selection-Based Linking” on page 3-10
- “Link to Requirements Document Using Selection-Based Linking” on page 3-11
- “Link Test Cases to Requirements Documents” on page 3-12
- “Configure RMI for IBM Rational DOORS or Microsoft ActiveX Navigation” on page 3-16
- “Requirements Traceability Link Editor” on page 3-17
- “Requirements Settings” on page 3-20
- “Link Model Objects” on page 3-22
- “Link from External Applications” on page 3-24
- “Link Multiple Model Objects to a Requirements Document” on page 3-25
- “Link to Requirements in Microsoft Word Documents” on page 3-30
- “Link to Requirements in IBM Rational DOORS Databases” on page 3-36
- “Link to Requirements in Microsoft Excel Workbooks” on page 3-38
- “Link to Requirements in MuPAD Notebooks” on page 3-43
- “Create Requirements Traceability Report for Model” on page 3-46
- “Link to Requirements Modeled in Simulink” on page 3-49
- “Requirements Linking with Simulink Annotations” on page 3-58
- “Link Signal Builder Blocks to Requirements Documents” on page 3-59

- “Link Signal Builder Blocks to Model Objects” on page 3-61
- “Link Requirements to Simulink Data Dictionary Entries” on page 3-63

Overview of the Requirements Management Interface (RMI)

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

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

Requirements Traceability Links

When you want to navigate from a Simulink model or from a region of MATLAB code to a location inside a requirements document, you can add requirements traceability links to the model or code.

Requirements traceability links have the following attributes:

- A description of up to 255 characters.
- A requirements document path name, such as a Microsoft Word file or a module in an IBM Rational DOORS database. (The RMI supports several built-in document formats. You can also register custom types of requirements documents. See “Supported Requirements Document Types” on page 3-6.)
- A designated location inside the requirements document, such as:
 - Bookmark
 - Anchor
 - ID
 - Page number
 - Line number
 - Cell range
 - Link target
 - Tags that you define

Requirements Link Storage

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

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

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

For more information on external storage of requirements links data, see “External Storage” on page 4-2 and “Guidelines for External Storage of Requirements Links” on page 4-3.

Supported Requirements Document Types

The Requirements Management Interface (RMI) supports linking with external documents of the types listed in the table below. For each supported requirements document type, the table lists the options for requirements locations within the document.

If you would like to implement linking with a requirements document of a type that is not listed in the table below, you can register a custom requirements document type with the RMI. For more information, see “Create a Custom Requirements Link Type” on page 10-11.

Requirements Document Type	Location Options
Microsoft Word 2003 or later	<ul style="list-style-type: none"> • Named item — A bookmark name. The RMI links to the location of that bookmark in the document. The most stable location identifier because the link is maintained when the target content is modified or moved. • Search text — A search string. The RMI links to the first occurrence of that string in the document. This search is not case sensitive. • Page/item number — A page number. The RMI links to the top of the specified page.
Microsoft Excel 2003 or later	<ul style="list-style-type: none"> • Named item — A named range of cells. The RMI links to that named item in the workbook. The most stable location identifier because the link is maintained when the target content is modified or moved. • Search text — A search string. The RMI links to the first occurrence of that string in the workbook. This search is not case sensitive. • Sheet range — A cell location in a workbook: <ul style="list-style-type: none"> • Cell number (A1, C13) • Range of cells (C5:D7) • Range of cells on another worksheet (Sheet1!A1:B4) <p>The RMI links to that cell or cells.</p>

Requirements Document Type	Location Options
IBM Rational DOORS	<p>Page/item number — The unique numeric ID of the target DOORS object. The RMI links to that object.</p>
MuPAD	<p>Named item — The name of a link target in a MuPAD notebook.</p>
Simulink DocBlock block (RTF format only)	<p>Create links to the RTF file associated with the DocBlock block to a Microsoft Word file:</p> <ul style="list-style-type: none"> • Search text — A search string. The RMI links to the first occurrence of that string in the document. This search is not case sensitive. • Named item — A bookmark name. The RMI links to the location of that bookmark in the document. • Page/item number — A page number. The RMI links to the top of that page.
Text	<ul style="list-style-type: none"> • Search text — A search string. The RMI links to the first occurrence of that string within the document. This search is not case sensitive. • Line number — A line number. The RMI links to the beginning of that line.
HTML	<p>You can link only to a named anchor.</p> <p>For example, in your HTML requirements document, if you define the anchor</p> <pre data-bbox="501 1135 1092 1159"> ...contents... </pre> <p>in the Location field, enter <code>valve_timing</code> or, from the document index, choose the anchor name.</p> <p>Select the Document Index tab in the “Requirements Traceability Link Editor” on page 3-17 to see available anchors in an HTML file.</p>

Requirements Document Type	Location Options
Web browser URL	<p>The RMI can link to a URL location. In the Document field, type the URL string. When you click the link, the document opens in a Web browser:</p> <ul style="list-style-type: none">• Named item — An anchor name. The RMI links to that location on the Web page at that URL.
PDF	<p>Navigation will open a PDF document but will not scroll to a specific page or bookmark.</p> <p>The RMI cannot create a document index of bookmarks in PDF files.</p>

Supported Model Objects for Requirements Linking

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

- Simulink block diagrams and subsystems
- Simulink blocks and annotations
- Simulink data dictionary entries
- Signal Builder signal groups
- Stateflow charts, subcharts, states, transitions, and boxes
- Stateflow functions
- Lines of MATLAB code
- Simulink Test Manager test cases

Selection-Based Linking

Abstract

Use selection-based linking to create links from a model object to another model object or to an object in a requirements document.

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

For examples of selection-based linking, see “Link to Requirements Document Using Selection-Based Linking” on page 3-11 and “Link Multiple Model Objects to a Requirements Document” on page 3-25.

Link to Requirements Document Using Selection-Based Linking

Abstract

Use selection-based linking to create a link from a model to a requirements document.

To create a link from a model to a requirements document, using selection-based linking:

- 1 In the requirements document, select text or objects to link to.
- 2 Right-click the model object. Select **Requirements Traceability** and then the option that corresponds to one of the types for which selection-based linking is available:
 - **Link to Selection in MATLAB**
 - **Link to Selection in Word**
 - **Link to Selection in Excel**
 - **Link to Selection in DOORS**
 - **Select for Linking with Simulink**

Link Test Cases to Requirements Documents

Since requirements specify behavior in response to particular conditions, you can build test cases (test inputs, expected outputs, and assessments) from the model requirements. Test cases reproduce specific conditions using test inputs, and assess the actual model output against the expected outputs. As you develop the model, build test files that check system behavior and link them to corresponding requirements. By defining these test cases in test files, you can periodically check your model and archive results to demonstrate model stability.

Establish Requirements Traceability for Testing

If you have a Simulink Test and a Simulink Verification and Validation license, you can link requirements to test harnesses, test sequences, and test cases. Before adding links, review “Supported Requirements Document Types” on page 3-6 and “Requirements Traceability” in the Simulink Verification and Validation documentation.

Requirements Traceability for Test Harnesses


When you edit requirements links to the component under test, the links immediately synchronize between the test harness and the main model. Other changes to the component under test, such as adding a block, synchronize when you close the test harness. If you add a block to the component under test, close and reopen the harness to update the main model before adding a requirement link.

To view items with requirements links, select **Analysis > Requirements Traceability > Highlight Model**.

Requirements Traceability for Test Sequences

In test sequences, you can link to test steps. To create a link, first find the model item, test case, or location in the document you want to link to. Right-click the test step, select **Requirements Traceability**, and add a link or open the link editor.

To highlight or unhighlight test steps that have requirements links, toggle the

requirements links highlighting button  in the Test Sequence Editor toolbar. Highlighting test steps also highlights the model block diagram.

Requirements Traceability for Test Cases

If you use many test cases with a single test harness, link to each specific test case to distinguish which blocks and test steps apply to it. To link test steps or test harness blocks to test cases,

- 1 Open the test case in the Test Manager.
- 2 Highlight the test case in the test browser.
- 3 Right-click the block or test step, and select **Requirements Traceability > Link to Current Test Case**.

Requirements Traceability Example

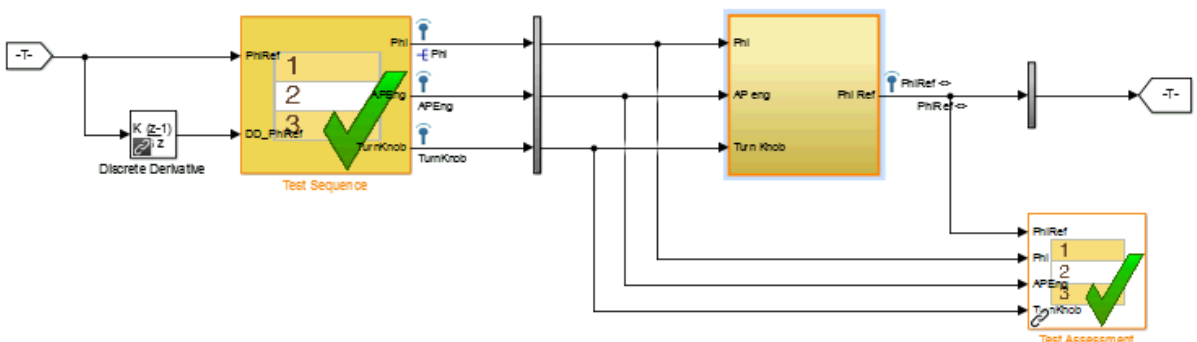
This example demonstrates adding requirements links to a test harness and test sequence. The model is a component of an autopilot roll control system. This example requires Simulink Test and Simulink Verification and Validation.

- 1 Open the test file, the model, and the harness.

```
open AutopilotTestFile.mldatx,
open_system RollAutopilotMdlRef,
sltest.harness.open('RollAutopilotMdlRef/Roll Reference',...
'RollReference_Requirement1_3')
```

- 2 In the test harness, select **Analysis > Requirements Traceability > Highlight Model**.

The test harness highlights the Test Sequence block, component under test, and Test Assessment block.



- 3 Add traceability to the Discrete Derivative block.
 - a Right-click the Discrete Derivative block and select **Requirements Traceability > Open Link Editor**.
 - b In the **Requirements** tab, click **New**.
 - c Enter the following to establish the link:
 - Description: DD link
 - Document type: Text file
 - Document: RollAutopilotRequirements.txt
 - Location: 1.3 Roll Hold Reference

The screenshot shows the 'Requirements Traceability > Open Link Editor' dialog box. It has two tabs: 'Requirements' (selected) and 'Document Index'. On the left side, there are five buttons: 'New', 'Up', 'Down', 'Delete', and 'Copy'. The main area contains a list with one entry, 'DD Link'. Below the list, there are several fields for configuring the link: 'Description:' with the value 'DD Link'; 'Document type:' with a dropdown menu set to 'Text file' and a 'Use current' button; 'Document:' with a dropdown menu set to 'RollAutopilotRequirements.txt' and a 'Browse...' button; 'Location: (Type/Identifier)' with a dropdown menu set to 'Search text' and a text input field containing '1.3 Roll Hold Reference'; and 'User tag:' with an empty dropdown menu.

- d Click **OK**. The Discrete Derivative block highlights.

- 4 To trace to the requirements document, right-click the Discrete Derivative block, and select **Requirements Traceability > DD Link**. The requirements document opens in the editor and highlights the linked text.

1.3 Roll Hold Reference

Navigate to test harness using MATLAB command:

```
web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu
```

REQUIREMENT

1.3.1 When roll hold mode becomes the active mode the roll hold

Navigate to test step using MATLAB command:

```
web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu
```

1.3.1.1. The roll hold reference shall be set to zero if the act

Navigate to test step using MATLAB command:

```
web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu
```

- 5 Open the Test Sequence block. Add a requirements link that links the InitializeTest step to the test case.
 - a In the Test Manager, highlight Requirement 1.3 Test in the test browser.
 - b Right-click the InitializeTest step in the Test Sequence Editor. Select **Requirements Traceability > Link to Current Test Case**.

When the requirements link is added, the Test Sequence Editor highlights the step.

Step	Transition
InitializeTest Phi = 0; APEng = false; TurnKnob = 0; % Initializes test sequence outputs	1. true

Configure RMI for IBM Rational DOORS or Microsoft ActiveX Navigation

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

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

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

You can setup the initial configuration for both cases by running MATLAB as an Administrator and then running this command:

```
rmi setup
```

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

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

```
rmi setup doors
```

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

Requirements Traceability Link Editor

In this section...

“Manage Requirements Traceability Links Using Link Editor” on page 3-17

“Requirements Tab” on page 3-19

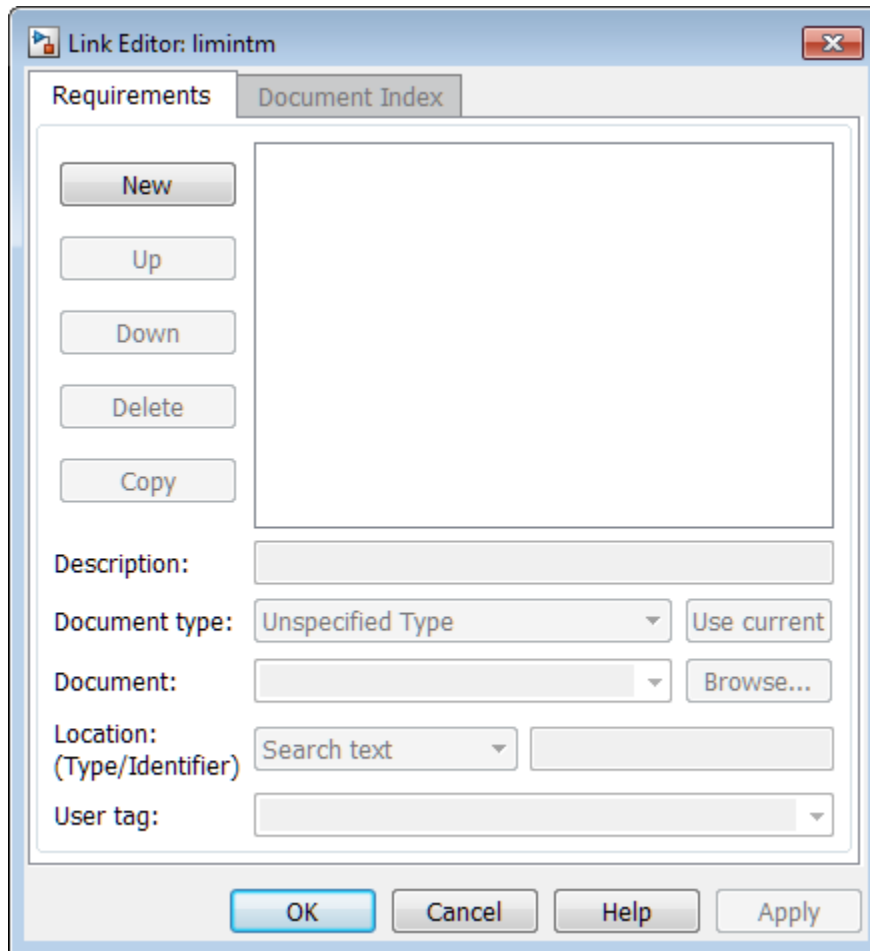
“Document Index Tab” on page 3-19

Manage Requirements Traceability Links Using Link Editor

You can create, edit, and delete requirements traceability links using the Link Editor. To open the Link Editor:

- in the Simulink Editor, right-click on a model object that has a requirements traceability link. From the context menu, select **Requirements Traceability > Open Link Editor**.
- in the MATLAB Editor, right-click inside a region of code that has a requirements traceability link. From the context menu, select **Requirements Traceability > Open Link Editor**.

The Link Editor opens, as shown below.



In the Link Editor, you can:

- Create requirements links from one or more Simulink model objects or MATLAB code lines.
- Customize information about requirements links, including specifying user tags to filter requirements highlighting and reporting.
- Delete existing requirements links.
- Modify the stored order of requirements to control the order of labels in context menus for linked objects.

Requirements Tab

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

- Description of the requirement (up to 255 words). If you create a link using the document index, *unless* a description already exists, the name of the index location becomes the description for the link .
- Path name to the requirements document.
- Document type (Microsoft Word, Microsoft Excel, IBM Rational DOORS, MuPAD, HTML, text file, etc.).
- Location of the requirement (search text, named location, or page or item number).
- User-specified tag or keyword.

Document Index Tab

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

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

Note: The RMI cannot create document indexes for PDF files.

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

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

Requirements Settings

You can manage your RMI preferences in the Requirements Settings dialog box. These settings are global and not associated with any particular model. To open the Requirements Settings dialog box, from the Simulink Editor, select **Analysis > Requirements Traceability > Settings**. In this dialog box, you can select the:

- **Storage** tab to set the default way in which the RMI stores requirements links in a model. For storage information, see “Specify Storage for Requirements Links” on page 4-4.
- **Selection Linking** tab to set the options for linking to the active selection in a supported document. For setting information, see “Selection Linking Tab” on page 3-20.
- **Filters** tab to set the options for filtering requirements in a model. For filtering information, see “Configure Requirements Filtering” on page 5-31.
- **Report** tab to customize the requirements report without using the Report Generator. For setting information, see “Customize Requirements Report Using the RMI Settings” on page 5-17.

Selection Linking Tab

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

Options	Description
For linking to the active selection within an external document:	
Enabled applications	Enable selection-based linking shortcuts to Microsoft Word, Microsoft Excel, or DOORS applications.
Document file reference	Select type of file reference. For information on what settings to use, see “Document Path Storage” on page 6-14.
Apply this user tag to new links	Enter text to attach to the links you create. For more information about user tags, see “Filter Requirements with User Tags” on page 5-25

Options	Description
When creating selection-based links:	
Modify destination for bidirectional linking	Creates links both to and from selected link destination.
Store absolute path to model file	Select type of file reference. For information, see “Document Path Storage” on page 6-14.
Use custom bitmap for navigation controls in documents	Select and browse for your bitmap. You can use your own bitmap file to control the appearance of navigation links in your document.
Use ActiveX buttons in Word and Excel (backward compatibility)	Select to use legacy ActiveX controls to create links in Microsoft Word and Microsoft Excel applications. By default, if not selected, you create URL-based links.

Link Model Objects

In this section...

“Link Objects in the Same Model” on page 3-22

“Link Objects in Different Models” on page 3-22

Link Objects in the Same Model

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

- 1 Right-click the link destination model object and select **Requirements Traceability > Select for Linking with Simulink**.
- 2 Right-click the link source model object and select **Requirements Traceability > Add Link to Selected Object**.
- 3 Right-click the link source model object again and select **Requirements Traceability**. The new link appears at the top of the **Requirements Traceability** submenu.

Link Objects in Different Models

You can create links between objects in related models. This example shows how to link model objects in `slvndemo_powerwindow_controller` and `slvndemo_powerwindow`.

- 1 Open the `slvndemo_powerwindow_controller` and `slvndemo_powerwindow` models.
- 2 In the `slvndemo_powerwindow` model window, double-click the `power_window_control_system` subsystem. The `power_window_control_system` subsystem opens.
- 3 In the `slvndemo_powerwindow/power_window_control_system` subsystem window, right-click the `control` subsystem. Select **Requirements Traceability > Select for Linking with Simulink**.
- 4 In the `slvndemo_powerwindow_controller` model window, right-click the `control` subsystem. Select **Requirements Traceability > Add Link to Selected Object**.

- 5 Right-click the `slvndemo_powerwindow_controller/control` subsystem and select **Requirements Traceability**. The new RMI link appears at the top of the **Requirements Traceability** submenu.
- 6 To verify that the links were created, in the `slvndemo_powerwindow_controller` model window, select **Analysis > Requirements Traceability > Highlight Model**.

The blocks with requirements links are highlighted.

- 7 Close the `slvndemo_powerwindow_controller` and `slvndemo_powerwindow` models.

Link from External Applications

You can navigate to Simulink objects or MATLAB code with requirements using the internal MATLAB HTTP server.

To get the URL for an object in your model, right-click the object and select **Requirements Traceability > Copy URL to Clipboard**. Note that if your model stores requirements traceability data internally, when you select **Requirements Traceability > Copy URL to Clipboard**, the Requirements Management Interface (RMI) creates a unique identifier in the model file. To enable navigation with the URL that you copied, resave your model. To use URLs that do not require changes to your model, configure your model for external storage of requirements traceability data. For more information, see “Move Internally Stored Requirements Links to External Storage” on page 4-7.

To get the URL for a line or lines of MATLAB code, in the MATLAB Editor, select the region of code you want to link to and right-click. From the context menu, select **Requirements Traceability > Copy Link to Clipboard**.

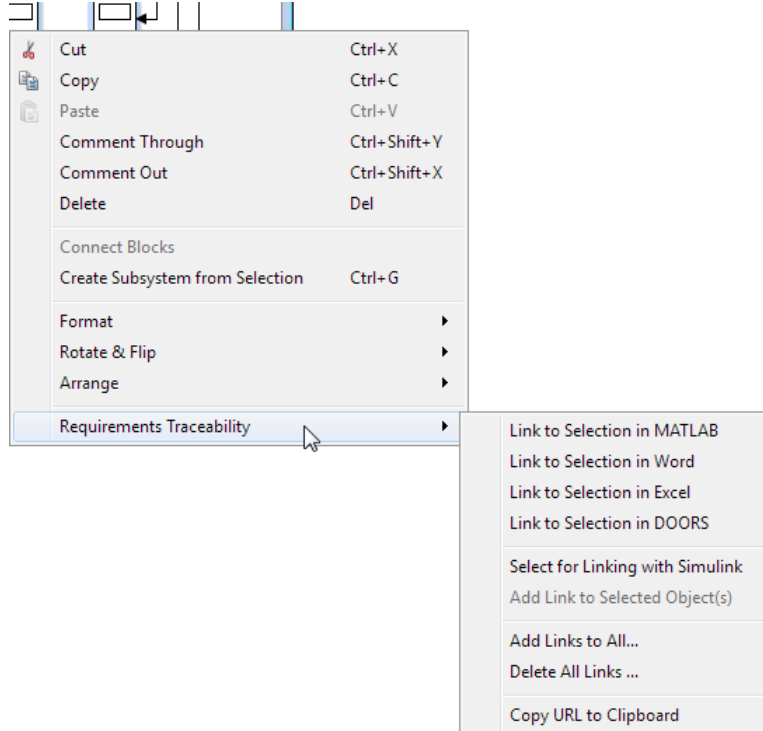
You can then paste the URL into an external application and use it to navigate back to the corresponding object in your model.

To enable navigation with these links, the internal MATLAB HTTP server must be activated on your local host. Selecting **Requirements Traceability > Copy URL to Clipboard** activates the internal HTTP server. You can also enter the command `rmi('httpLink')` at the MATLAB command prompt to activate the internal HTTP server.

Link Multiple Model Objects to a Requirements Document

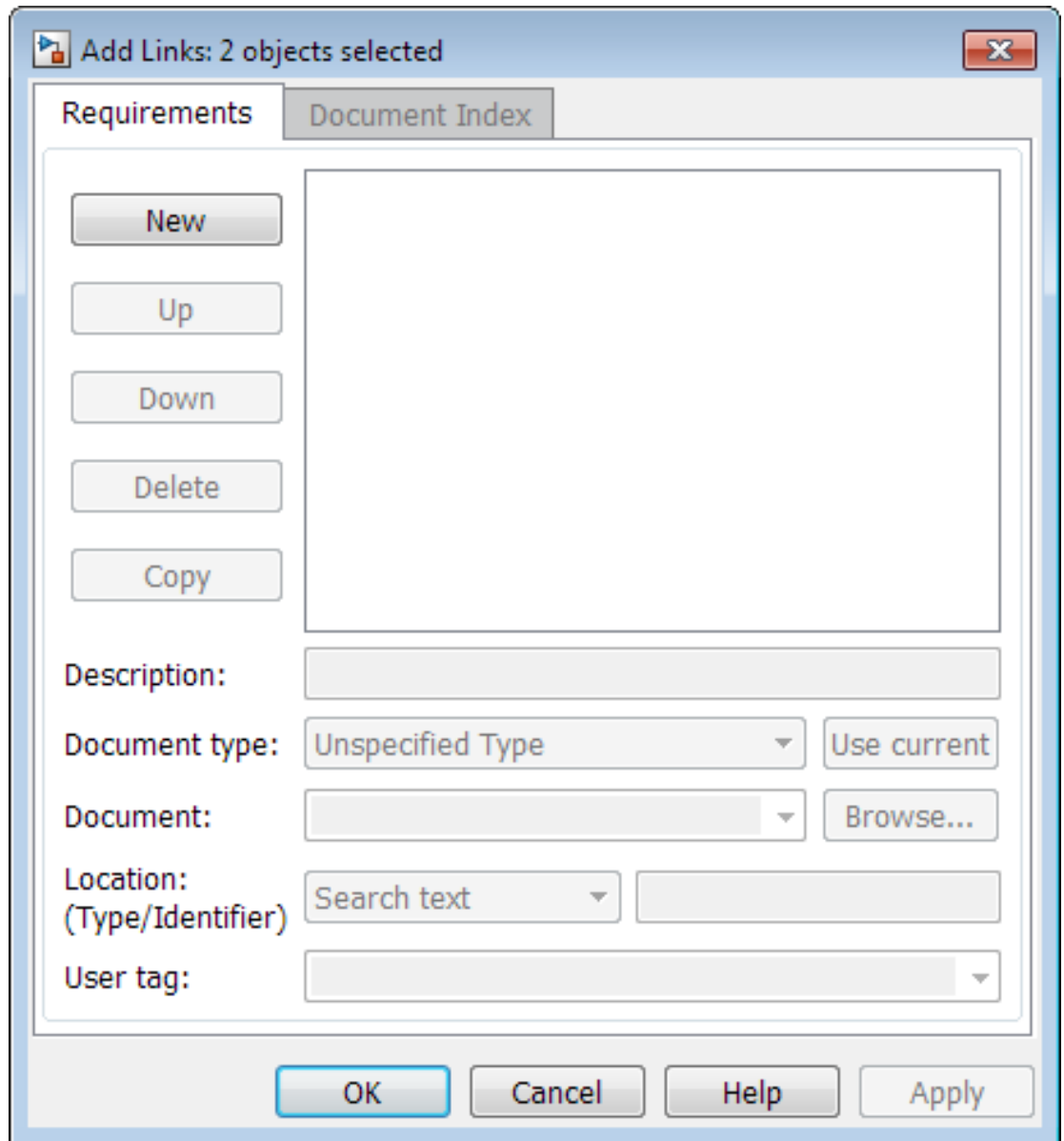
You can link multiple Simulink and Stateflow objects to a requirement. The workflow is:

- 1 In the requirements document, select the requirement.
- 2 In the Simulink Editor or Stateflow Editor, select the objects to link to the selected requirement. You can select multiple objects by holding down the **Shift** key while you click each object that you want to select. Note that you can only select multiple objects in the same diagram.
- 3 Right-click one of the selected objects to open the context menu and hover over **Requirements Traceability**.



- 4 Select one of the selection-based linking options:
 - **Link to Selection in MATLAB**
 - **Link to Selection in Word**

- **Link to Selection in Excel**
 - **Link to Selection in DOORS**
- 5** You can also add and edit links for multiple objects using the Requirements Traceability Link Editor. To open the Link Editor, in the **Requirements Traceability** context menu, select **Add Links to All**.



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

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

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

- 1 Open the example model:

```
slvndemo_fuelsys_officereq
```
- 2 Open a requirements document associated with that model:

```
rmi('view','slvndemo_fuelsys_officereq',1);
```
- 3 Insert a Simulink DocBlock into the model.
- 4 Right-click DocBlock and select **Mask Parameters**. The Block Parameters: DocBlock dialog box opens.
- 5 In the Block Parameters: DocBlock dialog box **Document type** drop-down list, select RTF.
- 6 Click **Apply** or **OK** to create the link.
- 7 In `slvndemo_FuelSys_DesignDescription.docx`, find and select the section titled **2.2 Determination of pumping efficiency**.
- 8 In the `slvndemo_fuelsys_officereq` model window, right-click the DocBlock and select **Requirements Traceability > Link to Selection in Word**.

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

- 9 In the `slvndemo_fuelsys_officereq` model window, right-click the DocBlock and select **Requirements Traceability > Select for Linking with Simulink**.
- 10 In the `slvndemo_fuelsys_officereq/fuel rate controller` subsystem window, select the `control logic` chart and the `Airflow calculation` subsystems. Use the shift key to select both.
- 11 Right-click the `Airflow calculation` subsystem and select **Requirements Traceability > Add Link to Selected Object**.
- 12 In the `slvndemo_fuelsys_officereq` model window, select DocBlock, right-click, and select **Requirements Traceability**.

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

- 13** To verify that the links were created, select **Analysis > Requirements Traceability > Highlight Model**.

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

- 14** Close the `slvnvdemo_fuelsys_officereq` model and the `slvnvdemo_FuelSys_DesignDescription.docx` requirements document.

Link to Requirements in Microsoft Word Documents

In this section...

“Create Bookmarks in a Microsoft Word Requirements Document” on page 3-30

“Open the Example Model and Associated Requirements Document” on page 3-32

“Create a Link from a Model Object to a Microsoft Word Requirements Document” on page 3-32

Create Bookmarks in a Microsoft Word Requirements Document

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

This approach offers advantages to creating new bookmarks:

- You can give the bookmarks meaningful names that represent the content of the requirement.
- When the RMI creates the links, it does not modify your requirements document.

Note: When you link to an existing bookmark, navigating the link highlights the entire range of the existing bookmark. Therefore, when you create a bookmark for requirement linking, make sure to select only the document information relevant to your requirement.

If you have a requirements document containing bookmarks, follow these steps to create requirements links from your Simulink model to the bookmarks:

- 1 Open your model.
- 2 Open your Microsoft Word requirements document that contains bookmarks that identify requirements.
- 3 Right-click a block in the model that you want to link to a requirement and select **Requirements Traceability > Open Link Editor**

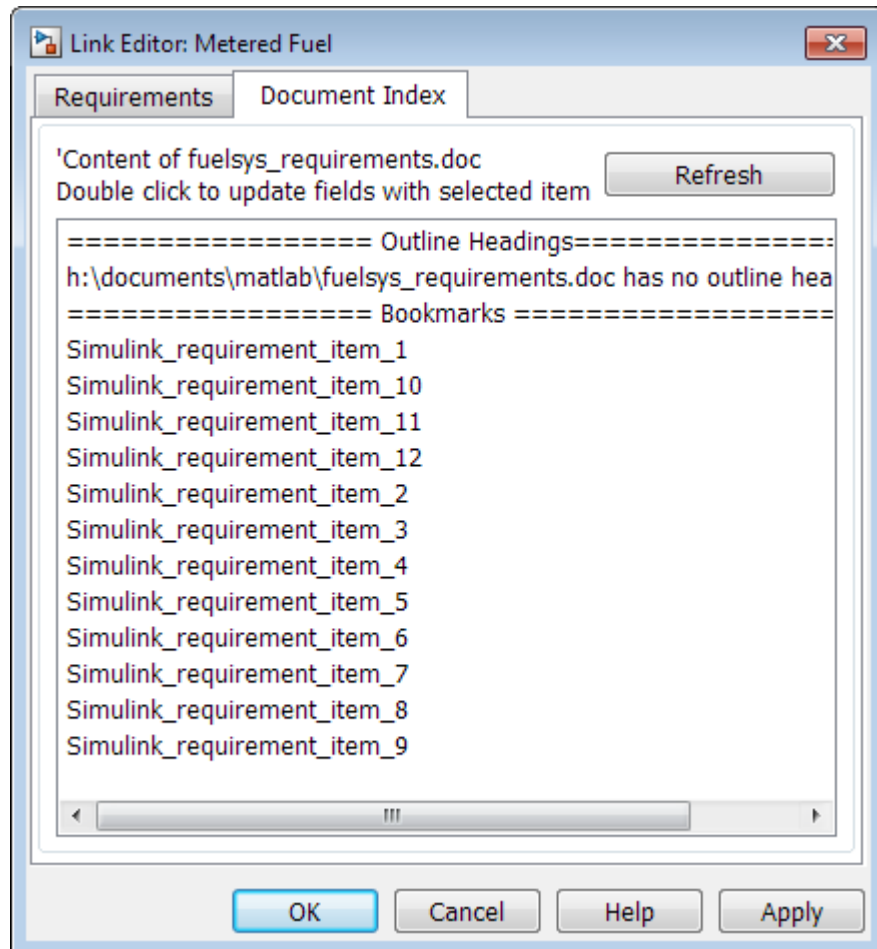
The Requirements Traceability Link Editor opens.

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

- 6 Open the document. The RMI populates the **Document** and **Document type** fields.
- 7 Click the **Document Index** tab of the Link Editor.

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

The document index lists the bookmarks in alphabetical order, not in order of location within the document.



- 8 Select the bookmark that you want to link the block to and click **OK**.

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

Open the Example Model and Associated Requirements Document

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

Navigate from the model to the requirements document:

- 1 Open the example model:

```
slvndemo_fuelsys_officereq
```

- 2 Open a requirements document associated with that model:

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

Keep the example model and the requirements document open.

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

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

- 1 In `slvndemo_FuelSys_DesignDescription.docx`, find the section titled **2.2 Determination of pumping efficiency**.

- 2 Select the header text.

- 3 Open the example model:

```
slvndemo_fuelsys_officereq
```

- 4 Select **Analysis > Requirements Traceability > Settings**. The Requirements Settings dialog box opens.

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

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

- Enable **Modify destination for bidirectional linking**.

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

For more information about the settings, see “Requirements Settings” on page 3-20.

- 6 Double-click the fuel rate controller subsystem to open it.
- 7 Open the Airflow calculation subsystem.
- 8 Right-click the Pumping Constant block and select **Requirements Traceability > Link to Selection in Word**.

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

Note: You cannot link to a Microsoft Word document from a model object if you run either one of MATLAB or Microsoft Word as an Administrator. Run both software with the same privilege level to link to Microsoft Word requirements documents.

- 9 To verify that the link was created, select **Analysis > Requirements Traceability > Highlight Model**.

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

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

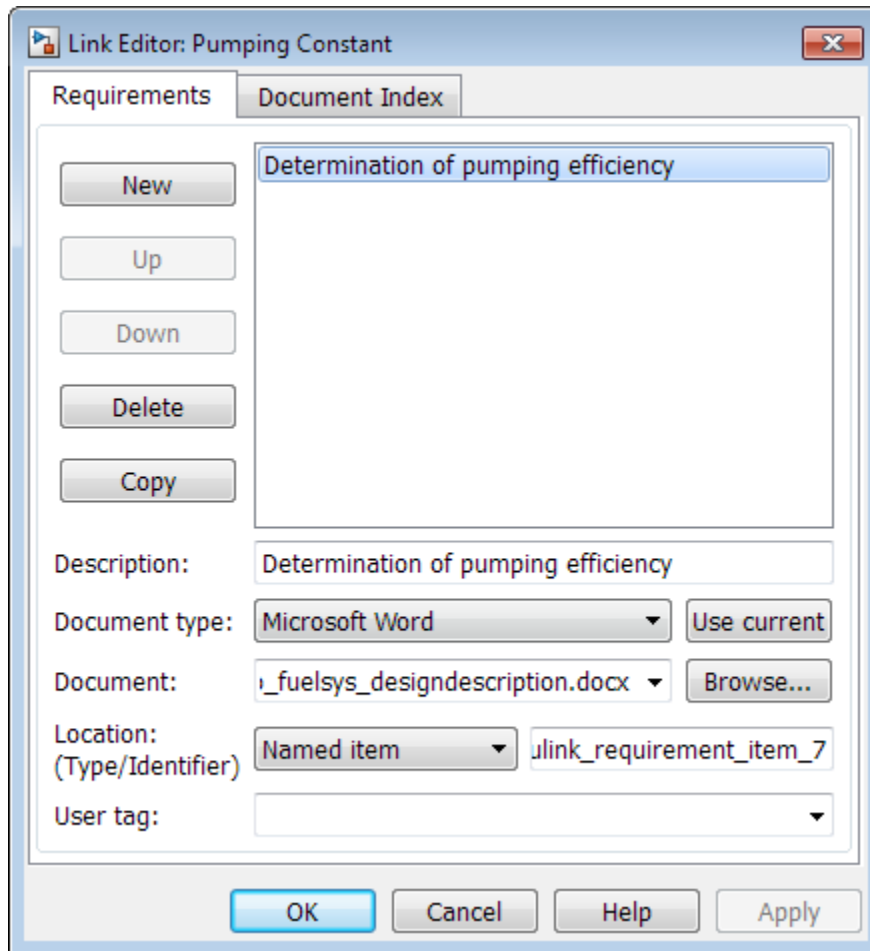
The section **2.2 Determination of pumping efficiency** is displayed, selected in the requirements document.

Keep the example model and the requirements document open.

View Link Details

To view the details of the link that you just created, right-click the Pumping Constant block and select **Requirements Traceability > Open Link Editor**.

The Requirements Traceability Link Editor opens.



This dialog box contains the following information for the link you just created:

- Description of the link, which for selection-based links, matches the text of the selected requirements document, in this case **Determination of pumping efficiency**.
- Name of the requirements document, in this case **slvnvdemo_Fuelsys_DesignDescription.docx**.
- Document type, in this case, **Microsoft Word**.

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

If you do not want the RMI to modify the Microsoft Word requirements document when it creates links, create bookmarks in your Microsoft Word file, as described in “Create Bookmarks in a Microsoft Word Requirements Document” on page 3-30.

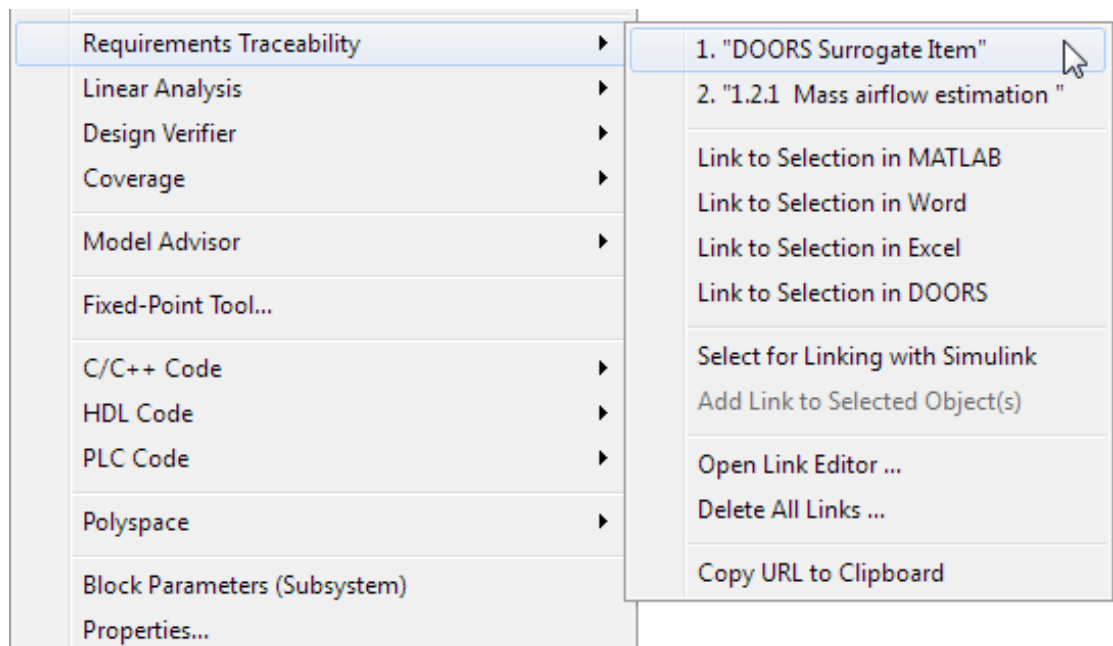
- User tag, a user-defined keyword. This link does not have a user tag.

Note: For more information about user tags, see “Filter Requirements with User Tags” on page 5-25

Link to Requirements in IBM Rational DOORS Databases

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

- 1 Open a DOORS formal module.
- 2 Click to select one of the requirement objects.
- 3 Open the example model:
sldemo_fuelsys
- 4 Open the fuel_rate_control subsystem.
- 5 Right-click the airflow_calc subsystem and select **Requirements Traceability > Link to Selection in DOORS**.
- 6 To confirm the requirement link, right-click the airflow_calc subsystem and select **Requirements Traceability**. In the submenu, the top item is the heading text for the DOORS requirement object.



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

Note: To view an example of using the RMI with an IBM Rational DOORS database, run the Managing Requirements for Fault-Tolerant Fuel Control System (IBM Rational DOORS) example at the MATLAB command prompt.

Link to Requirements in Microsoft Excel Workbooks

In this section...

“Navigate from a Model Object to Requirements in a Microsoft Excel Workbook” on page 3-38

“Create Requirements Links to the Workbook” on page 3-38

“Link Multiple Model Objects to a Microsoft Excel Workbook” on page 3-39

“Change Requirements Links” on page 3-40

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

- 1 Open the example model:
`slvndemo_fuelsys_officereq`
- 2 Select **Analysis > Requirements Traceability > Highlight Model** to highlight the model objects with requirements.
- 3 Right-click the Test inputs Signal Builder block and select **Requirements Traceability > 1. “Normal mode of operation”**.

The `slvndemo_FuelSys_TestScenarios.xlsx` file opens, with the associated cell highlighted.

Keep the example model and Microsoft Excel requirements document open.

For information about creating requirements links in Signal Builder blocks, see “Link Signal Builder Blocks to Requirements Documents” on page 3-59.

Create Requirements Links to the Workbook

- 1 At the top level of the `slvndemo_fuelsys_officereq` model, right-click the speed sensor block and select **Requirements Traceability > Open Link Editor**.

The Requirements Traceability Link Editor opens.

- 2 To create a requirements link, click **New**.
- 3 In the **Description** field, enter:

Speed sensor failure

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

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

For information about which setting to use for your working environment, see “Document Path Storage” on page 6-14.

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

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

- 6 In the workbook, the **Speed sensor failure** information is in cells B22:E22. For the **Location (Type/Identifier)** field, select **Sheet** range and in the second field, enter B22:E22. (The cell range letters are not case sensitive.)
- 7 Click **Apply** or **OK** to create the link.
- 8 To confirm that you created the link, right-click the speed sensor block and select **Requirements Traceability > 1. “Speed sensor failure”**.

The workbook opens, with cells B22:E22 highlighted.

Keep the example model and Microsoft Excel requirements document open.

Link Multiple Model Objects to a Microsoft Excel Workbook

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

- 1 In the model window, select the objects to link to a requirement.
- 2 Right-click one of the selected objects and select **Requirements Traceability > Open Link Editor**.
- 3 When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

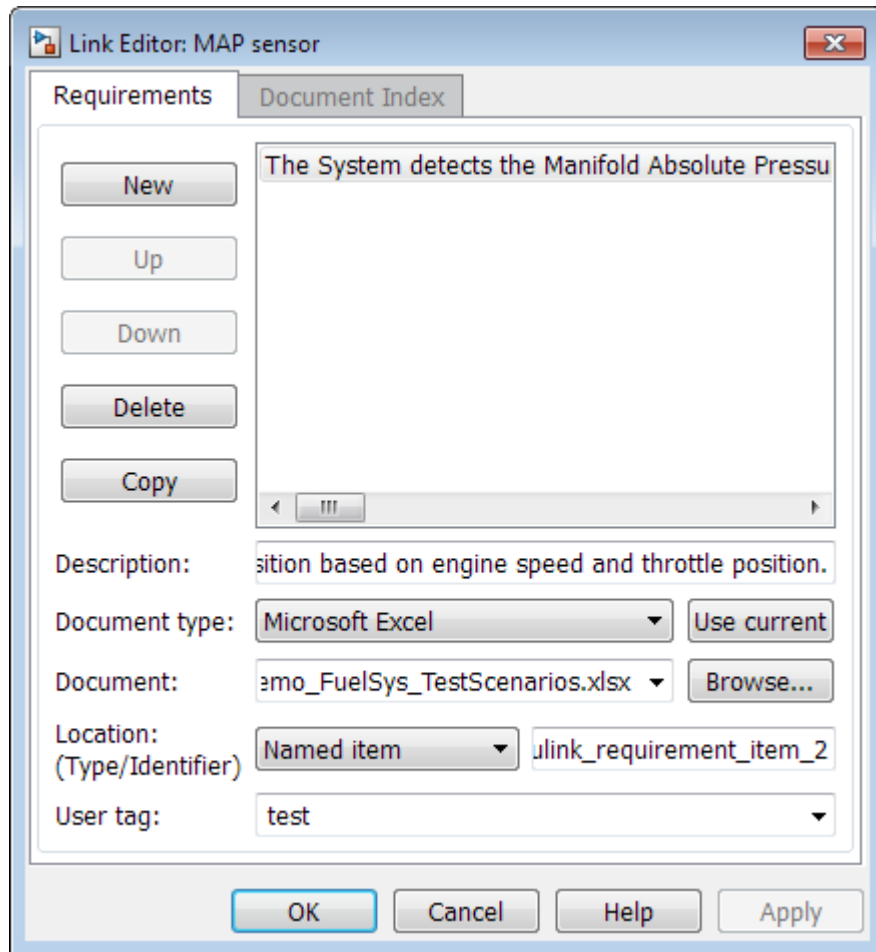
For information about which setting to use for your working environment, see “Document Path Storage” on page 6-14.

- 4 Use the Link Editor to specify information about the Microsoft Excel requirements document, the requirement, and the link.
- 5 Click **Apply** or **OK** to create the link.

Change Requirements Links

- 1 In the `slvndemo_fuelsys_officereq` model, right-click the MAP sensor block and select **Requirements Traceability > Open Link Editor**.

The Requirements Traceability Link Editor opens displaying the information about the requirements link.



- 2 In the **Description** field, enter:

MAP sensor test scenario

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

Note: For more information about user tags, see “Filter Requirements with User Tags” on page 5-25.

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

Keep the example model open.

Link to Requirements in MuPAD Notebooks

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

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

Note: You must have the Symbolic Math Toolbox™ installed on your system to open a MuPAD notebook. For information about creating or opening MuPAD notebooks, see “Create MuPAD Notebooks” and “Open MuPAD Notebooks”.

- 1 Open an example model for the van der Pol equation:

vdp

- 2 Right-click a blank area of the model and select **Requirements Traceability at This Level > Open Link Editor**.

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

- 3 Click **New**.
- 4 In the **Document type** drop-down list, select **MuPAD Notebook**.
- 5 When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

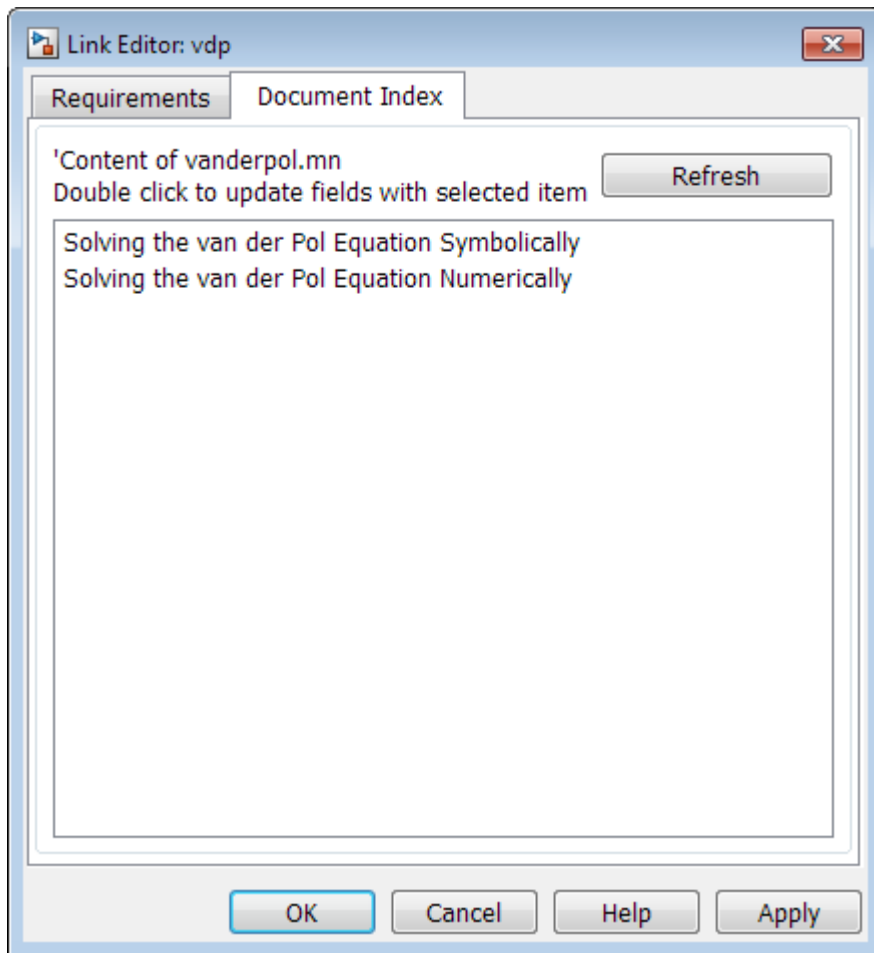
For information about which setting to use for your working environment, see “Document Path Storage” on page 6-14.

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

Use a notebook that has link targets in it.

- 7 Make sure that the MuPAD notebook is in a saved state. Any link targets created since the last save do not appear in the RMI document index.
- 8 To list the link targets, in the Requirements Traceability Link Editor, click the **Document Index** tab.

This figure shows two link targets.



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

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

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

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

Create Requirements Traceability Report for Model

Abstract

Create the default requirements report for a Simulink model.

To create the default requirements report for a Simulink model:

- 1 Open the example model:

```
slvndemo_fuelsys_officereq
```
- 2 Make sure that your current working folder is writable.
- 3 In the Simulink Editor, select **Analysis > Requirements Traceability > Generate Report**.

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

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

Requirements Report for slvndemo_fuelsys_officereq

username

17-Jun-2010 10:57:04

Table of Contents

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

A typical requirements report includes:

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

For detailed information about requirements reports, see “Customize Requirements Traceability Report for Model” on page 5-7.

If Your Model Has Library Reference Blocks

To include requirements links associated with library reference blocks, you must select **Include links in referenced libraries and data dictionaries** under the **Report** tab of the **Requirements Settings**, as described in “Customize Requirements Report” on page 5-17.

If Your Model Has Model Reference Blocks

By default, requirements links within model reference blocks in your model are not included in requirements traceability reports. To generate a report that includes requirements information for referenced models, follow the steps in Report for Requirements in Model Blocks on page 5-15.

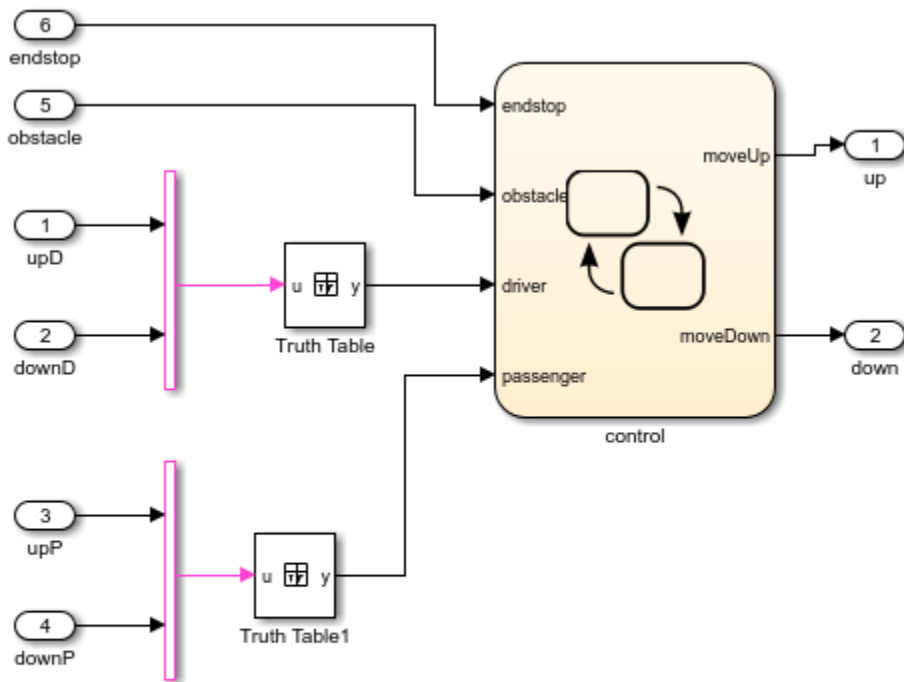
Link to Requirements Modeled in Simulink

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

Open Example Model

Open the Power Window Controller model by typing the command:

```
open_system('slvndemo_powerwindowController');
```



Copyright 1990-2010 The MathWorks, Inc.

Verification Subsystems for Power Window Controller Model

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

Consider the following design requirements for the controller:

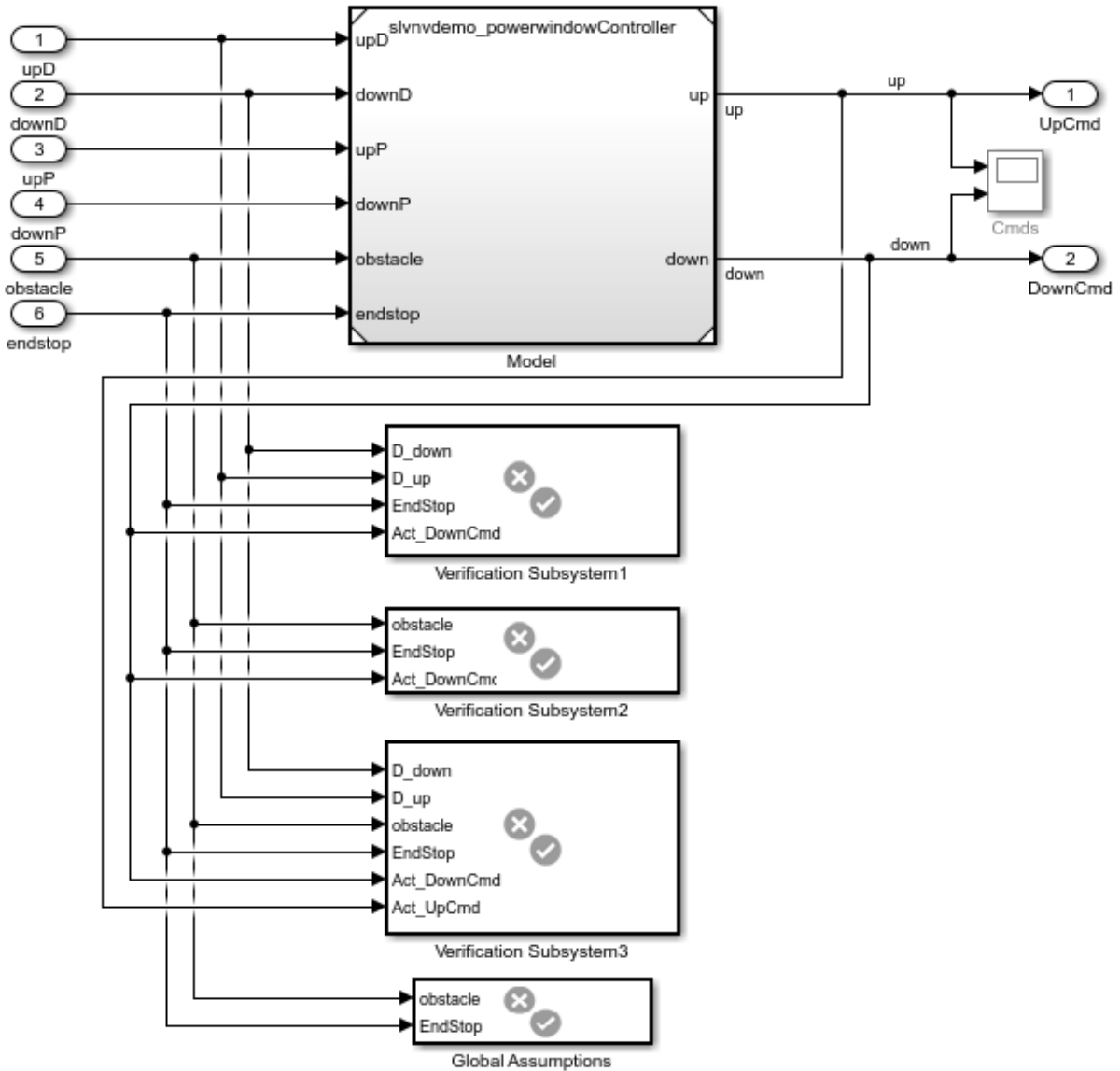
- 1 **Requirement 1** (Obstacle Response) - Whenever an obstacle is detected, the controller shall give the down command for one second. This requirement is modeled in Verification Subsystem2.

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

See Design Verifier Temporal Properties example for more details.

```
open_system('slvndemo_powerwindow_vs');
```

Power Window Controller Temporal Property Specification



Copyright 1990-2010 The MathWorks, Inc.

Create RMI Link to a Simulink Object

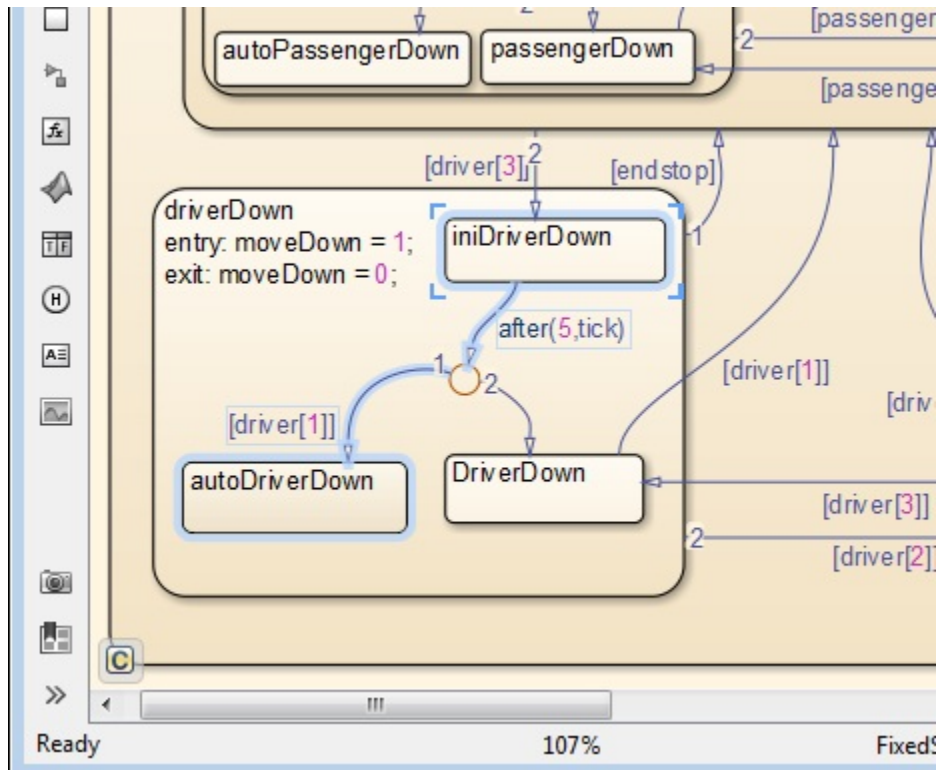
Create an RMI link from Verification Subsystem2 to the emergencyDown state in the slvndemo_powerwindowController model.

- 1 Open slvndemo_powerwindowController model.
- 2 Right-click on emergencyDown state and select **Requirements Traceability > Select for linking with Simulink**.
- 3 Right-click on Verification Subsystem2 and select **Requirements Traceability > Add link to selected object**.
- 4 Right-click the Verification Subsystem2. The new RMI link appears at the top of **Requirements Traceability** submenu.
- 5 Close slvndemo_powerwindowController model.
- 6 Right-click on Verification Subsystem2. Navigate the new link at the top of the **Requirements Traceability** submenu. Model opens and emergencyDown state is highlighted.

Link Simultaneously to Multiple Simulink Objects

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

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



- 1 Right-click on this group of objects, select **Requirements Traceability > Select for linking with Simulink**. Be careful to not lose the selections when you right-click.
- 2 Right-click on Verification Subsystem3 and select **Requirements Traceability > Link to 4 selected objects**.

Link to a Group of Simulink Objects

- 1 Right-click on NAND block in Global Assumptions and select **Requirements Traceability > Select for linking with Simulink**.
- 2 Drag the mouse across endstop and obstacle inputs in `slvndemo_powerwindowController` to select both inputs.
- 3 Right-click on this group of objects and select **Add link to selected object**. Be careful to not lose the selection.

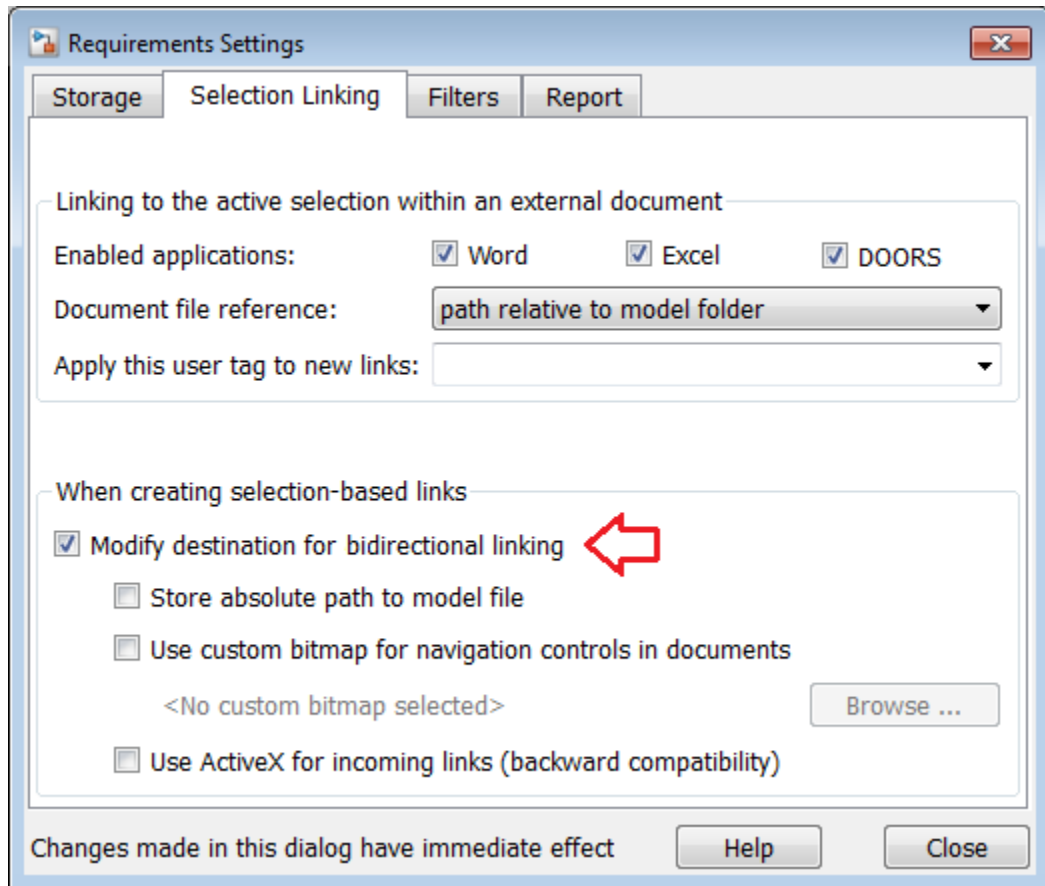
- 4 Click on the background of `slvndemo_powerwindowController` to clear the group selection.
- 5 Right-click each input and select **Requirements Traceability** to display new links. Click the new link, confirm that NAND is highlighted.

Create Links for Navigation in Both Directions

To create links for navigation in both directions:

- 1 Open Requirements Settings dialog box.
- 2 Select the Selection Linking tab.
- 3 Enable **Modify destination for bidirectional linking**.

Now, when you create a link from one Simulink object to another, a corresponding "return" link is also created.



Highlight and Report RMI Links Between Simulink Objects

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

- 1 In the `slvndemo_powerwindow_vs` model window, select **Analysis > Requirements Traceability > Highlight Model** to highlight all RMI links in the model, including links to Simulink objects.
- 2 In the `slvndemo_powerwindow_vs` model window, select **Analysis > Requirements Traceability > Generate Report**.
- 3 In the generated report, click a hyperlink in any requirements table. This navigates to the corresponding object in Simulink diagram.

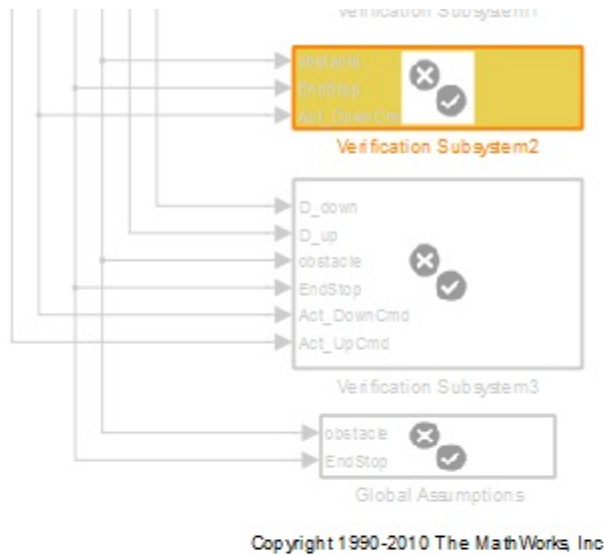


Table 3.1. Blocks in "slvndemo_powerwindow_vs" that have requirements

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

Cleanup

Close all open models. Do not save changes.

```
close_system('slvndemo_powerwindowController', 0);
close_system('slvndemo_powerwindow_vs', 0);
```

Requirements Linking with Simulink Annotations

You can create requirements links to and from `Simulink.Annotation` objects using the Requirements Management Interface (RMI). Annotations are free-floating text boxes that you can place inside a Simulink model. For more information, see “Describe Models Using Annotations” in the Simulink documentation. You can use RMI linking to associate annotations with other Simulink objects or external requirements documents. Note that requirements linking for annotations is not supported in Stateflow.

Requirements linking for Simulink annotations is enabled only if you configure your model to store requirements data externally. To specify external storage of requirements data for a model, in the Requirements Settings dialog box under **Storage > Default storage location for requirements links data**, select **Store externally (in a separate *.req file)**. If you have an existing model that contains internally stored requirements links, you must convert the model to store requirements data externally before you can use requirements linking for annotations in the model. To switch the model from internal to external RMI storage, move all existing requirements links in it to an external `.req` file by selecting **Analysis > Requirements Traceability > Move to File**. For more information on requirements links storage, see “Specify Storage for Requirements Links” on page 4-4.

Note: If you later change your model to store requirements links internally as described in “Move Externally Stored Requirements Links to the Model File” on page 4-8, or if you save the model in a version of MATLAB prior to R2012b, annotation requirements links are either discarded or attached to a corresponding parent diagram.

Annotations are context-specific. Unlike copying other Simulink objects, copying an annotation does not copy its associated requirements links. Requirements links from annotations do not appear in generated code, but requirements links to annotations from other Simulink objects do appear.

Link Signal Builder Blocks to Requirements Documents

Abstract

Create links from a signal group in a Signal Builder block to a requirements document.


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

- 1 Open the model:

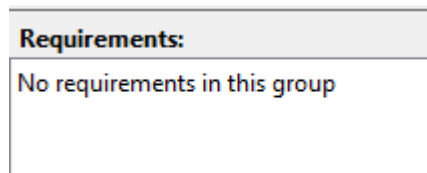
`sf_car`

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

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

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

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



- 4 Place your cursor in the window, right-click, and select **Open Link Editor**.

The Requirements Traceability Link Editor opens.

- 5 Click **New**. In the **Description** field, enter `User input requirements`.
- 6 When you browse and select a requirements document, the RMI stores the document path as specified by the **Document file reference** option on the Requirements Settings dialog box, **Selection Linking** tab.

For information about which setting to use for your working environment, see “Document Path Storage” on page 6-14.

- 7 Browse to a requirements document and click **Open**.
- 8 In the **Location** drop-down list, select **Search text** to link to specified text in the document.
- 9 Next to the **Location** drop-down list, enter `User Input Requirements`.
- 10 Click **Apply** to create the link.
- 11 To verify that the RMI created the link, in the Simulink Editor, select the User Inputs block, right-click, and select **Requirements Traceability**.

The link to the new requirement is the option at the top of the submenu.

- 12 Save the `sf_car_linking` model.

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


In this example, switch to a different active group in the drop-down list to link a requirement to another signal group.

Link Signal Builder Blocks to Model Objects

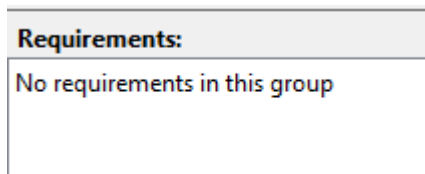
This example shows how to create links from a signal group in a Signal Builder block to a model object:

- 1 Open the `sf_car` model.
- 2 Open the `sf_car/shift_logic` chart.
- 3 Right-click `upshifting` and select **Requirements Traceability > Select for Linking with Simulink**.
- 4 In the `sf_car` model window, double-click the User Inputs block.

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

- 5 In the Signal Builder dialog box, click the **Gradual Acceleration** tab.
- 6 At the far-right end of the toolbar, click the **Show verification settings** button . (You might need to expand the Signal Builder dialog box for this button to become visible.)

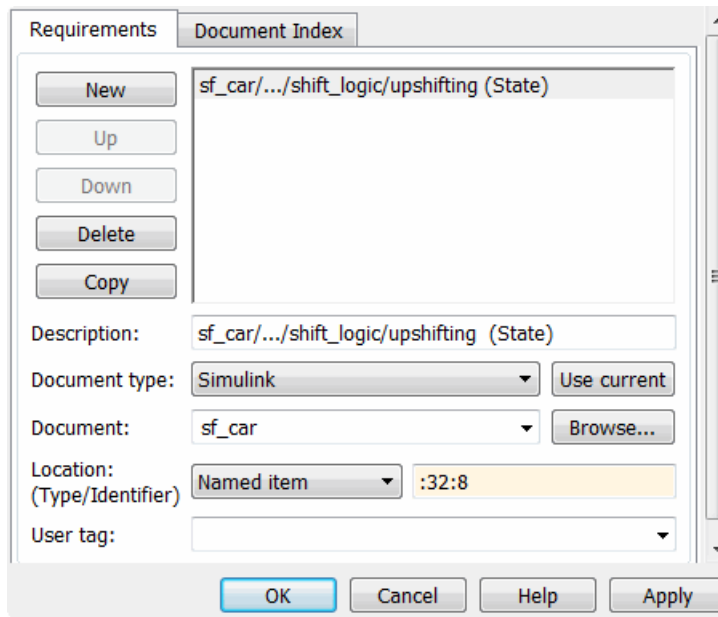
A **Requirements** pane opens on the right-hand side of the Signal Builder dialog box.



- 7 Place your cursor in the window, right-click, and select **Open Link Editor**.

The Requirements Traceability Link Editor opens.

- 8 Click **New**. In the **Description** field, enter `Upshifting`.
- 9 In the **Document type** field, select `Simulink`. Click **Use current**. The software fills in the field with the **Location: (Type/Identifier)** information for `upshifting`.



- 10 Click **Apply** to create the link.
- 11 In the model window, select the User Inputs block, right-click, and select **Requirements Traceability**.

The link to the new requirement is the option at the top of the submenu.

- 12 To verify that the links were created, in the `sf_car` model window, select **Analysis > Requirements Traceability > Highlight Model**.

The blocks with requirements links are highlighted.

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

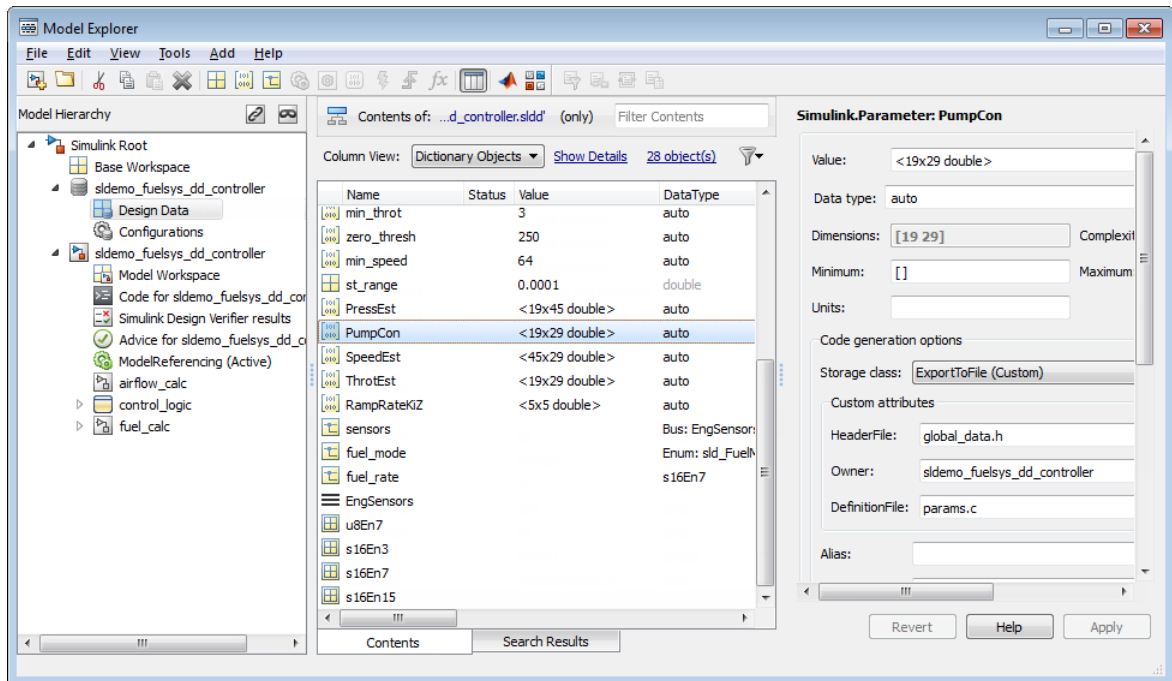
- 13 Close the `sf_car` model.

Link Requirements to Simulink Data Dictionary Entries

You can create requirements traceability links for entries in Simulink data dictionaries. The process is similar to linking for other model objects. In the Model Explorer, right-click a data dictionary entry, select **Requirements Traceability**, and choose one of the selection-based linking options. You can also use the Link Editor.

This example demonstrates linking to a data dictionary entry.

- 1 Enter `sldemo_fuelsys_dd_controller` at the command line to open `sldemo_fuelsys_dd_controller`.
- 2 Open the linked data dictionary by clicking the data dictionary badge in the bottom left corner of the model.
- 3 In the **Model Hierarchy** pane of the Model Explorer, select **Design Data** in the `sldemo_fuelsys_dd_controller` data dictionary.
- 4 You will link the `PumpCon` parameter to the `Pumping Constant` lookup table in the model.



- 5 Open the `airflow_calc` subsystem and select the Pumping Constant lookup table.
- 6 In the Model Explorer, right-click the `PumpCon` parameter and select **Requirements Traceability > Link to Selection in Simulink**.

The two objects are linked.

- 7 Check the link. Right-click the `PumpCon` parameter and select **Requirements Traceability**, then select the navigation shortcut at the top of the **Requirements Traceability** submenu. Simulink highlights the lookup table.

How Is Requirements Link Information Stored?

- “External Storage” on page 4-2
- “Guidelines for External Storage of Requirements Links” on page 4-3
- “Specify Storage for Requirements Links” on page 4-4
- “Save Requirements Links in External Storage” on page 4-5
- “Load Requirements Links from External Storage” on page 4-6
- “Move Internally Stored Requirements Links to External Storage” on page 4-7
- “Move Externally Stored Requirements Links to the Model File” on page 4-8

External Storage

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

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

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

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

To see an example of the external storage capability using a Simulink model, at the MATLAB command line, enter:

```
slvndemo_powerwindow_external
```

Guidelines for External Storage of Requirements Links

Follow these guidelines when storing requirements links in an external file.

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

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

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

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

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

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

Specify Storage for Requirements Links

Default storage mode for traceability data

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

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

To specify the requirements storage setting:

- 1 In the model window, select **Analysis > Requirements Traceability > Settings**.
- 2 In the Requirements Settings dialog box, select the **Storage** tab.
- 3 Under **Default storage location for requirements links data**:
 - To enable internal storage, select **Store internally (embedded in model file)**.
 - To enable external storage, select **Store externally (in a separate *.req file)**.

These settings go into effect immediately.

Duplicating outgoing links when copying Simulink and Stateflow objects

By default, the Requirements Management Interface (RMI) always duplicates requirements links in copied Simulink and Stateflow objects. To only duplicate requirements links when you highlight model requirements before copying objects, select **Duplicate links only when model requirements are highlighted**.

Save Requirements Links in External Storage

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

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

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

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

Load Requirements Links from External Storage

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

To explicitly load requirements links from an external file:

- 1 Select **Analysis > Requirements Traceability > Load Links**.

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

- 2 Select the file from which to load the requirements links.
- 3 Click **Open** to load the links from the selected file.

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

Move Internally Stored Requirements Links to External Storage

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

- 1 Open the model that contains internally stored requirements links.
- 2 Select **Analysis > Requirements Traceability > Move to File**.

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

- 3 Accept the default name, or enter a different file name if required.
- 4 Click **Save**.

Note: Use the default name for externally stored requirements. For more information about this recommendation, see “Guidelines for External Storage of Requirements Links” on page 4-3.

Move Externally Stored Requirements Links to the Model File

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

- 1 Open the model that has only externally stored requirements links.
- 2 Make sure the right set of requirements links are loaded from the external file.
- 3 Select **Analysis > Requirements Traceability > Copy to Model**.

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

- 4 Save the model with the requirements links.

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

Reviewing Requirements

- “Highlight Model Objects with Requirements” on page 5-2
- “Navigate to Requirements from Model” on page 5-5
- “Customize Requirements Traceability Report for Model” on page 5-7
- “Filter Requirements with User Tags” on page 5-25
- “Create Requirements Traceability Report for Simulink Project” on page 5-33
- “View Requirements Details for a Selected Block” on page 5-34

Highlight Model Objects with Requirements

Abstract

Highlight a model to see which objects in the model have links to requirements in external documents.

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

In this section...

“Highlight Model Objects with Requirements Using Model Editor” on page 5-2

“Highlight Model Objects with Requirements Using Model Explorer” on page 5-3

Highlight Model Objects with Requirements Using Model Editor

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

- 1 Open the example model:

```
slvndemo_fuelsys_officereq
```

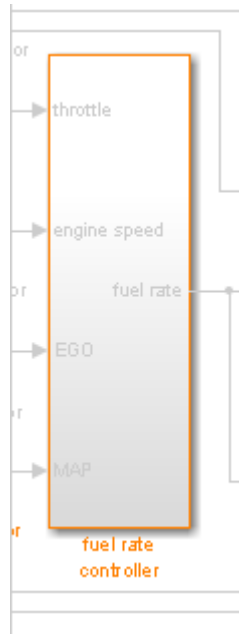
- 2 Select **Analysis > Requirements Traceability > Highlight Model**.

Two types of highlighting indicate model objects with requirements:

- Yellow highlighting indicates objects that have requirements links for the object itself.



- Orange outline indicates objects, such as subsystems, whose child objects have requirements links.



Objects that do not have requirements are colored gray.




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

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

Highlight Model Objects with Requirements Using Model Explorer

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

- 1 Open the example model:
`slvndemo_fuelsys_officereq`
- 2 Select **View > Model Explorer**.
- 3 To highlight all model objects with requirements, click the **Highlight items with requirements on model** icon ()

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

Note: If you are running a 64-bit version of MATLAB, when you navigate to a requirement in a PDF file, the file opens at the beginning of the document, not at the specified location.

Navigate to Requirements from Model

In this section...

“Navigate from Model Object” on page 5-5

“Navigate from System Requirements Block” on page 5-5

Navigate from Model Object

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

- 1 Open the example model:
`slvndemo_fuelsys_officereq`
- 2 Open the fuel rate controller subsystem.
- 3 To open the linked requirement, right-click the Airflow calculation subsystem and select **Requirements Traceability > 1. “Mass airflow estimation”**.

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

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

Navigate from System Requirements Block

Sometimes you want to see all the requirements links at a given level of the model hierarchy. In such cases, you can insert a System Requirements block to collect all requirements links in a model or subsystem. The System Requirements block lists requirements links for the model or subsystem in which it resides; it does not list requirements links for model objects inside that model or subsystem, because those are at a different level of the model hierarchy.

In the following example, you insert a System Requirements block at the top level of the `slvndemo_fuelsys_officereq` model, and navigate to the requirements using the links inside the block.

- 1 Open the example model:

`slvndemo_fuelsys_officereq`

- 2 In the Simulink Editor, select **Analysis > Requirements Traceability > Highlight Model**.
- 3 Open the fuel rate controller subsystem.

The Airflow calculation subsystem has a requirements link.

- 4 Open the Airflow calculation subsystem.
- 5 In the Simulink Editor, select **View > Library Browser**.
- 6 On the **Libraries** pane, select **Simulink Verification and Validation**.

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

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

The RMI software collects and displays any requirements links for that subsystem in the System Requirements block.

- 8 In the System Requirements block, double-click 1. “**Mass airflow subsystem**”.

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

Customize Requirements Traceability Report for Model

In this section...

“Create Default Requirements Report” on page 5-7

“Report for Requirements in Model Blocks” on page 5-15

“Customize Requirements Report” on page 5-17

“Generate Requirements Reports Using Simulink” on page 5-23

Create Default Requirements Report

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

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

Note: If the model for which you are creating a report contains Model blocks, see “Report for Requirements in Model Blocks” on page 5-15.

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

- 1 Open the example model:

```
slvndemo_fuelsys_officereq
```

- 2 Open the fuel rate controller subsystem.

- 3 Open the Microsoft Word requirements document:

```
matlabroot/toolbox/slvnv/rmidemos/fuelsys_req_docs/...  
slvndemo_FuelSys_RequirementsSpecification.docx
```

- 4 Create a link from the control logic Stateflow chart to a location in this document.
- 5 Keep the example model open, but close the requirements document.

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

1 Select **Analysis > Requirements Traceability > Generate Report.**

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

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

The report contains the following content:

- “Table of Contents” on page 5-8
- “List of Tables” on page 5-9
- “Model Information” on page 5-10
- “Documents Summary” on page 5-10
- “System” on page 5-11
- “Chart” on page 5-14

Table of Contents

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

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

Requirements Report for slvndemo_fuelsys_officereq

username

17-Jun-2010 10:57:04

Table of Contents

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

List of Tables

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

List of Tables

- 1.1. [slvndemo_fuelsys_officereq Version Information](#)
- 2.1. [Requirements documents linked in model](#)
- 3.1. [slvndemo_fuelsys_officereq Requirements](#)
- 3.2. [Blocks in "slvndemo_fuelsys_officereq" that have requirements](#)
- 3.3. [Test inputs : Normal operation signal requirements](#)
- 4.1. [Blocks in "engine gas dynamics" that have requirements](#)
- 5.1. [Blocks in "fuel rate controller" that have requirements](#)
- 6.1. [Blocks in "Mixing & Combustion" that have requirements](#)
- 7.1. [slvndemo_fuelsys_officereq/fuel rate controller/Airflow calculation Requirements](#)
- 7.2. [Blocks in "Airflow calculation" that have requirements](#)
- 8.1. [Blocks in "Sensor correction and Fault Redundancy" that have requirements](#)
- 9.1. [slvndemo_fuelsys_officereq/fuel rate controller/Sensor correction and Fault Redundancy/MAP Estimate Requirements](#)
- 10.1. [Stateflow objects with requirements](#)

Model Information

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

Chapter 1. Model Information for "slvnvdemo_fuelsys_officereq"

Table 1.1. slvnvdemo_fuelsys_officereq Version Information

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

Documents Summary

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

Chapter 2. Document Summary for "slvnvdemo_fuelsys_officereq"

Table 2.1. Requirements documents linked in model

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

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

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

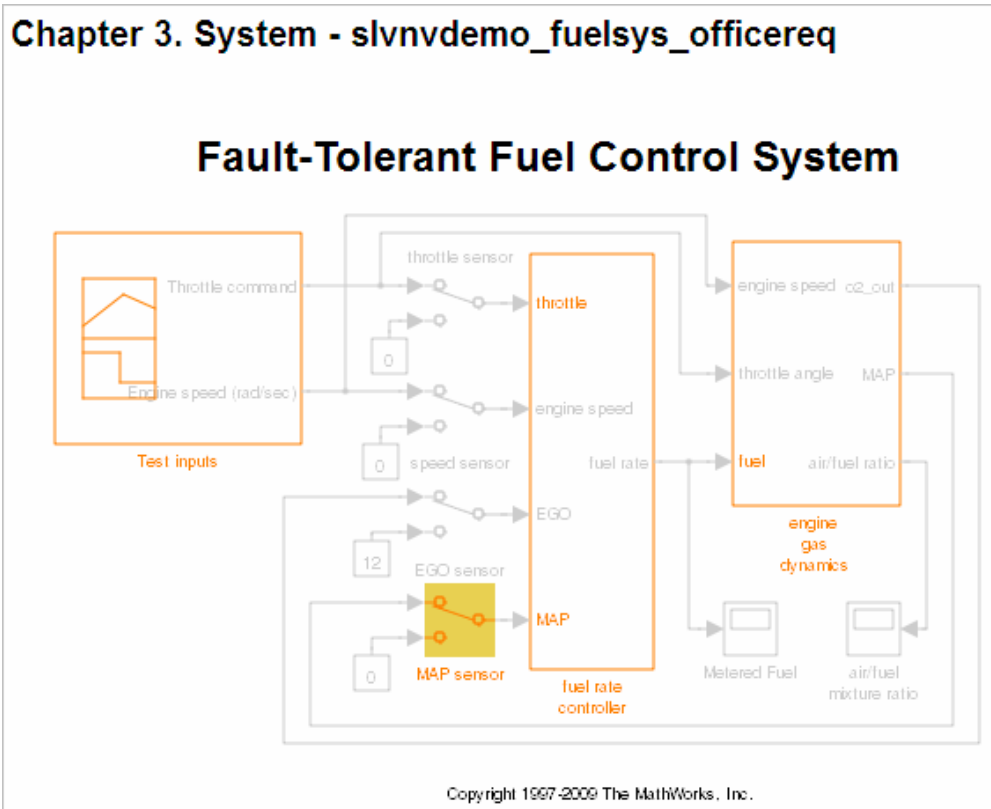
When your requirements documents have long path names that can clutter the report, select the **User document IDs in requirements tables** option. This option is disabled by default, as you can see in the examples in this section.

- **Document paths stored in the model** — Click this link to open the requirements document in its native application.
- **Last modified** — The date the requirements document was last modified.
- **# links** — The total number of links to a requirements document.

System

Each **System** section includes:

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



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

Table 3.1. slvndemo_fuelsys_officereq Requirements

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

- A list of blocks in the top-level model that have requirements. In this example, only the MAP sensor block has a requirement at the top level. Click the link next to **Target:** to open the requirements document associated with the MAP sensor block.

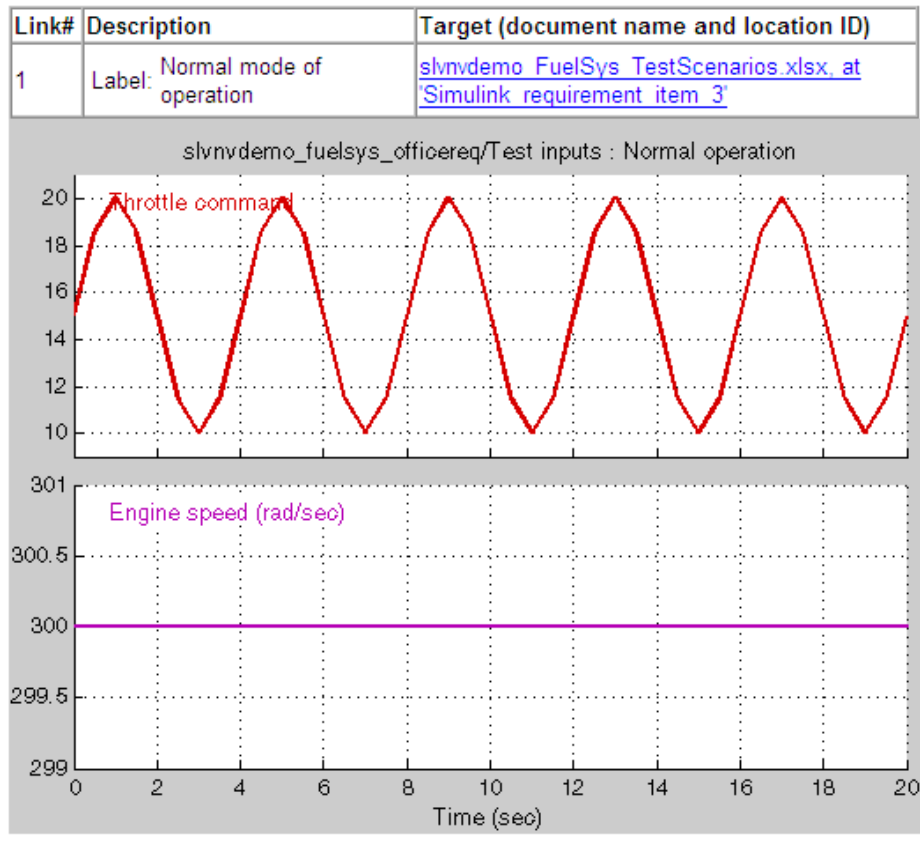
Table 3.2. Blocks in "slvndemo_fuelsys_officereq" that have requirements

Name	Requirements
MAP sensor	<p>Link#1 label: "The System detects the Manifold Absolute Pressure sensor short to ground or open circuit by monitoring when the signal is below a calibratable threshold. The failure is detected within 100mSec of the occurrence and the MAP sensor reading is reverted to an estimated throttle position based on engine speed and throttle position."</p> <p>Target: fuelsys_req_docs\slvndemo_FuelSys_TestScenarios.xlsx, at 'Simulink requirement item 2'</p>

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

- The fuel rate controller and engine gas dynamics subsystems are in dedicated chapters of the report.
- The report lists Signal Builder blocks separately, in this example, in Table 3.3.
- A list of requirements associated with each signal group in any Signal Builder block, and a graphic of that signal group. In this example, the Test inputs Signal Builder block in the top-level model has one signal group that has a requirement link. Click the link under **Target (document name and location ID)** to open the requirements document associated with this signal group in the Test inputs block.

Table 3.3. Test inputs : Normal operation signal requirements



Chart

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

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

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

Table 10.1. Stateflow objects with requirements

Name	Requirements
warmup	Link#1 label: "During a calibratable warm up period the oxygen sensor correction shall be disabled." Target: fuelsys_req_docs\slvndemo_FuelSys_RequirementsSpecification.docx, at 'Simulink requirement item 3'
[speed==0 & press < zero_thresh]/...	Link#1 label: "Speed sensor failure detection" Target: fuelsys_req_docs\slvndemo_FuelSys_DesignDescription.docx, at 'Simulink requirement item 6'
Rich_Mixture	Link#1 label: "Enriched mixture usage" Target: fuelsys_req_docs\slvndemo_FuelSys_DesignDescription.docx, at 'Simulink requirement item 4'
Warmup	Link#1 label: "During a calibratable warm up period the oxygen sensor correction shall be disabled." Target: fuelsys_req_docs\slvndemo_FuelSys_RequirementsSpecification.docx, at 'Simulink requirement item 3'

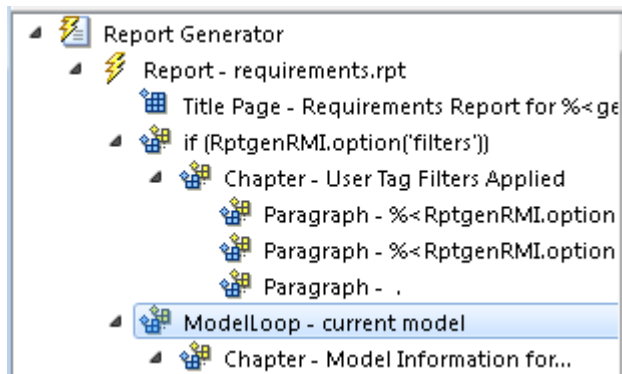
Report for Requirements in Model Blocks

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

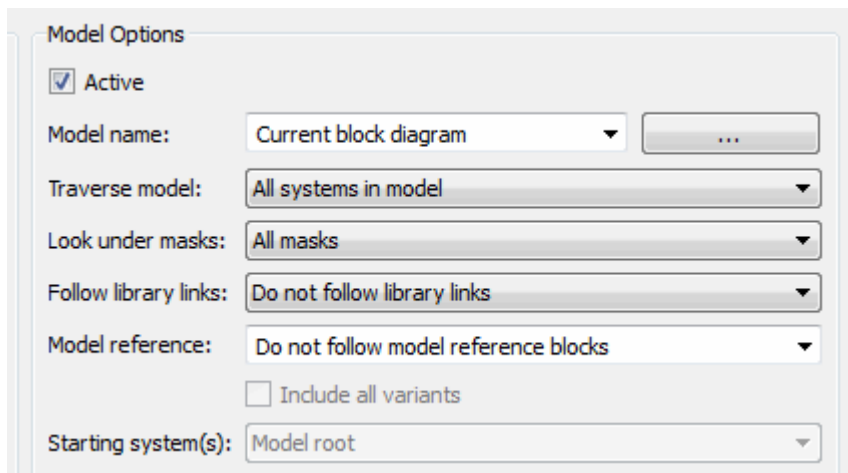
If you have a Simulink Report Generator license, before generating a requirements report, take the following steps:


- 1 Open the model for which you want to create a requirements report. This workflow uses the example model `slvndemo_fuelsys_officereq`.
- 2 To open the template for the default requirements report, at the MATLAB command prompt, enter:

```
setedit requirements
```
- 3 In the Simulink Report Generator software window, in the far-left pane, click the Model Loop component.



- 4 On the far-right pane, locate the **Model reference** field. If you cannot see the drop-down arrow for that field, expand the pane.



- 5 In the **Model reference** field drop-down list, select **Follow all model reference blocks**.
- 6 To generate a requirements report for the open model that includes information about referenced models, click the **Report** icon .

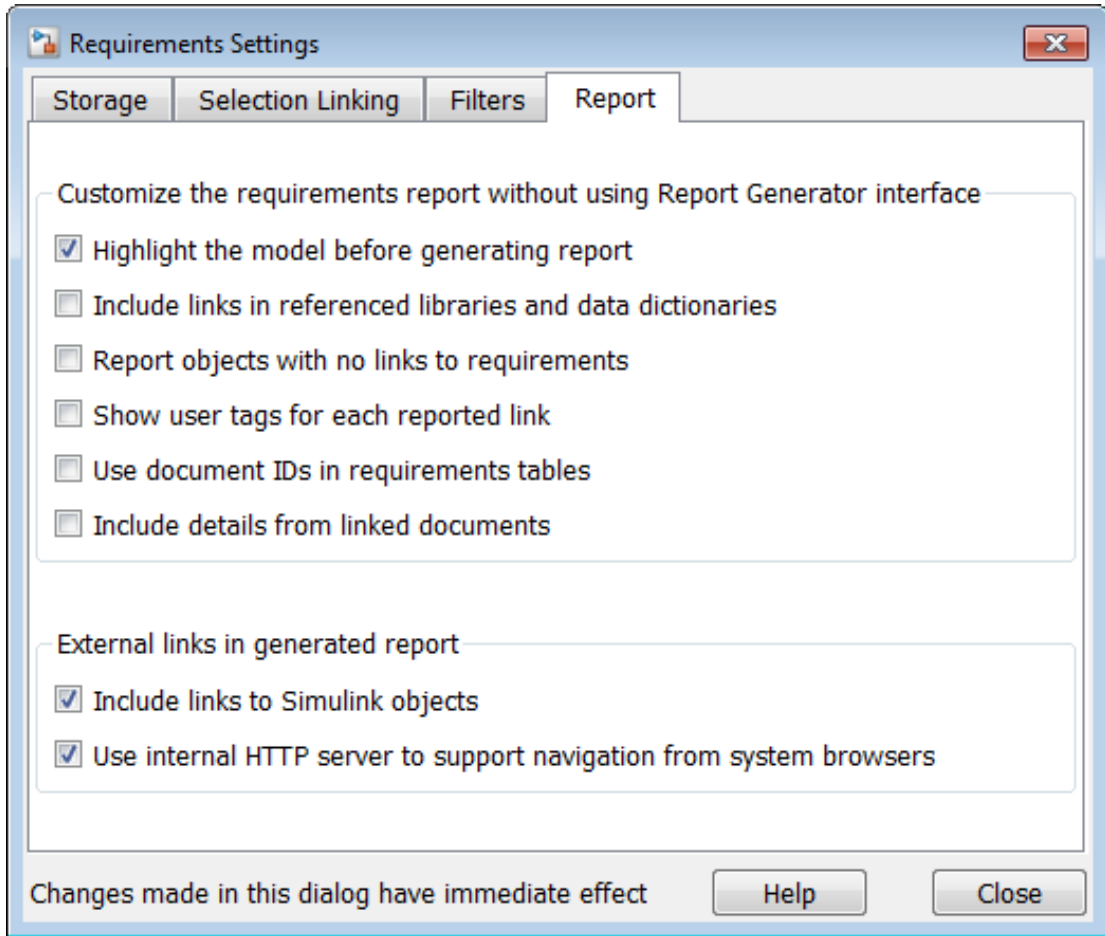
Customize Requirements Report

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

- “Customize Requirements Report Using the RMI Settings” on page 5-17
- “Customize Requirements Report Using Simulink Report Generator” on page 5-21

Customize Requirements Report Using the RMI Settings

There are several options for customizing a requirements report using the Requirements Settings dialog box.



On the **Report** tab, select options that specify the contents that you want in the report.

Requirements Settings Report Option	Description
Highlight the model before generating report	Enables highlighting of Simulink objects with requirements in the report graphics.
Include links in referenced libraries and data dictionaries	Includes requirements links in referenced libraries in the generated report.
Report objects with no links to requirements	Includes lists of model objects that have no requirements.

Requirements Settings Report Option	Description
Show user tags for each reported link	Lists the user tags, if any, for each reported link.
Use document IDs in requirements tables	Uses a document ID, if available, instead of a path name in the tables of the requirements report. This capability prevents long path names to requirements documents from cluttering the report tables.
Include details from linked documents	Includes additional content from linked requirements. The following requirements documents are supported: <ul style="list-style-type: none"> • Microsoft Word • Microsoft Excel • IBM Rational DOORS
Include links to Simulink objects	Includes links from the report to objects in Simulink.
Use internal HTTP server to support navigation from system browsers	Specifies use of internal MATLAB HTTP server for navigation from generated report to documents and model objects. By selecting this setting, this navigation is available from system browsers as long as the MATLAB internal HTTP server is active on your local host. To start the internal HTTP server, at the MATLAB command prompt, type <code>rmi('httpLink')</code> .

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

- 1 Open the `slvndemo_fuelsys_officereq` model:
`slvndemo_fuelsys_officereq`
- 2 In the model window, select **Analysis > Requirements Traceability > Settings**.
- 3 In the Requirements Settings dialog box, click the **Report** tab.
- 4 For this example, select **Highlight the model before generating report**.

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

- 5 To close the Requirements Settings dialog box, click **Close**.
- 6 Generate a requirements report by selecting **Analysis > Requirements Traceability > Generate Report**.

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

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

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

- 8 In the model window, select **Analysis > Requirements Traceability > Settings**.
- 9 On the **Report** tab, select:
 - **Show user tags for each reported link** — The report lists the user tags (if any) associated with each requirement.
 - **Include details from linked documents** — The report includes additional details for requirements in the following types of requirements documents.

Requirements Document Format	Includes in the Report
Microsoft Word	Full text of the paragraph or subsection of the requirement, including tables.
Microsoft Excel	If the target requirement is a group of cells, the report includes all those cells as a table. If the target requirement is one cell, the report includes that cell and all the cells in that row to the right of the target cell.
IBM Rational DOORS	By default, the report includes: <ul style="list-style-type: none"> • DOORS Object Heading • DOORS Object Text

Requirements Document Format	Includes in the Report
	<ul style="list-style-type: none"> All other attributes except Created Thru, attributes with empty string values, and system attributes that are false. <p>Use the <code>RptgenRMI.doorsAttribs</code> function to include or exclude specific attributes or groups of attributes.</p>

- 10 Close the Requirements Settings dialog box.
- 11 Generate a new requirements report by selecting **Analysis > Requirements Traceability > Generate Report**.
- 12 Compare this new report to the report that you renamed in step 7:
 - User tags associated with requirements links are included.
 - Details from the requirement content are included as specified in step 9.
- 13 When you are done reviewing the report, close the report and the model.

To see an example of including details in the requirements report, enter the following command at the MATLAB command prompt:

```
slvndemo_powerwindow_report
```

Customize Requirements Report Using Simulink Report Generator

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

At the MATLAB command prompt, enter the following command:

```
setedit requirements
```

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

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

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

Simulink Report Generator Component	Report Information
Missing Requirements Block Loop	Applies all child components to blocks that have no requirements
Missing Requirements System Loop	Applies all child components to systems that have no requirements
Requirements Block Loop	Applies all child components to blocks that have requirements
Requirements Documents Table	Inserts a table that lists requirements documents
Requirements Signal Loop	Applies all child components to signal groups with requirements
Requirements Summary Table	Inserts a property table that lists requirements information for blocks with associated requirements
Requirements System Loop	Applies all child components to systems with requirements
Requirements Table	Inserts a table that lists system and subsystem requirements
Data Dictionary Traceability Table	Inserts a table that links data dictionary information to requirements
MATLAB Code Traceability Table	Inserts a table that links MATLAB code to requirements
Simulink Test Suite Traceability Table	Inserts a table that links a Simulink test suite to requirements

To customize the requirements report, you can:

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

For more information, see the Simulink Report Generator documentation.

Generate Requirements Reports Using Simulink

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

- “System Design Description Report” on page 5-23
- “Design Requirements Report” on page 5-24

System Design Description Report

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

You can use the System Design Description report to:

- Review a system design without having the model open.
- Generate summary and detailed descriptions of the design.
- Assess compliance with design requirements.
- Archive the system design in a format independent of the modeling environment.
- Build a customized version of the report using the Simulink Report Generator software.

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

- 1 Open the model for which you want to create a report.
- 2 Select **File > Reports > System Design Description**.
- 3 In the Design Description dialog box, select **Requirements traceability**.
- 4 Select any other options that you want for this report.
- 5 Click **Generate**.

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

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

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

- Lists of model objects that have requirements with hyperlinks to display the objects
- Images of each subsystem, highlighting model objects with requirements

Design Requirements Report

In the Simulink Editor, the menu option **File > Reports > System Requirements** creates a requirements report, as described in “Create Default Requirements Report” on page 5-7. This menu option is equivalent to **Analysis > Requirements Traceability > Generate Report**.

To specify options for the report, select **Analysis > Requirements Traceability > Settings**. Before generating the report, on the **Report** tab, set the options that you want. For detailed information about these options, see “Customize Requirements Report” on page 5-17.

Filter Requirements with User Tags

In this section...

“User Tags and Requirements Filtering” on page 5-25

“Apply a User Tag to a Requirement” on page 5-25

“Filter, Highlight, and Report with User Tags” on page 5-27

“Apply User Tags During Selection-Based Linking” on page 5-29

“Configure Requirements Filtering” on page 5-31

User Tags and Requirements Filtering

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

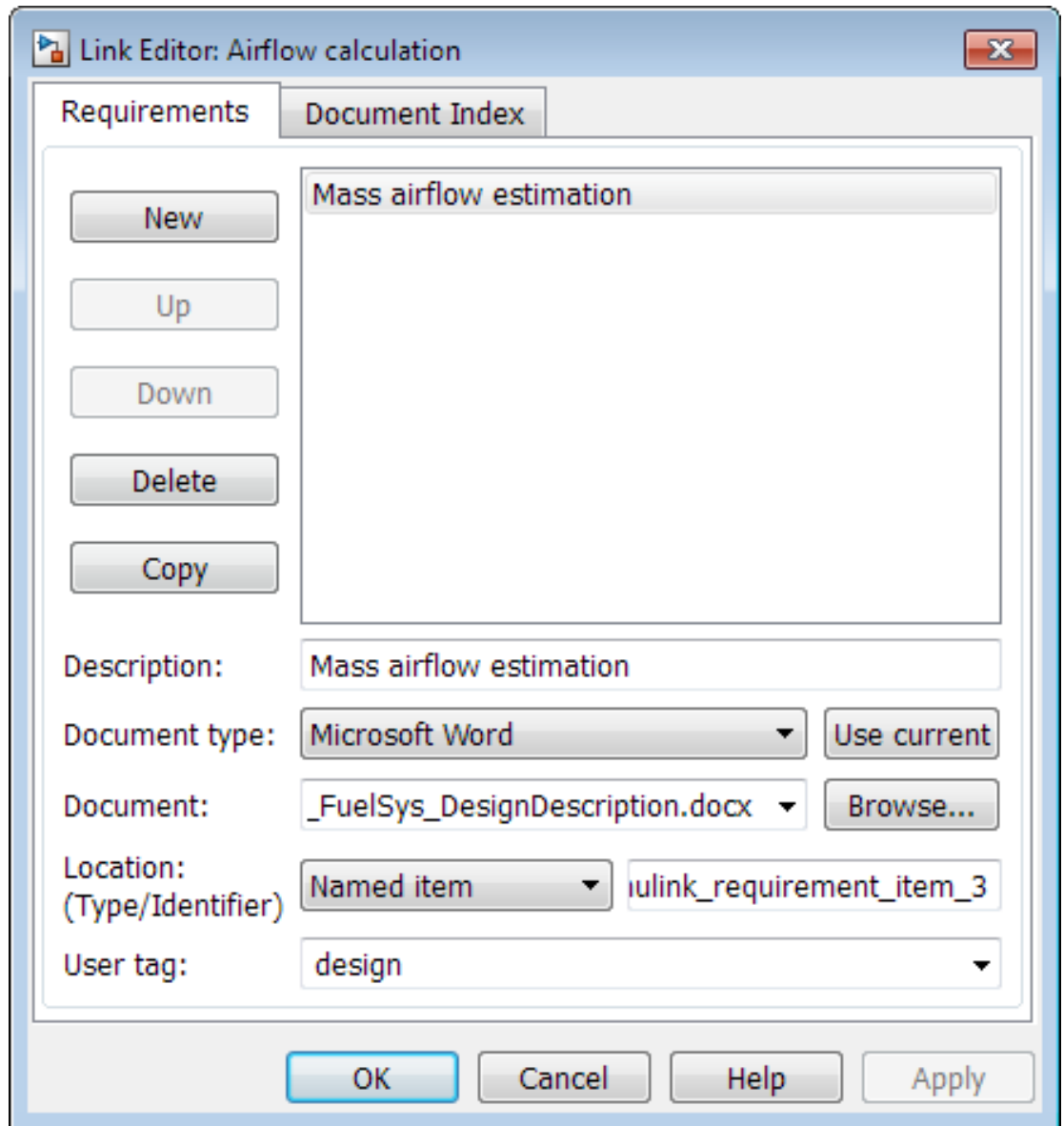
- Highlight or report only those requirements that have a specific user tag.
- Highlight or report only those requirements that have one of several user tags.
- Do not highlight and report requirements that have a specific user tag.

Apply a User Tag to a Requirement

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

- 1 Open the example model:
`slvndemo_fuelsys_officereq`
- 2 Open the fuel rate controller subsystem.
- 3 To open the requirements document, right-click the Airflow calculation subsystem and select **Requirements Traceability > Open Link Editor**.

The Requirements Traceability Link Editor opens with the details about the requirement that you created.



- 4 In the **User tag** field, enter one or more keywords, separated by commas, that the RMI can use to filter requirements. In this example, after **design**, enter a comma, followed by the user tag **test** to specify a second user tag for this requirement.

User tags:

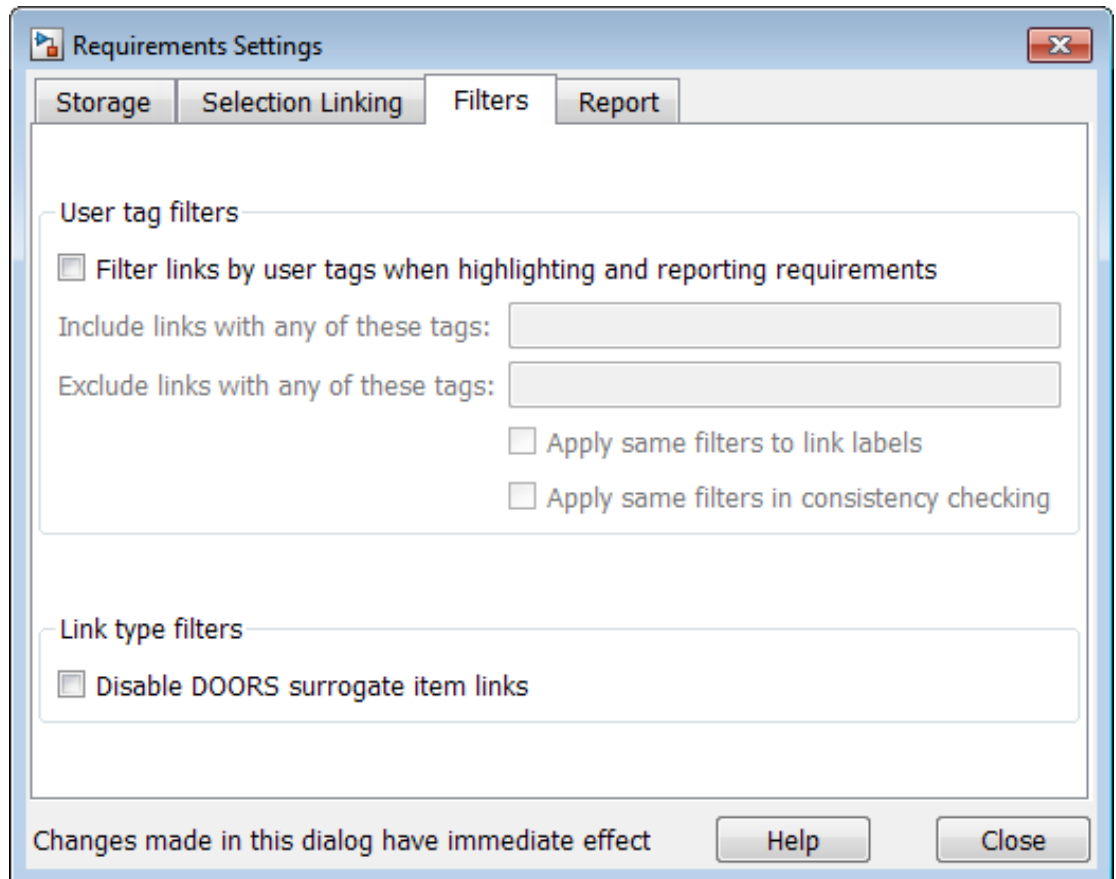
- Are not case sensitive.
- Can consist of multiple words. For example, if you enter **design requirement**, the entire phrase constitutes the user tag. Separate user tags with commas.

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

Filter, Highlight, and Report with User Tags

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

- 1 In the Simulink Editor, remove highlighting from the `slvndemo_fuelsys_officereq` model by selecting **Analysis > Requirements Traceability > Unhighlight model**.
- 2 Select **Analysis > Requirements Traceability > Settings**.
- 3 In the Requirements Settings dialog box, click the **Filters** tab.



- 4 To enable filtering with user tags, click the **Filter links by user tags when highlighting and reporting requirements** option.
- 5 To include only those requirements that have the user tag, `test`, enter `test` in the **Include links with any of these tags** field.
- 6 Click **Close**.
- 7 In the Simulink Editor, select **Analysis > Requirements Traceability > Highlight model**.

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

- 8 Reopen the Requirements Settings dialog box to the **Filters** tab.
- 9 In the **Include links with any of these tags** field, delete `test`. In the **Exclude links with any of these tags** field, add `test`.

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

- 10 In the Simulink Editor, select **Analysis > Requirements Traceability > Generate Report**.

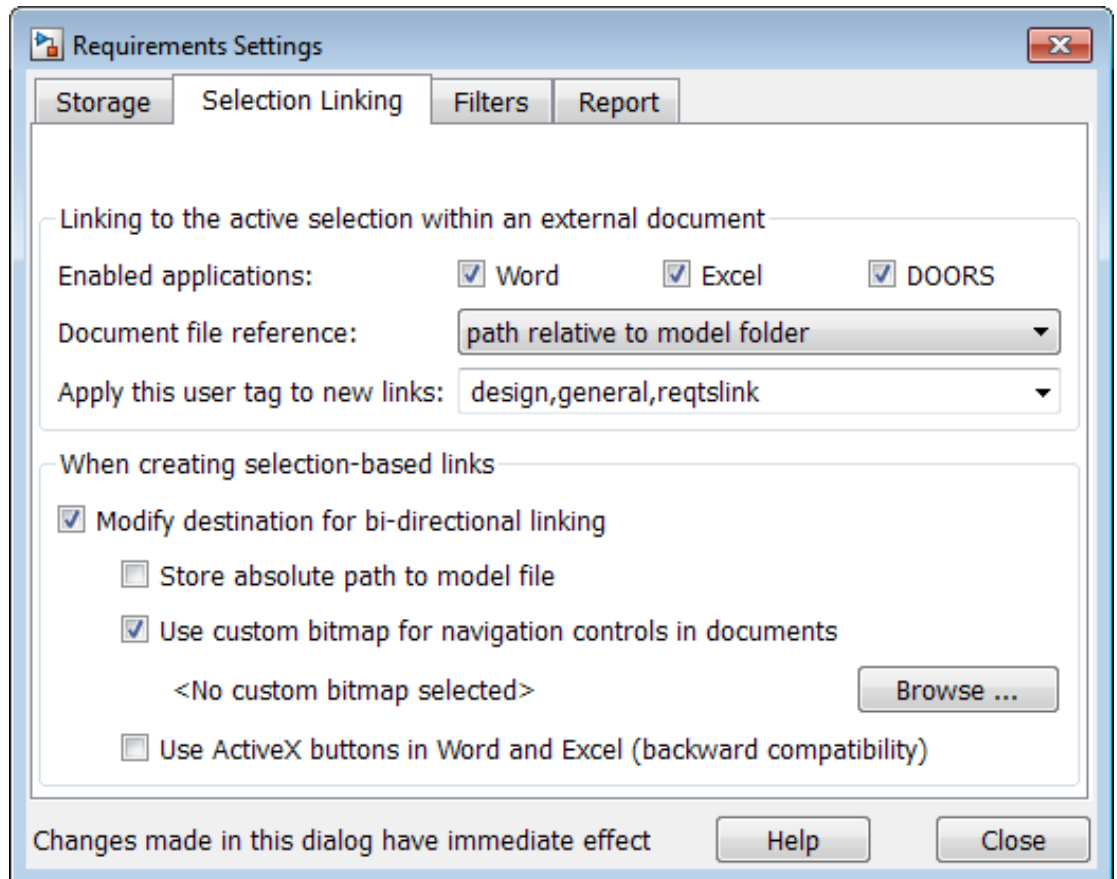
The report does not include information about objects whose requirements have the `test` user tag.

Apply User Tags During Selection-Based Linking

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

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

- 1 In the Simulink Editor, select **Analysis > Requirements Traceability > Settings**.
- 2 Select the **Selection Linking** tab.

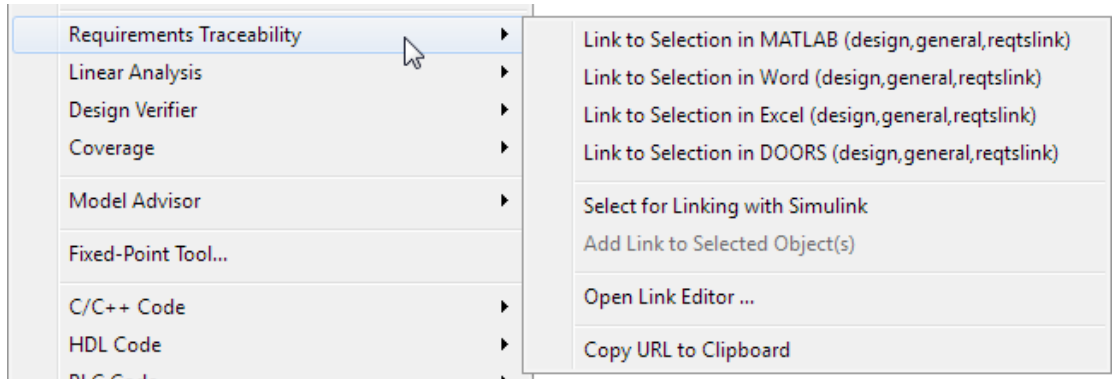


- 3 In the **Apply this user tag to new links** field, enter one or more user tags, separated by commas.

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

- 4 Click **Close** to close the Requirements Settings dialog box.
- 5 In a requirements document, select the specific requirement text.
- 6 Right-click a model object and select **Requirements Traceability**.

The selection-based linking options specify which user tags the RMI applies to the link that you create. In the following example, you can apply the user tags **design**, **general**, and **reqtslink** to the link that you create to your selected text.



Configure Requirements Filtering

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

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

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

Create Requirements Traceability Report for Simulink Project

Abstract

Generate a report of requirements traceability links for files in a Simulink Project.

To create a report for requirements traceability data in a Simulink Project:

- 1 Open your Simulink Project.
- 2 At the MATLAB command prompt, enter the following:

```
rmi('projectreport')
```

The MATLAB Web browser opens, showing the traceability report for the Simulink Project.

This top-level HTML report contains a separate section for Simulink model files, MATLAB code files, and other files included in the project. For each individual file with one or more associated requirements links, a separate HTML report, or sub-report, shows the requirements traceability data for that file. The top-level report contains links to each sub-report.

If you have a MATLAB file with requirements traceability links that is not part of a Simulink Project, you can create a separate report for the MATLAB file using the `rmi('report', matlabfilepath)` command. For more information, see `rmi`.

View Requirements Details for a Selected Block

In this section...

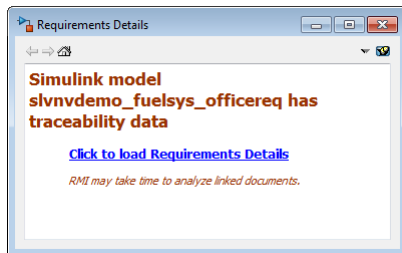
“Requirements Details Workflow” on page 5-34

“Requirements Details Limitations” on page 5-35

Requirements Details Workflow

When you highlight model objects with requirements, you can view the requirements details for a selected block. Follow this workflow:

- 1 From the menu bar, select **Analysis > Requirements Traceability > Highlight Model**.
- 2 In your model, highlights indicate model objects with requirements links.



- 3 In the requirements details window, click the link to load the requirements details.

If your Simulink model has links to Microsoft Word, Microsoft Excel, or IBM Rational DOORS documents, the requirements management interface loads requirements context from the documents and additional link labels stored by Simulink.

- 4 Select highlighted model objects to display associated RMI links in the requirements details window.
- 5 Close the requirements details window to remove highlights from the Simulink model.

For more information see “Navigate to Requirements from Model” on page 5-5.

Requirements Details Limitations

Security restrictions in Microsoft Office can interrupt the requirements details loading process. Requirement details loaded from read-only Microsoft Office documents, documents stored on network drives, and documents with Microsoft Office Trust Center ActiveX control restrictions may not work with RMI. Consider enabling ActiveX controls without prompting and using requirements stored in a writable location on the MATLAB path.

Before loading requirements details for IBM Rational DOORS links, you must be logged in to the IBM Rational DOORS Client.

Requirements Links Maintenance

- “Validation of Requirements Links” on page 6-2
- “Validate Requirements Links in a Model” on page 6-4
- “Validate Requirements Links in a Requirements Document” on page 6-10
- “Document Path Storage” on page 6-14
- “Delete Requirements Links from Simulink Objects” on page 6-16
- “Requirements Links for Library Blocks and Reference Blocks” on page 6-18

Validation of Requirements Links

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

In this section...
“When to Check Links in a Requirements Document” on page 6-2
“How the rmi Function Checks a Requirements Document” on page 6-3

When to Check Links in a Requirements Document

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

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

- Microsoft Word
- Microsoft Excel
- IBM Rational DOORS

The rmi function only checks requirements documents that contain navigation controls; to check links in your Simulink model, see “Validate Requirements Links in a Model” on page 6-4.



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

- “Insert Navigation Objects in Microsoft Office Requirements Documents” on page 8-16

- “Insert Navigation Objects into DOORS Requirements” on page 8-7
-

How the rmi Function Checks a Requirements Document

rmi performs the following actions:

- Locates all links to Simulink objects in the specified requirements document.
- Checks each link to verify that the target object is present in a Simulink model. If the target object is present, rmi checks that the link label matches the target object.
- Modifies the navigation controls in the requirements document to identify any detected problems. This allows you to see invalid links at a glance:
 - Valid link: 
 - Invalid link: 

Validate Requirements Links in a Model

In this section...

“Check Requirements Links with the Model Advisor” on page 6-4

“Fix Invalid Requirements Links Detected by the Model Advisor” on page 6-6

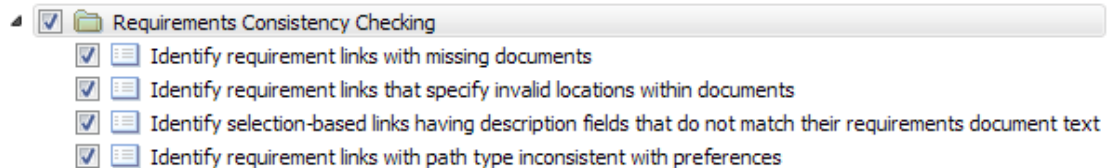
Check Requirements Links with the Model Advisor

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

- 1 Open the example model:

```
slvndemo_fuel_sys_officereq
```
- 2 Open the Model Advisor to run a consistency check by selecting **Analysis > Requirements Traceability > Check Consistency**.

In the **Requirements Consistency Checking** category, all the checks are selected. For this tutorial, keep all the checks selected.



These checks identify the following problems with your model requirements.

Consistency Check	Problem Identified
Identify requirement links with missing documents	The Model Advisor cannot find the requirements document. This might indicate a problem with the path to the requirements document.
Identify requirement links that specify invalid locations within documents	The Model Advisor cannot find the designated location for the requirement. This check is implemented for: <ul style="list-style-type: none"> • Microsoft Word documents

Consistency Check	Problem Identified
	<ul style="list-style-type: none"> • Microsoft Excel documents • IBM Rational DOORS documents • Simulink objects
<p>Identify selection-based links having description fields that do not match their requirements document text</p>	<p>The Description field for the link does not match the requirements document text. When you create selection-based links, the Requirements Management Interface (RMI) saves the selected text in the link Description field. This check is implemented for:</p> <ul style="list-style-type: none"> • Microsoft Word documents • Microsoft Excel documents • IBM Rational DOORS documents • Simulink objects
<p>Identify requirement links with path type inconsistent with preferences</p>	<p>The path to the requirements document does not match the Document file reference field in the Requirements Settings dialog box Selection Linking tab. This might indicate a problem with the path to the requirements document.</p> <p>On Linux[®] systems, this check is named Identify requirement links with absolute path type. The check reports a warning for each requirements links that uses an absolute path.</p> <hr/> <p>Note: For information about how the RMI resolves the path to the requirements document, see “Document Path Storage” on page 6-14.</p>

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

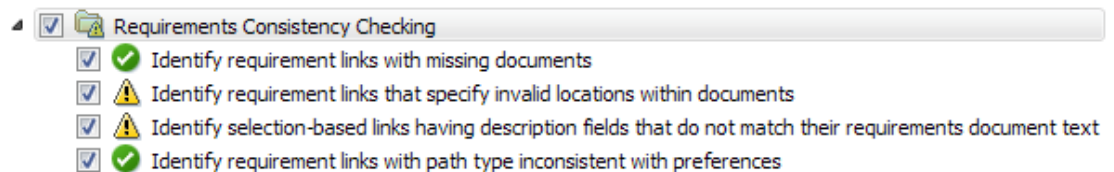
- If your model has links to Microsoft Word or Microsoft Excel documents, the consistency check requires that you close all instances of those applications. If you have one of these applications open, it displays a warning and does not

continue the checks. The consistency checks must verify up-to-date stored copies of the requirements documents.

- If your model has links to DOORS requirements, you must be logged in to the DOORS software. Your DOORS database must include the module that contains the target requirements.
- 3 For this tutorial, make sure that you close both Microsoft Word and Microsoft Excel.
 - 4 Click **Run Selected Checks**.

After the check is complete:

- The green circles with the check mark indicate that two checks passed.
- The yellow triangles with the exclamation point indicate that two checks generated warnings.



The right-hand pane shows that two checks passed and two checks had warnings. The Report box includes a link to the HTML report.

Keep the Model Advisor open. The next section describes how to interpret and fix the inconsistent links.

Note: To step through an example that uses the Model Advisor to check requirements links in an IBM Rational DOORS database, run the Managing Requirements for Fault-Tolerant Fuel Control System (IBM Rational DOORS) example in the MATLAB command prompt.

Fix Invalid Requirements Links Detected by the Model Advisor

In “Check Requirements Links with the Model Advisor” on page 6-4, three requirements consistency checks generate warnings in the `slvnvdemo_fuelsys_officereq` model.

Resolve Warning: Identify requirement links that specify invalid locations within documents

To fix the warning about attempting to link to an invalid location in a requirements document:

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

Inconsistencies:

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

Block	Requirements
slvnvdemo_fuelsys_officereq/fuel_rate_controller/Sensor_correction_and_Fault_Redundancy/Terminator1	This section will be deleted

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

- 2 Get more information about this link:
 - a To navigate to the Terminator1 block, under **Block**, click the hyperlink.
 - b To open the “Requirements Traceability Link Editor” on page 3-17 for this link, under **Requirements**, click the hyperlink.
- 3 To fix the problem from the Requirements Traceability Link Editor, do one of the following:
 - In the **Location** field, specify a valid location in the requirements document.
 - Delete the requirements link by selecting the link and clicking **Delete**.
- 4 In the Model Advisor, select the **Requirements Consistency Checking** category of checks.

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

Resolve Warning: Identify selection-based links having description fields that do not match their requirements document text

To fix the warnings about the **Description** field not matching the requirements document text:

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

Unable to check:

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

Inconsistencies:

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

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

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

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

- The **Description** field in the Test inputs Signal Builder block link is **Normal mode of operation**. The requirement text is **The simulation is run with a throttle input that ramps from 10 to 20 degrees over a period of two seconds, then back to 10 degrees over the next two seconds. This cycle repeats continuously while the engine is held at a constant speed.**
 - The **Description** field in the MAP Estimate block link is **Manifold pressure failure**. The requirement text in `slvndemo_FuelSys_DesignDescription.docx` is **Manifold pressure failure mode**.
- 2** Get more information about this link:
 - a** To navigate to a block, under **Block**, click the hyperlink.
 - b** To open the “Requirements Traceability Link Editor” on page 3-17 for this link, under **Current Description**, click the hyperlink.
 - 3** Fix this problem in one of two ways:
 - In the Model Advisor, click **Update**. This action automatically updates the **Description** field for that link so that it matches the requirement.
 - In the Link Editor, manually edit the link from the block so that the **Description** field matches the selected requirements text.
 - 4** In the Model Advisor, select the **Requirements Consistency Checking** category of checks.
 - 5** Click **Run Selected Checks** again, and verify that the warning no longer occurs.

Validate Requirements Links in a Requirements Document

In this section...
“Check Links in a Requirements Document” on page 6-10
“When Multiple Objects Have Links to the Same Requirement” on page 6-11
“Fix Invalid Links in a Requirements Document” on page 6-12

Check Links in a Requirements Document

To check the links in a requirements document:

- 1 At the MATLAB command prompt, enter

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

`docName` is a character vector that represents one of the following:

- Module ID for a DOORS requirements document
- Full path name for a Microsoft Word requirements document
- Full path name for a Microsoft Excel requirements document

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

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

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

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

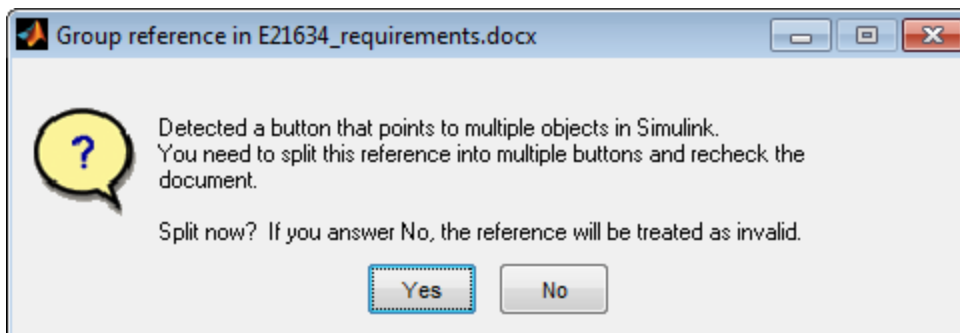
- 3 When there are invalid links in your requirements document, you have the following options:

If you want to...	Do the following...
Fix the invalid links	Follow the instructions in “Fix Invalid Links in a Requirements Document” on page 6-12.
Keep the changes to the navigation controls without fixing the invalid links	Save the requirements document.
Ignore the invalid links	Close the requirements document without saving it.

When Multiple Objects Have Links to the Same Requirement

When you link multiple objects to the same requirement, as described in “Link Multiple Model Objects to a Requirements Document” on page 3-25, only one navigation object is inserted into the requirements document. When you double-click that navigation object, all of the linked model objects are highlighted.

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



You have two options:

- If you click **Yes**, or you close this dialog box, the RMI creates additional navigation objects, one for each model object that links to that requirement. The document check continues, but the RMI does not recheck that navigation; the report only shows one

link for that requirement. To rerun the check so that all requirements are checked, at the top of the report, click **Refresh**.

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

Fix Invalid Links in a Requirements Document

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

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

References with Unresolved Models - 1 unique problem in 2 links

Document content	Target model
Transmission Requirements	sf_car_doors.mdl
Engine Torque Requirements	

To fix invalid links:

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

The requirements document opens with the requirement text highlighted.

- 2 In the requirements document, depending on the document format, take these steps:
 - In DOORS:
 - a Select the navigation control for an invalid link.
 - b Select **MATLAB > Select item**.
 - In Microsoft Word, double-click the navigation control.

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

- 3 Click one of the following options.

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

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

Document Path Storage

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

- Absolute path
- Path relative to current folder
- Path relative to model folder
- Filename only (on MATLAB path)

You can also manually enter an absolute or relative path for the document location. A relative path can be a partial path or no path at all, but you must specify the file name of the requirements document. If you use a relative path, the document is not constrained to a single location in the file system. With a relative path, the RMI resolves the exact location of the requirements document in this order:

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

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

Relative (Partial) Path Example

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

Relative (No) Path Example

Current MATLAB folder	C:\work\scratch
-----------------------	-----------------

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

Absolute Path Example

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

Delete Requirements Links from Simulink Objects

In this section...
“Delete a Single Link from a Simulink Object” on page 6-16
“Delete All Links from a Simulink Object” on page 6-16
“Delete All Links from Multiple Simulink Objects” on page 6-17

Delete a Single Link from a Simulink Object

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

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

- 1 Right-click a model object and select **Requirements Traceability > Open Link Editor**.
- 2 In the top-most pane of the Link Editor, select the link that you want to delete.
- 3 Click **Delete**.
- 4 Click **Apply** or **OK** to complete the deletion.

Delete All Links from a Simulink Object

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

- 1 Right-click the model object and select **Requirements Traceability > Delete All Links**
- 2 Click **OK** to confirm the deletion.

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

Delete All Links from Multiple Simulink Objects

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

- 1 Select the model objects whose requirements links you want to delete.
- 2 Right-click one of the objects and select **Requirements Traceability > Delete All**.
- 3 Click **OK** to confirm the deletion.

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

Requirements Links for Library Blocks and Reference Blocks

In this section...
“Introduction to Library Blocks and Reference Blocks” on page 6-18
“Library Blocks and Requirements” on page 6-18
“Copy Library Blocks with Requirements” on page 6-19
“Manage Requirements on Reference Blocks” on page 6-19
“Manage Requirements Inside Reference Blocks” on page 6-20
“Links from Requirements to Library Blocks” on page 6-23

Introduction to Library Blocks and Reference Blocks

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

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

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

Note: For more information about reference blocks and library links, see “Libraries” in the Simulink documentation.

Library Blocks and Requirements

Library blocks themselves can have links to requirements. In addition, if a library block is a subsystem or atomic subchart, the objects inside the library blocks can have library links. You use the Requirements Management Interface (RMI) to create and manage requirements links in libraries and in models.

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

Copy Library Blocks with Requirements

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

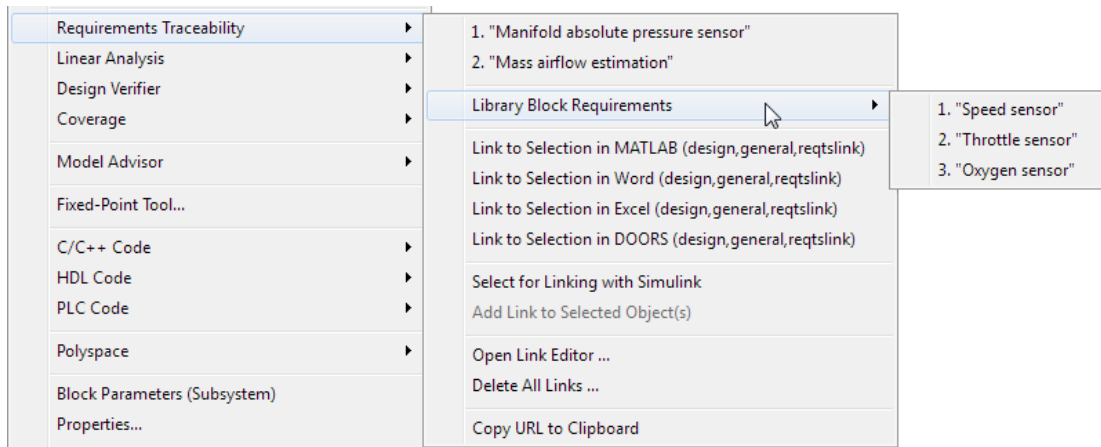
You cannot add, modify, or delete requirements links on the library block from the context of the reference block. If you disable the link from the reference block to the library block, you can modify requirements on objects that are inside library blocks just as you can for other block attributes when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

Manage Requirements on Reference Blocks

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

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

- Locally created requirements links — Can be modified or deleted without changing the library block:
 - **Manifold absolute pressure sensor**
 - **Mass airflow estimation**
- Requirements links on the library block — Cannot be modified or deleted from the context of the reference block:
 - **Speed sensor**
 - **Throttle sensor**
 - **Oxygen sensor**



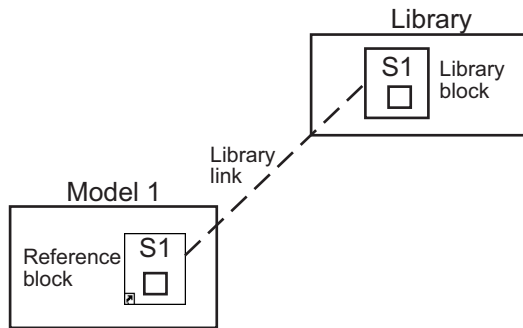
Manage Requirements Inside Reference Blocks

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

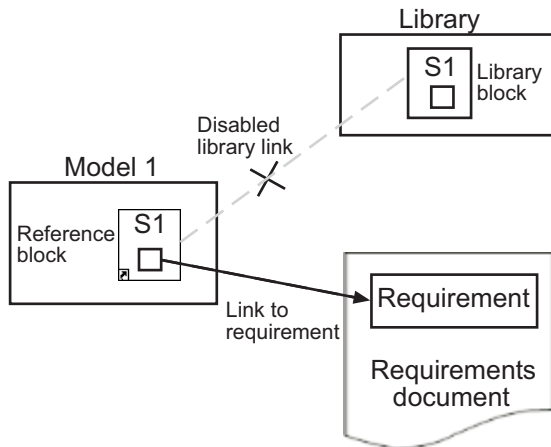
After you make changes to requirements links on objects inside a reference block, you can resolve the link so that those changes are pushed to the library block. The next time you create an instance of that library block, the changes you made are copied to the new instance of the library block.

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

- 1 Within a library you have a subsystem S1. Drag S1 to a model, creating a new subsystem. This subsystem is the reference block.



- 2 Disable the library link between the reference block and the library block. Keep the library loaded while you disable the link to maintain RMI data. To disable the link, select the reference block and select **Diagram > Library Link > Disable Link**.
- 3 Create a link from the object inside the reference block to the requirements document.

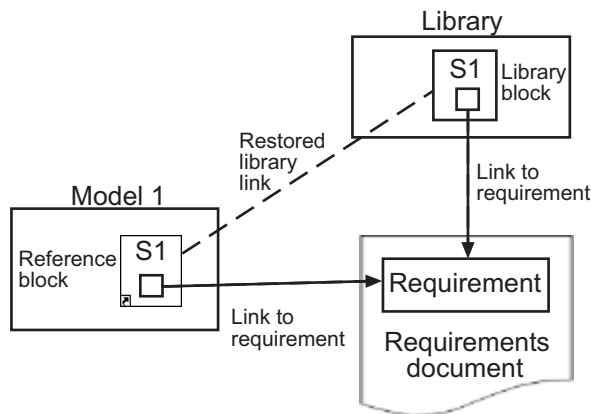


Note: When linking to a requirement from inside a reference block, you can create links only in one direction: from the model to the requirements document. The RMI does not support inserting navigation objects into requirements documents for objects inside reference blocks.

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

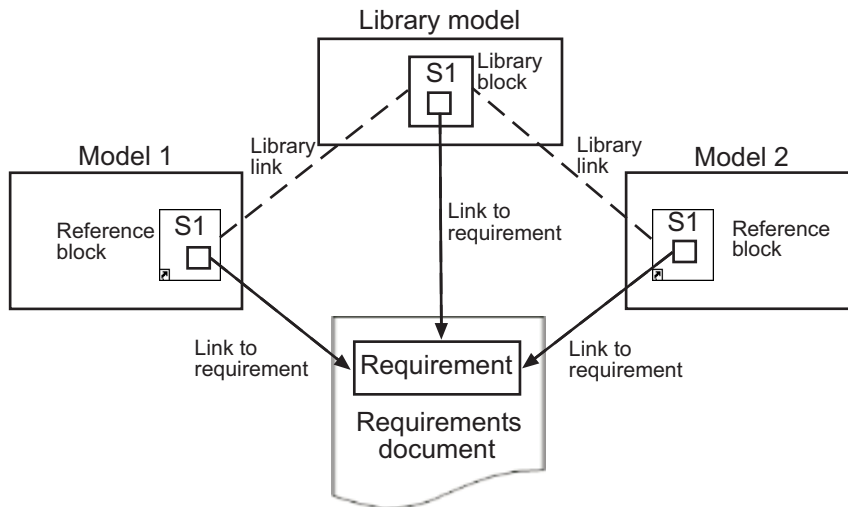
- a Select the reference block.
- b Select **Diagram > Library Link > Resolve Link**.
- c In the **Action** column, click **Push**.
- d Click **OK** to resolve the link to the library block and push the newly added requirement to the object inside the library block.

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



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

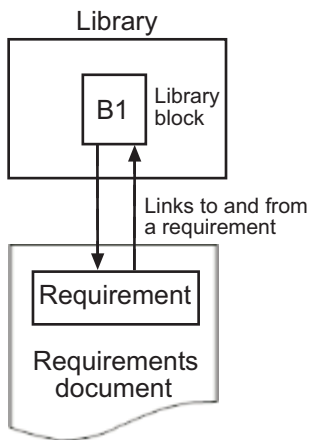
- 5 If you reuse library block S1, which now has an object with a requirement link, in another model, the new subsystem contains an object that links to that requirement.



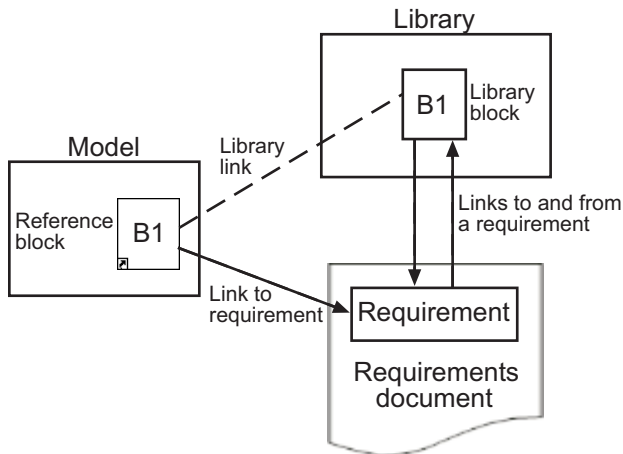
Links from Requirements to Library Blocks

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

For example, consider the situation where you have established linking between a library block (B1 in the following graphic) and a requirement in both directions.



When you use library block B1 in a model, you can navigate from the reference block to the requirement. However, the link from the requirement still points only to library block B1, not to the reference block.



As discussed in the previous section, you can create requirements links on objects inside instances of library block after disabling library links. However, the RMI prohibits you from creating a link from the requirements document to such an object because that link would become invalid when you restored the library link.

IBM Rational DOORS Surrogate Module Synchronization

- “Synchronization with DOORS Surrogate Modules” on page 7-2
- “Advantages of Synchronizing Your Model with a Surrogate Module” on page 7-4
- “Synchronize a Simulink Model to Create a Surrogate Module” on page 7-5
- “Create Links Between Surrogate Module and Formal Module in a DOORS Database” on page 7-7
- “Customize DOORS Synchronization” on page 7-8
- “Resynchronize DOORS Surrogate Module to Reflect Model Changes” on page 7-15
- “Navigate with the Surrogate Module” on page 7-17

Synchronization with DOORS Surrogate Modules

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

When you synchronize a model for the first time, the DOORS software creates a surrogate module. The surrogate module contains a representation of the model, depending on your synchronization settings. (To learn how to customize the links and level of detail in the synchronization, see “Customize DOORS Synchronization” on page 7-8.)

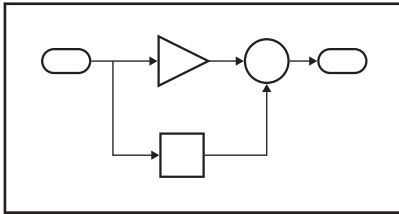
If you create or remove model objects or links, keep your surrogate module up to date by resynchronizing. The updated surrogate module reflects any changes in the requirements links since the previous synchronization.

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

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

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

Objects in a Simulink Model



DOORS Surrogate Module

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

DOORS Formal Module(s) with Requirements

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

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

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

Advantages of Synchronizing Your Model with a Surrogate Module

Abstract

Discover the benefits of synchronizing your Simulink model with a surrogate module.

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

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

Synchronize a Simulink Model to Create a Surrogate Module

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

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

Note: Before you begin, make sure you know how to create links from a Simulink model object to a requirement in a DOORS database. For a tutorial on creating links to DOORS requirements, see “Link to Requirements in IBM Rational DOORS Databases” on page 3-36.

- 1 To create a surrogate module, start the DOORS software and open a project. If the DOORS software is not already running, start the DOORS software and open a project.
- 2 Open the `sf_car` model.
- 3 Rename the model to `sf_car_doors`, and save the model in a writable folder.
- 4 Create links to a DOORS formal module from two objects in `sf_car_doors`:
 - The transmission subsystem
 - The engine torque block inside the Engine subsystem
- 5 Save the changes to the model.
- 6 In the Simulink Editor, select **Analysis > Requirements Traceability > Synchronize with DOORS**.

The DOORS synchronization settings dialog box opens.

- 7 For this tutorial, accept the default synchronization options.

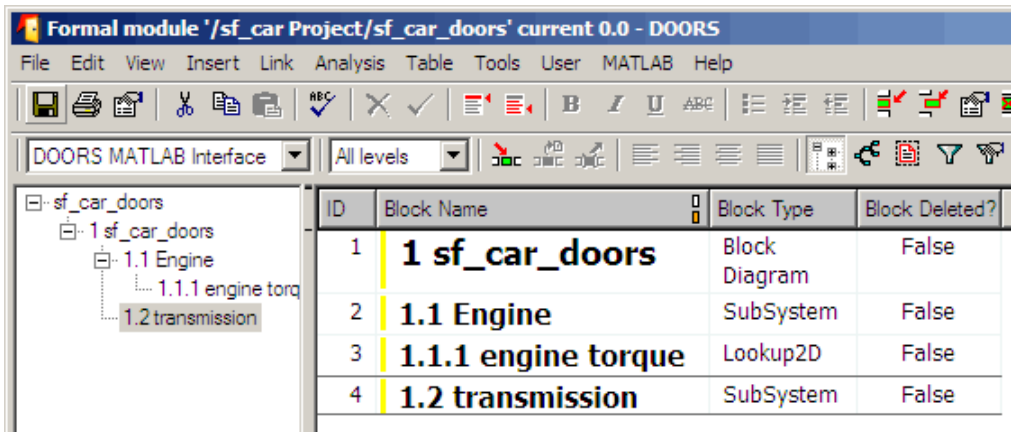
The default option under **Extra mapping additionally to objects with links**, **None**, creates objects in the surrogate module only for the model and any model objects with links to DOORS requirements.

Note: For more information about the synchronization options, see “Customize DOORS Synchronization” on page 7-8.

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

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

- A top-level object for the model (`sf_car_doors`)
- Objects that represent model objects with links to DOORS requirements (transmission, engine torque), and their parent objects (Engine).



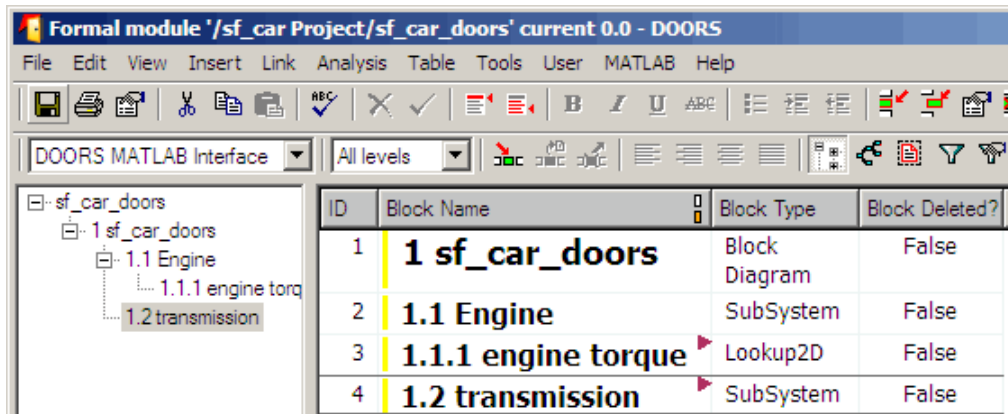
- 9 Save the surrogate module and the model.

Create Links Between Surrogate Module and Formal Module in a DOORS Database

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

- 1 Open the `sf_car_doors` model.
- 2 In the Simulink Editor, select **Analysis > Requirements Traceability > Synchronize with DOORS**.
- 3 In the DOORS synchronization settings dialog box, select two options:
 - **Update links during synchronization**
 - **from Simulink to DOORS**.
- 4 Click **Synchronize**.

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



The screenshot shows the DOORS MATLAB Interface window titled 'Formal module '/sf_car Project/sf_car_doors' current 0.0 - DOORS'. The interface includes a menu bar (File, Edit, View, Insert, Link, Analysis, Table, Tools, User, MATLAB, Help) and a toolbar. Below the toolbar, there is a 'DOORS MATLAB Interface' dropdown and an 'All levels' dropdown. The main area is divided into two panes. The left pane shows a hierarchical tree view of the surrogate module structure: 'sf_car_doors' containing '1 sf_car_doors', which contains '1.1 Engine', which contains '1.1.1 engine torque', and '1.2 transmission'. The right pane is a table with the following data:

ID	Block Name	Block Type	Block Deleted?
1	1 sf_car_doors	Block Diagram	False
2	1.1 Engine	SubSystem	False
3	1.1.1 engine torque	Lookup2D	False
4	1.2 transmission	SubSystem	False

Customize DOORS Synchronization

In this section...

“DOORS Synchronization Settings” on page 7-8

“Resynchronize a Model with a Different Surrogate Module” on page 7-10

“Customize the Level of Detail in Synchronization” on page 7-11

“Resynchronize to Include All Simulink Objects” on page 7-12

DOORS Synchronization Settings

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

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

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

DOORS Settings Option	Description
DOORS surrogate module path and name	Specifies a unique DOORS path to a new or an existing surrogate module. For information about how the RMI resolves the path to the requirements document, see “Document Path Storage” on page 6-14.
Extra mapping additionally to objects with links	Determines the completeness of the Simulink model representation in the DOORS surrogate module. None specifies synchronizing only those Simulink objects that have linked requirements, and their parent objects. For more information about these synchronization options, see “Customize the Level of Detail in Synchronization” on page 7-11.
Update links during synchronization	Specifies updating any unmatched links the RMI encounters during synchronization, as designated in the Copy unmatched links and Delete unmatched links options.

DOORS Settings Option	Description
Copy unmatched links	<p>During synchronization, selecting the following options has the following results:</p> <ul style="list-style-type: none"> • from Simulink to DOORS: For links between the model and the formal module, the RMI creates matching links between the DOORS surrogate and formal modules. • from DOORS to Simulink: For links between the DOORS surrogate and formal modules, the RMI creates matching links between the model and the DOORS modules.
Delete unmatched links	<p>During synchronization, selecting the following options has the following results:</p> <ul style="list-style-type: none"> • Remove unmatched in DOORS: For links between the formal and surrogate modules, when there is not a corresponding link between the model and the DOORS modules, the RMI deletes the link in DOORS. <p>This option is available only if you select the from Simulink to DOORS option.</p> <ul style="list-style-type: none"> • Remove unmatched in Simulink: For links between the model and the DOORS modules, when there is not a corresponding link between the formal and surrogate modules, the RMI deletes the link from the model. <p>This option is available only if you select the from DOORS to Simulink option.</p>
Save DOORS surrogate module	<p>After the synchronization, saves changes to the surrogate module and updates the version of the surrogate module in the DOORS database.</p>

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

Resynchronize a Model with a Different Surrogate Module

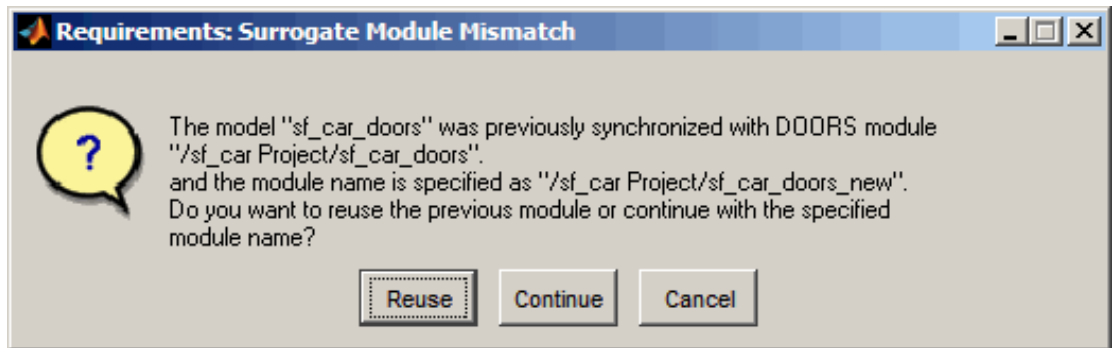
You can synchronize the same Simulink model with a new DOORS surrogate module. For example, you might want the surrogate module to contain only objects that have requirements to DOORS, rather than all objects in the model. In this case, you can change the synchronization options to reduce the level of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

- 1 In the DOORS synchronization settings dialog box, change the **DOORS surrogate module path and name** to the path and name of the new surrogate module in the DOORS database.
- 2 Specify a module with either a relative path (starting with ./) or a full path (starting with /).

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

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

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



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

Customize the Level of Detail in Synchronization

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

In “Synchronize a Simulink Model to Create a Surrogate Module” on page 7-5, you synchronized the model with the **Extra mapping additionally to objects with links** option set to **None**. As a result, the surrogate module contains only Simulink objects that have requirement links, and their parent objects. Additional synchronization options, described in this section, can increase the level of surrogate detail. Increasing the level of surrogate detail can slow down synchronization.

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

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

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

Resynchronize to Include All Simulink Objects

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

- 1 Open the `sf_car_doors` model that you synchronized in “Synchronize a Simulink Model to Create a Surrogate Module” on page 7-5 and again in “Create Links Between Surrogate Module and Formal Module in a DOORS Database” on page 7-7.
- 2 In the Simulink Editor, select **Analysis > Requirements Traceability > Synchronize with DOORS**.

The DOORS synchronization settings dialog box opens.

- 3 Resynchronize with the same surrogate module, making sure that the **DOORS surrogate module path and name** specifies the surrogate module path and name that you used in “Synchronize a Simulink Model to Create a Surrogate Module” on page 7-5.

For information about how the RMI resolves the path to the requirements document, see “Document Path Storage” on page 6-14.

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

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

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

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

Detailed Information About The Surrogate Module You Created

Notice the following information about the surrogate module that you created in “Resynchronize to Include All Simulink Objects” on page 7-12:

- The name of the surrogate module is `sf_car_doors`, as you specified in the DOORS synchronization settings dialog box.
- DOORS object headers are the names of the corresponding Simulink objects.
- The **Block Type** column identifies each object as a particular block type or a subsystem.
- If you delete a previously synchronized object from your Simulink model and then resynchronize, the **Block Deleted** column reads **true**. Otherwise, it reads **false**.

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

- Each Simulink object has a unique ID in the surrogate module. For example, the ID for the surrogate module object associated with the Mux block in the preceding figure is 11.
- Before the complete synchronization, the surrogate module contained the transmission subsystem, with an ID of 3. After the complete synchronization, the transmission object retains its ID (3), but is listed farther down in the surrogate module. This order reflects the model hierarchy. The transmission object in the surrogate module retains the red arrow that indicates that it links to a DOORS formal module object.

Resynchronize DOORS Surrogate Module to Reflect Model Changes

Abstract

Resynchronize your model to reflect changes to the Simulink model.

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

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

- 1 In the `sf_car_doors` model, make a copy of the vehicle mph (yellow) & throttle % Scope block and paste it into the model. The name of the new Scope block is vehicle mph (yellow) & throttle %1.
- 2 Select **Analysis > Requirements Traceability > Synchronize with DOORS**.
- 3 In the DOORS synchronization settings dialog box, leave the **Extra mapping additionally to objects with links** option set to **Complete - All blocks, subsystems, states, and transitions**. Click **Synchronize**.

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

89	1.10.6 Ti	Outport	False
90	1.10.7 Tout	Outport	False
91	1.11 vehicle mph (yellow) & throttle %0	Scope	False
92	1.12 vehicle mph (yellow) & throttle %01	Scope	True

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

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

89	1.10.6 Ti	Output	False
90	1.10.7 Tout	Output	False
91	1.11 vehicle mph (yellow) & throttle %0	Scope	False
92	1.12 vehicle mph (yellow) & throttle %01	Scope	False

- 5** Delete the copied object (vehicle mph (yellow) & throttle %1 and resynchronize the model.
- 6** Save the surrogate module.
- 7** Save the sf_car_doors model.

Navigate with the Surrogate Module

In this section...

“Navigate Between Requirements and the Surrogate Module in the DOORS Database” on page 7-17

“Navigate Between DOORS Requirements and the Simulink Module via the Surrogate Module” on page 7-18

Navigate Between Requirements and the Surrogate Module in the DOORS Database

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

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

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

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

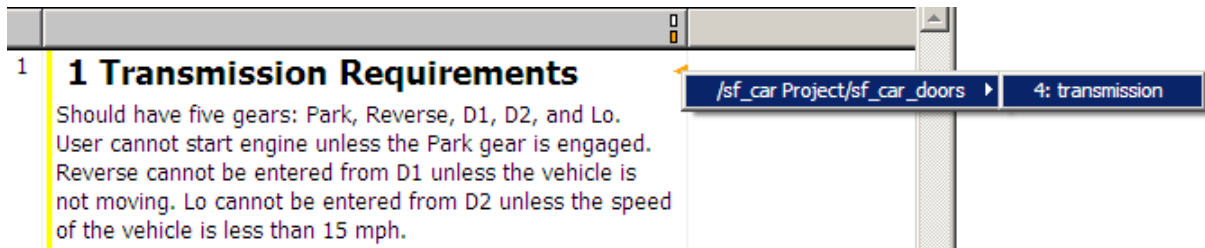
65	1.9.3.7 up_th	Outport	False
4	1.10 transmission	/sf_car Project/sf_car Requirements ▶ 1: Transmission Requirements: Shoul	
66	1.10.1 Ne	Inport	False

- 2 Select the requirement name.

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

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

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



- 2 Select the object name.

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

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

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

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

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

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

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

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

The formal module opens, at **Transmission Requirements**.

Navigate from a Requirement to the Model via the Surrogate Module

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

- 1** In the formal module, in the `Transmission Requirements` object, right-click the left-facing orange arrow.
- 2** Select the path to the linked surrogate object: `/sf_car Project/sf_car_doors > 4. transmission`.

The surrogate module opens, at the transmission object.

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

The linked object is highlighted in `sf_car_doors`.

Navigation from Requirements Documents

- “Why Add Navigation Objects to DOORS Requirements?” on page 8-2
- “Configure Requirements Management Interface for DOORS Software” on page 8-3
- “Enable Linking from DOORS Databases to Simulink Objects” on page 8-5
- “Insert Navigation Objects into DOORS Requirements” on page 8-7
- “Customize DOORS Navigation Objects” on page 8-9
- “Navigate Between DOORS Requirement and Model Object” on page 8-10
- “Diagnose and Fix DXL Errors” on page 8-12
- “Why Add Navigation Objects to Microsoft Office Requirements?” on page 8-13
- “Enable Linking from Microsoft Office Documents to Simulink Objects” on page 8-14
- “Insert Navigation Objects in Microsoft Office Requirements Documents” on page 8-16
- “Customize Microsoft Office Navigation Objects” on page 8-18
- “Navigate Between Microsoft Word Requirement and Model” on page 8-19
- “Navigate with Objects Created Using ActiveX in Microsoft Office 2007 and 2010” on page 8-21

Why Add Navigation Objects to DOORS Requirements?

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

When you create a link from a Simulink object to a DOORS requirement, the RMI stores the link data in Simulink. Using this link, you can navigate from the Simulink object to its associated requirement.

You can also configure the RMI to insert a navigation object in the DOORS database. This navigation object serves as a link from the DOORS requirement to its associated Simulink object.

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

Configure Requirements Management Interface for DOORS Software

In this section...

“Before You Begin” on page 8-3

“Manually Install Additional Files for DOORS Software” on page 8-3

Before You Begin

If you plan to use DOORS software with the RMI, make sure to install additional files to establish communication between the DOORS application and the Simulink software. Follow the instructions in “Configure RMI for IBM Rational DOORS or Microsoft ActiveX Navigation” on page 3-16.

Manually Install Additional Files for DOORS Software

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

You can also manually install the required files into the specified folders, as described in the following steps:

- 1 If the DOORS software is running, close the application.
- 2 Copy the following files from `matlabroot\toolbox\shared\reqmgt\dx1` to the `<doors_install_dir>\lib\dx1\addins` folder.

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

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

- 3 Copy the following files from `matlabroot\toolbox\shared\reqmgt\dx1` to the `<doors_install_dir>\lib\dx1\addins\dmi` folder.

```
dmi.hlp  
dmi.idx
```

```
dmi.inc  
runsim.dxl  
selblk.dxl
```

Replace any existing versions of these files.

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

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

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

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

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

```
rmi setup
```


Enable Linking from DOORS Databases to Simulink Objects

By default, the RMI does not insert navigation objects into requirements documents. If you want to insert a navigation object into the requirements document when you create a link from a Simulink object to a requirement, you must change the RMI's settings. The following tutorial uses the `sldemo_fuelsys` example model to illustrate how to do this.

To enable linking from the DOORS database to the example model:

- 1 Open the model:

```
sldemo_fuelsys
```

Note: You can modify requirements settings in the Requirements Settings dialog box. These settings are global and not specific to open models. Changes you make apply not only to open models, but also persist for models you subsequently open. For more information about these settings, see “Requirements Settings” on page 3-20.

- 2 Select **Analysis > Requirements Traceability > Settings**.

The Requirements Settings dialog box opens.

- 3 Click the **Selection Linking** tab.
- 4 Select **Modify destination for bidirectional linking**.

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

- 5 Select **Store absolute path to model file**.

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

If you add requirements to a model that is not on the MATLAB path, you must select this option to enable linking from your requirements document to your model.

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

For more information about user tags, see “User Tags and Requirements Filtering” on page 5-25.

- 7 Click **Close** to close the Requirements Settings dialog box. Keep the `sldemo_fuelsys` model open.


Insert Navigation Objects into DOORS Requirements


When you enable **Modify destination for bidirectional linking** as described in “Enable Linking from DOORS Databases to Simulink Objects” on page 8-5, the RMI can insert a navigation object into both the Simulink object and its associated DOORS requirement. This tutorial uses the `sldemo_fuelsys` example model to illustrate how to do this. For this tutorial, you also need a DOORS formal module that contains requirements.

- 1 Rename the `sldemo_fuelsys` model and save it in a writable folder on the MATLAB path.
- 2 Start the DOORS software and open a formal module that contains requirements.
- 3 Select the requirement that you want to link to by left-clicking that requirement in the DOORS database.
- 4 In the `sldemo_fuelsys` model, select an object in the model.

This example creates a requirement from the `fuel_rate_control` subsystem.

- 5 Right-click the Simulink object (in this case, the `fuel_rate_control` subsystem) and select **Requirements Traceability > Link to Selection in DOORS**.

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

ID	
1	1 Fuel rate controller requirements The controller will use engine speed, throttle position and manifold pressure to airflow through the engine.
2	[Simulink reference: <code>sldemo_fuelsys/fuel_rate_control</code> (SubSystem)] 

- 6 Close the model.

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

Insert Navigation Objects to Multiple Simulink Objects

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

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

- 1 Make sure that you have enabled **Modify destination for bidirectional linking**.
- 2 Select the DOORS requirement to link to.
- 3 Select the Simulink objects that need to link to that requirement.
- 4 Right-click one of the objects and select **Requirements Traceability > Link to Selection in DOORS**.


A single navigation object is inserted at the selected requirement.

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

Customize DOORS Navigation Objects

Abstract

Edit and customize navigation objects in your DOORS requirements documents.

If the RMI is configured to modify the destination for bidirectional linking as described in “Enable Linking from DOORS Databases to Simulink Objects” on page 8-5, the RMI can insert a navigation object into your requirements document. This object looks like the icon for the Simulink software: 

Note: In IBM Rational DOORS requirements documents, clicking a navigation object does not navigate back to your Simulink object. Select **MATLAB > Select object** to find the Simulink object that contains the requirements link.

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

- 1 Select **Analysis > Requirements Traceability > Settings**.
- 2 Select the **Selection Linking** tab.
- 3 Select **Modify destination for bidirectional linking**.

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

- 4 Select **Use custom bitmap for navigation controls in documents**.
- 5 Click **Browse** to locate the file you want to use for the navigation objects.

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

- 6 Select the desired file to use for navigation objects and click **Open**.
- 7 Close the Requirements Settings dialog box.

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

Tip You can specify a custom template for labels of requirements links to DOORS objects. For more information, see the `rmi` command.

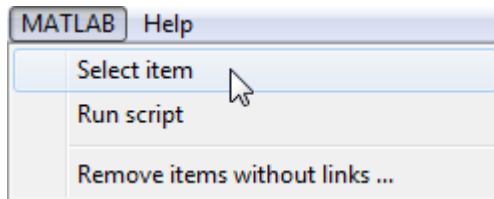
Navigate Between DOORS Requirement and Model Object

Abstract

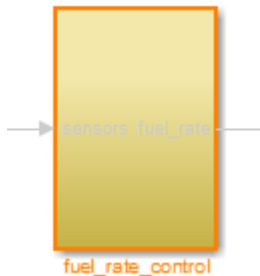
Navigate from a model to a DOORS requirements document and vice-versa.

In “Insert Navigation Objects into DOORS Requirements” on page 8-7, you created a link between a DOORS requirement and the `fuel_rate_control` subsystem in the `sldemo_fuelsys` model. Navigate the links in both directions:

- 1 With the `sldemo_fuelsys` model closed, go to the DOORS requirement in the formal module.
- 2 Left-click the Simulink reference object that you inserted to select it.
- 3 Select **MATLAB > Select item**.



Your version of the `sldemo_fuelsys` model opens, with the `fuel_rate_control` subsystem highlighted.



- 4 Log in to the DOORS software.
- 5 Navigate from the model to the DOORS requirement. In the Model Editor, right-click the `fuel_rate_control` subsystem and select **Requirements Traceability > 1**.

“<**requirement name**>” where <**requirement name**> is the name of the DOORS requirement that you created.

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

1 Fuel rate controller requirements

The controller will use engine speed, throttle position and manifold pressure airflow through the engine.

[Simulink reference: sldemo_fuelsys_test/fuel_rate_control (SubSystem)]



Diagnose and Fix DXL Errors

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

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

If you see these errors, exit the DOORS software, rerun the steps in “Configure RMI for IBM Rational DOORS or Microsoft ActiveX Navigation” on page 3-16, and restart the DOORS software.

Why Add Navigation Objects to Microsoft Office Requirements?

You can use the Microsoft Word and Microsoft Excel applications to capture, track, and manage requirements. The Requirements Management Interface (RMI) allows you to link Simulink objects to requirements managed by external applications.

When you create a link from a Simulink object to a requirement in a Microsoft Office document, the RMI stores the link data in Simulink. Using this link, you can navigate from the Simulink object to its associated requirement.

You can also configure the RMI to insert a navigation object in a Microsoft Office requirements document. This navigation object serves as a link from the requirement to its associated Simulink object.

Enable Linking from Microsoft Office Documents to Simulink Objects

By default, the RMI does not insert navigation objects into requirements documents. If you want to insert a navigation object into the requirements document when you create a link from a Simulink object to a requirement, you must change the RMI's settings. The following tutorial uses the `slvndemo_fuelsys_officereq` example model to illustrate how to do this.

The RMI can insert navigation objects into the following Microsoft Office applications:

- Microsoft Excel
- Microsoft Word

To enable linking from a Microsoft Office document to the example model:

- 1 Open the model:

```
slvndemo_fuelsys_officereq
```

Note: You can modify requirements settings in the Requirements Settings dialog box. These settings are global and not specific to open models. Changes you make apply not only to open models, but also persist for models you subsequently open. For more information about these settings, see “Requirements Settings” on page 3-20.

- 2 Select **Analysis > Requirements Traceability > Settings**.

The Requirements Settings dialog box opens.

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

- Enable **Modify destination for bidirectional linking**.

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

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

For more information about user tags, see “User Tags and Requirements Filtering” on page 5-25.

- 4 Click **Close** to close the Requirements Settings dialog box. Keep the `slvndemo_fuelsys_officereq` model open.

Insert Navigation Objects in Microsoft Office Requirements Documents

Abstract

Use selection-based linking to insert navigation objects in Microsoft Office requirements documents.

Use selection-based linking to create a link from the `slvndemo_fuelsys_officereq` model to a requirements document. If you have configured the RMI as described in “Enable Linking from Microsoft Office Documents to Simulink Objects” on page 8-14, the RMI can insert a navigation object into the requirements document.

- 1 Open the Microsoft Word requirements document:

```
matlabroot/toolbox/slvnv/rmidemos/fuelsys_req_docs/  
slvndemo_FuelSys_RequirementsSpecification.docx
```

- 2 Select the **Throttle Sensor** header.
- 3 In the `slvndemo_fuelsys_officereq` model, open the engine gas dynamics subsystem.
- 4 Right-click the Throttle & Manifold subsystem and select **Requirements Traceability > Link to Selection in Word**.
- 5 The RMI inserts an URL-based link into the requirements document.

1.1.6. Throttle Sensor

Insert Navigation Object That Links to Multiple Simulink Objects

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

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

- 1 Make sure that the RMI is configured to insert navigation objects into requirements documents, as described in “Enable Linking from Microsoft Office Documents to Simulink Objects” on page 8-14.
- 2 Select the Microsoft Word requirement to link to.
- 3 Select the Simulink objects that need to link to that requirement.
- 4 Right-click one of the Simulink objects and select **Requirements Traceability > Link to Selection in Word**.


A single navigation object is inserted at the selected requirement.

- 5 Follow the navigation object link in Microsoft Word to highlight the Simulink objects that are linked to that requirement.

Customize Microsoft Office Navigation Objects

Abstract

Edit and customize navigation objects in your Microsoft Office requirements documents.

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

Note: In Microsoft Office requirements documents, following a navigation object link highlights the Simulink object that contains a bidirectional link to the associated requirement.

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

- 1 Select **Analysis > Requirements Traceability > Settings**.
- 2 Select the **Selection Linking** tab.
- 3 Select **Modify destination for bidirectional linking**.

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

- 4 Select **Use custom bitmap for navigation controls in documents**.
- 5 Click **Browse** to locate the file you want to use for the navigation objects.

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

- 6 Select the desired file to use for navigation objects and click **Open**.
- 7 Close the Requirements Settings dialog box.

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

Navigate Between Microsoft Word Requirement and Model

In “Insert Navigation Objects in Microsoft Office Requirements Documents” on page 8-16, you created a link between a Microsoft Word requirement and the Throttle & Manifold subsystem in the `slvndemo_fuelsys_officereq` example model. Navigate these links in both directions:

- 1 In the `slvndemo_fuelsys_officereq` model, right-click the Throttle & Manifold subsystem and select **Requirements Traceability > 1. “Throttle Sensor”**.

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

1.1.6. Throttle Sensor


- 2 In the requirements document, next to **Throttle Sensor**, follow the navigation object link.

The engine gas dynamics subsystem opens, with the Throttle & Manifold subsystem highlighted.



Navigation from Microsoft Office requirements documents is not automatically enabled upon MATLAB startup. Navigation is enabled when you create a new requirements link or when you have enabled bidirectional linking as described in “Insert Navigation Objects in Microsoft Office Requirements Documents” on page 8-16.

Note: You cannot navigate to requirements from Microsoft Word 2013 onwards when the document is open in read-only mode. Alternately, consider disabling the “Open e-mail attachments and other uneditable files in reading view” option in the Microsoft Word options or using editable documents.

When attempting navigation from requirements links with the  icon, if you get a “Server Not Found” or similar message, enter the command `rmi('httpLink')` to activate the internal MATLAB HTTP server.

Navigate with Objects Created Using ActiveX in Microsoft Office 2007 and 2010

In this section...

“Save Requirements Documents to Microsoft Word 2007 or 2010 Format” on page 8-21

“Field Codes in Requirements Documents” on page 8-22

“ActiveX Control Does Not Link to Model Object” on page 8-24

“Delete an ActiveX Control from Microsoft Excel 2007” on page 8-26

“Delete an ActiveX Control from Microsoft Excel 2010” on page 8-27

Save Requirements Documents to Microsoft Word 2007 or 2010 Format

Abstract

Work with navigation objects for earlier versions of Microsoft Office.

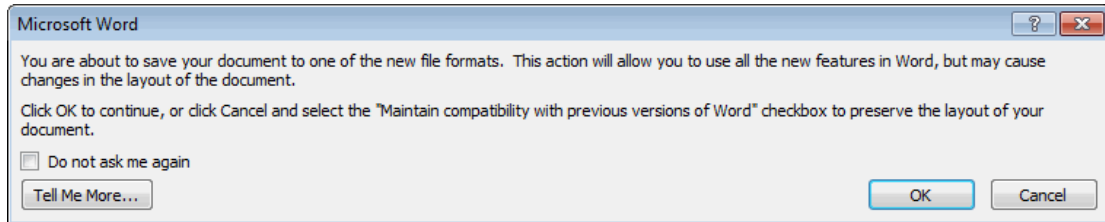
If you create a requirements document with an earlier version of Microsoft Word than Word 2007, links to the corresponding Simulink objects automatically work. If you open a document created in an earlier version and then save it in Microsoft Word 2007 format, make sure that the links to the models continue to work:

- 1 Open a requirements document saved in a release of Microsoft Word earlier than Microsoft Word 2007.
- 2 Depending on which version of Microsoft Word you are running, do one of the following:
 - In Microsoft Word 2007, in the upper-left corner, click the **Microsoft Office Button**.



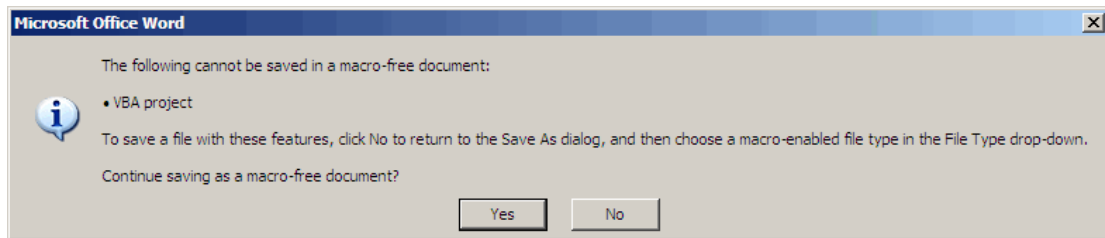
- In Microsoft Word 2010, select the **File** tab.
- 3 Select **Save As > Word Document**.

You see the following dialog box.



- 4 Click **OK**.

If you are running Microsoft Word 2007, you see the following dialog box.




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

Note: You might need to enable ActiveX controls in the Microsoft Office Trust Center.

Field Codes in Requirements Documents

If your Microsoft Word requirements document displays the field codes in addition to, or instead of, the ActiveX icon, clear the **Show field codes instead of their values** option.

The following graphic shows a requirements document created in Microsoft Word 2003, with the field codes (CONTROL mwSimuLink1.SLRefButton \s) displayed.

Determination of pumping efficiency{ CONTROL
 mwSimulink1.SLRefButton `s } 
Requirement ID: REQ2
Model Element: fuelsys/fuel rate controller/Airflow calculat
Details: The airflow calculation will use a calibratib
 pumping efficiency of the engine based on
 manifold pressure.

The following graphic shows a requirements document created in Microsoft Word 2007, with the field codes (CONTROL mwSimulink1.SLRefButton) displayed.

Primary Requirements

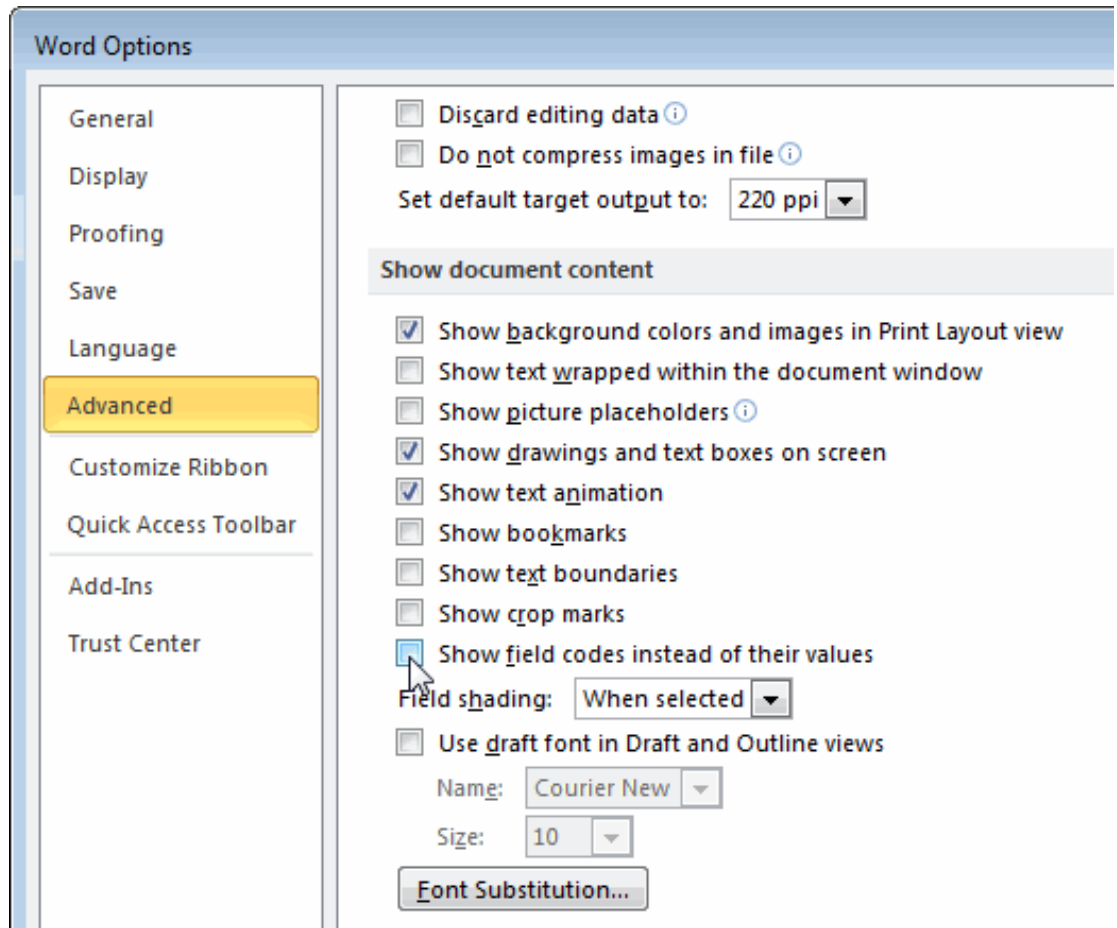
Requirement text. Requirement text. Requirement text.
 Requirement text. Requirement text. Requirement text.
 Requirement text. Requirement text. Requirement text. { CONTROL mwSimulink1.SLRefButton }
 Requirement text. Requirement text. Requirement text.
 Requirement text. Requirement text. Requirement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

To hide the field codes and display the ActiveX icon:

- 1 Depending on which version of Microsoft Word you are running, do one of the following:
 - In Microsoft Word 2007, in the upper-left corner, click the **Microsoft Office Button** and at the bottom of the window, click **Word Options**.



- In Microsoft Word 2010, select **File > Options**.
- 2 In the left-hand portion of the pane, click **Advanced**.
 - 3 In the **Advanced** pane, scroll to the **Show document content** section and clear the **Show field codes instead of their values** option.



ActiveX Control Does Not Link to Model Object

If you click an ActiveX control that links to a Simulink or Stateflow object, and the object does not open, do one of the following:

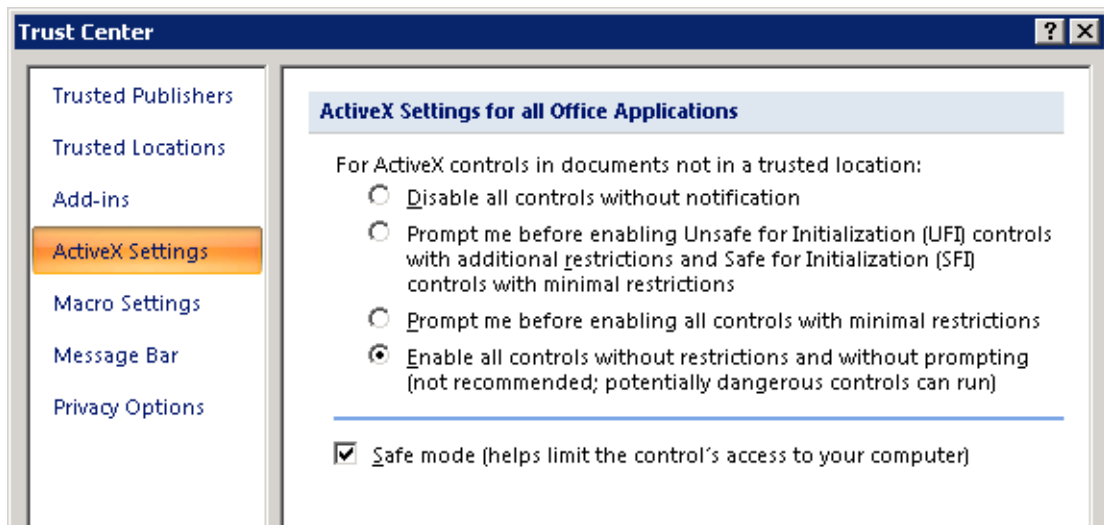
- Store your requirements documents in trusted locations, as described in the Microsoft Office 2007 documentation. The Trust Center does not check files for ActiveX controls stored in trusted locations, so you can maintain your Trust Center restrictions.
- Enable ActiveX controls:

- 1 Depending on which version of Microsoft Office you are using, do one of the following:
 - In Microsoft Word 2007 or Microsoft Excel 2007, in the upper-left corner, click the **Microsoft Office Button**.



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

- In Microsoft Word 2010 or Microsoft Excel 2010, select **File > Options**.
- 2 In the left-hand portion of the pane, click **Trust Center**.
 - 3 In the **Trust Center** pane, click **Trust Center Settings**.
 - 4 In the **Trust Center** pane, on the right, select **ActiveX Settings**.

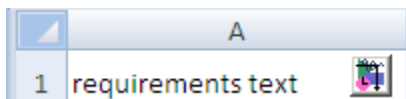


- 5 Select the setting that you want for ActiveX controls:
 - **Prompt me for enabling all controls with minimal restrictions** to decide each time you click an ActiveX control if you want to enable all controls.

- **Enable all controls without restrictions and without prompting** to enable all ActiveX controls whenever you open the document.
- 6 Close the open dialog boxes.
 - 7 Restart the application for the settings to take effect.

Delete an ActiveX Control from Microsoft Excel 2007

Your document may have an ActiveX control in a worksheet cell:



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

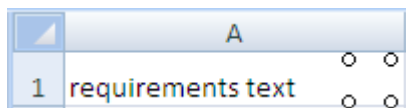
- 1 In Microsoft Excel 2007, in the upper-left corner, click the **Microsoft Office Button**.



- 2 In the pane that opens, at the bottom, click **Excel Options**.
- 3 In the Microsoft Excel Options dialog box, in the left-hand pane, click **Popular**.
- 4 On the **Popular** pane, in the **Top options for working with Excel** section, select **Show Developer tab in the Ribbon**.
- 5 Click **OK**.
- 6 In the Ribbon, on the **Developer** tab, select **Design Mode**.

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

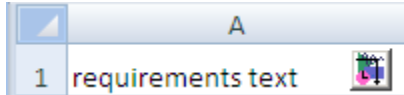
- 7 Click where the ActiveX control was, and you see four handles showing the location of the control.



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

Delete an ActiveX Control from Microsoft Excel 2010

Your document may have an ActiveX control in a worksheet cell:



To remove an ActiveX control from your Microsoft Excel 2010 spreadsheet:

- 1 Select the control by clicking it.
- 2 Select **Home** > **Cut** to delete the control.

MATLAB Code Traceability

- “Link Between MATLAB Code Lines and Requirements Information in External Documents” on page 9-2
- “Enable or Disable Highlighting of Traceability Links for MATLAB Code” on page 9-4
- “Remove Traceability Links from MATLAB Code Lines” on page 9-5
- “Traceability for MATLAB Code Lines” on page 9-6
- “Associate Traceability Information with MATLAB Code Lines in Simulink” on page 9-8

Link Between MATLAB Code Lines and Requirements Information in External Documents

In this section...

“Create Link Using Context Menu Shortcuts” on page 9-2

“Create Link Using Link Editor” on page 9-2

Create Link Using Context Menu Shortcuts

To create requirements traceability links from MATLAB code lines to selections in Microsoft Word, Microsoft Excel, or IBM Rational DOORS documents, you can use shortcuts in the Requirements Traceability context menu.

- 1 In your requirements document, select the target for the traceability link that you want to create.
- 2 In the MATLAB Editor, select the line or lines of code that you want to link.
- 3 In the MATLAB Editor, right-click your selection.
- 4 From the context menu, select **Requirements Traceability**. Depending on the type of your requirements document, select one of the following options:
 - **Link to Selection in Word**
 - **Link to Selection in Excel**
 - **Link to Selection in DOORS**

The software creates a traceability link from the selected MATLAB code range to the selection in the requirements document. It also inserts a navigation object for the selection in the requirements document. The navigation object links to the selected MATLAB code range.

Create Link Using Link Editor

You can create, edit, and delete traceability links with the Link Editor. To open the Link Editor:

- In the MATLAB Editor, select the line or lines of code that you want to link.
- Right-click your selection.

- From the context menu, select **Requirements Traceability > Open Link Editor**.

For detailed information on the Link Editor, see “Requirements Traceability Link Editor” on page 3-17.

Enable or Disable Highlighting of Traceability Links for MATLAB Code

In this section...

“Enable Traceability Highlighting of MATLAB Code” on page 9-4

“Disable Traceability Highlighting of MATLAB Code” on page 9-4

Enable Traceability Highlighting of MATLAB Code

To highlight traceability links in your MATLAB code, do one of the following:

- In the **View** tab, in the **Display** section, select **Highlight Traceability**.
- In the MATLAB Editor, right-click in a line of code with a traceability link. From the context menu, select **Requirements Traceability > Enable Traceability Highlighting**.

Disable Traceability Highlighting of MATLAB Code

To turn off highlighting of traceability links in your MATLAB code, do one of the following:

- In the **View** tab, in the **Display** section, clear **Highlight Traceability**.
- In the MATLAB Editor, right-click in a line of code with a traceability link. From the context menu, select **Requirements Traceability > Disable Traceability Highlighting**.

Remove Traceability Links from MATLAB Code Lines

In this section...

“Delete Links to Requirements from MATLAB Code Lines” on page 9-5

“Delete Link Targets in MATLAB Code Lines” on page 9-5

Delete Links to Requirements from MATLAB Code Lines

To remove requirements traceability links from a line or lines of MATLAB code:

- 1 In the MATLAB Editor, right-click within a range of code that has requirements traceability links.
- 2 From the context menu, select **Requirements Traceability > Delete All Links**.

All links to requirements from this MATLAB code range are deleted. Links to this MATLAB code range from external requirements documents are not deleted.

Delete Link Targets in MATLAB Code Lines

If you have links to MATLAB code ranges from external requirements documents, you can delete the targets for these links from your MATLAB code.

To remove requirements traceability targets from a line or lines of MATLAB code, after you have deleted its outgoing links as described in the previous section:

- 1 In the MATLAB Editor, right-click within a previously linked range of code.
- 2 From the context menu, select **Requirements Traceability > Discard Named Range**.

When you discard a named range, links to that MATLAB code range from external documents no longer work. Note, however, that this does not delete navigation objects in external requirements documents.

Traceability for MATLAB Code Lines

In this section...

“Traceability Link Targets” on page 9-6

“Storage of Traceability Links” on page 9-6

“Limitations of MATLAB Code Traceability” on page 9-7

Traceability Link Targets

You can create MATLAB code traceability links for:

- Lines of MATLAB code in a standalone file.
- Lines of MATLAB code inside a MATLAB Function block.

You can create links from a line or lines of MATLAB code to:

- Objects in Simulink models.
- Targets in Microsoft Word or Microsoft Excel documents.
- Targets in IBM Rational DOORS databases.
- Targets in text, HTML, or PDF documents.
- HTTP URLs.

Bidirectional linking is supported for targets in MATLAB, Simulink, Microsoft Word, Microsoft Excel, and IBM Rational DOORS. Bidirectional linking creates links to and from the selected link destination. To enable bidirectional linking, in the Requirements Settings dialog box, under the Selection Linking tab, select **Modify destination for bidirectional linking**. For more information, see “Selection Linking Tab” on page 3-20.

You can also create links to MATLAB code lines from any external application that supports HTTP navigation. For more information, see “Link from External Applications” on page 3-24.

Storage of Traceability Links

In a standalone MATLAB file, you can create, navigate, and delete traceability links for lines of code without changing the MATLAB file. The Requirements Management Interface stores requirements traceability data for a MATLAB file in a `.req` file with the same name and location as the MATLAB file.

If you want to create traceability links for lines of code in a MATLAB Function block, set the parent model to store requirements data externally. For a new model, see “Specify Storage for Requirements Links” on page 4-4. For an existing model, see “Move Internally Stored Requirements Links to External Storage” on page 4-7. When you create traceability links for code inside a MATLAB Function block, the RMI stores them in a `.req` file for the parent model. The `.req` file for the model contains requirements traceability data for linked model objects and for linked code in MATLAB Function blocks in the model.

Limitations of MATLAB Code Traceability

Overlapping Linked Ranges

The software does not support traceability links for overlapping regions of MATLAB code. If one linked range of code completely overlaps another smaller region of code, the link for the larger range overshadows the link for the smaller range. To avoid complications from overlapping linked ranges, when you create traceability links for MATLAB code lines, choose ranges of code that do not overlap.

Cut and Paste Operations

You can cut or copy a selection of code that has traceability links. When you paste that selection, the software attempts to recreate the corresponding traceability links at the paste location. Depending on paste location and code formatting, you might need to recreate the traceability links manually.

Drag Operation

If you select code that has traceability links and drag that code to a new location, you might need to recreate traceability links for the code in the new location.

MATLAB Function Block Code Traceability in Web View

Requirements linked to individual MATLAB code lines inside a MATLAB Function block appear in HTML requirements traceability reports, but do not appear the Simulink Report Generator Web view. See “Create and Use a Web View” in the Simulink Report Generator documentation.

Traceability for MATLAB Live Editor

Requirements traceability is not supported for MATLAB Live Editor.

Associate Traceability Information with MATLAB Code Lines in Simulink

Traceability management support in the MATLAB Editor is an extension of the Simulink-based Requirements Management Interface to allow associations between MATLAB code lines and external artifacts. This capability does not require editing MATLAB files; all traceability data is stored separately. This is similar to "external" storage of RMI links when working with Simulink models.

In addition, using "external storage" mode for managing traceability information, Simulink and Stateflow users can benefit from finer granularity when associating external documents with contents of MATLAB Function blocks.

The included example model has traceability data associated both with Simulink blocks and individual code lines of MATLAB Function blocks.

Open Example Model

This example demonstrates linking between external documents and MATLAB code lines when modeling stimulated spiking in connected neural cells.

Open the `slvndemo_synaptic_transmission` model. There are three Model blocks referencing the same model of a spiking neural cell. The neural cell model follows a "Leaky Integrators" equation:

$$\frac{C_m dV}{dt} = I_{total} - \frac{V - V_0}{R_m}$$

C_m, R_m – capacitance and resistance of cell membrane

I_{total} – includes injected stimulation current and all ion channel currents

V_0 – resting cross-membrane potential, typically -70mV

For the purpose of simulation, we converted to:

$$V \rightarrow V_0 + \int_0^t \frac{1}{C_m} \left(I_{total} - \frac{V - V_0}{R_m} \right) dt$$

Two MATLAB Functions between neurons calculate post-synaptic currents. When presynaptic depolarization crosses the neurotransmitter release threshold, we increment post-synaptic current by one pulse of given amplitude:

$$I \rightarrow I + I_{\text{amplitude}}$$

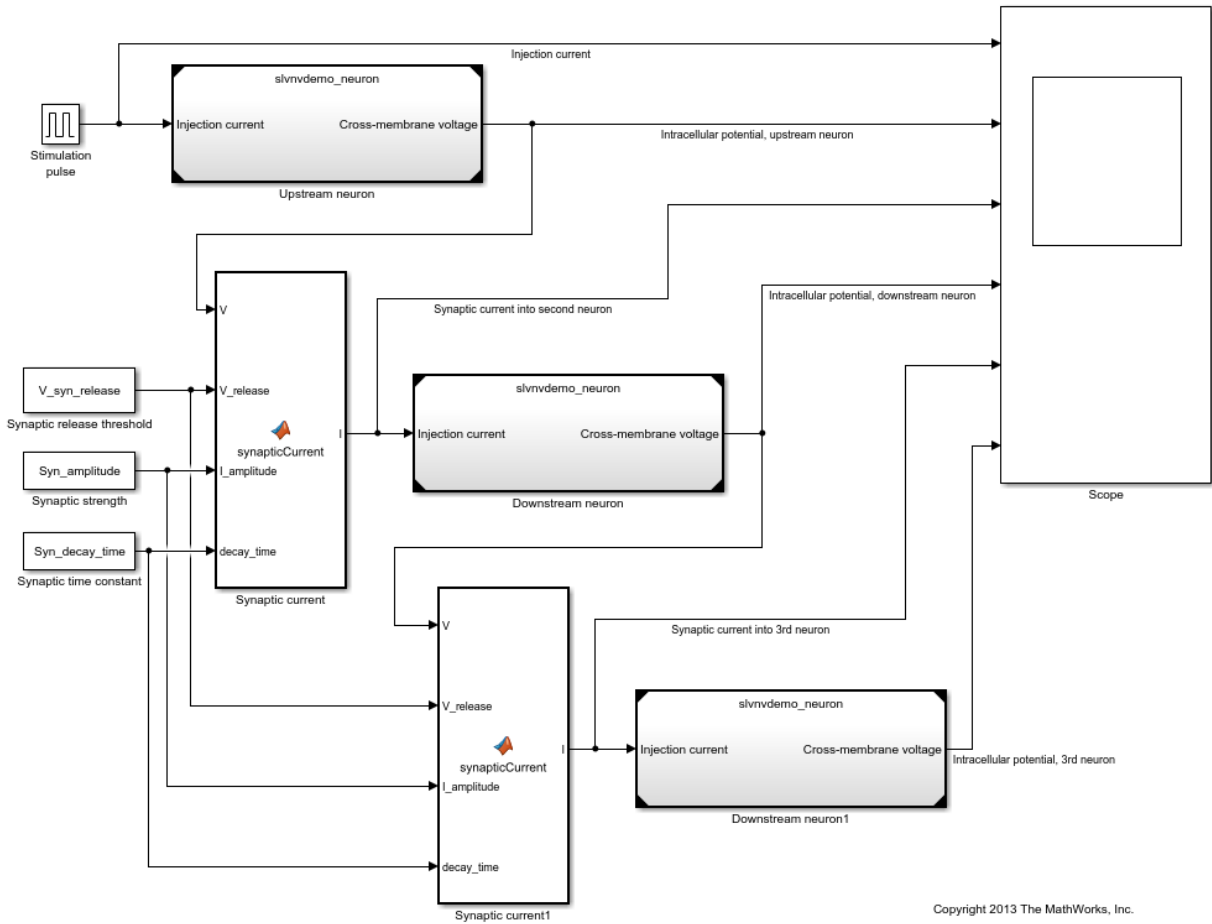
Resulting total current decays exponentially according to

$$dI/dt = -I * \frac{t}{\tau}$$

We disallow the next increment for a certain timeframe after the previous pulse, to model the effect of short-term synaptic depression. Our model neglects the time delay of axonal transmission.

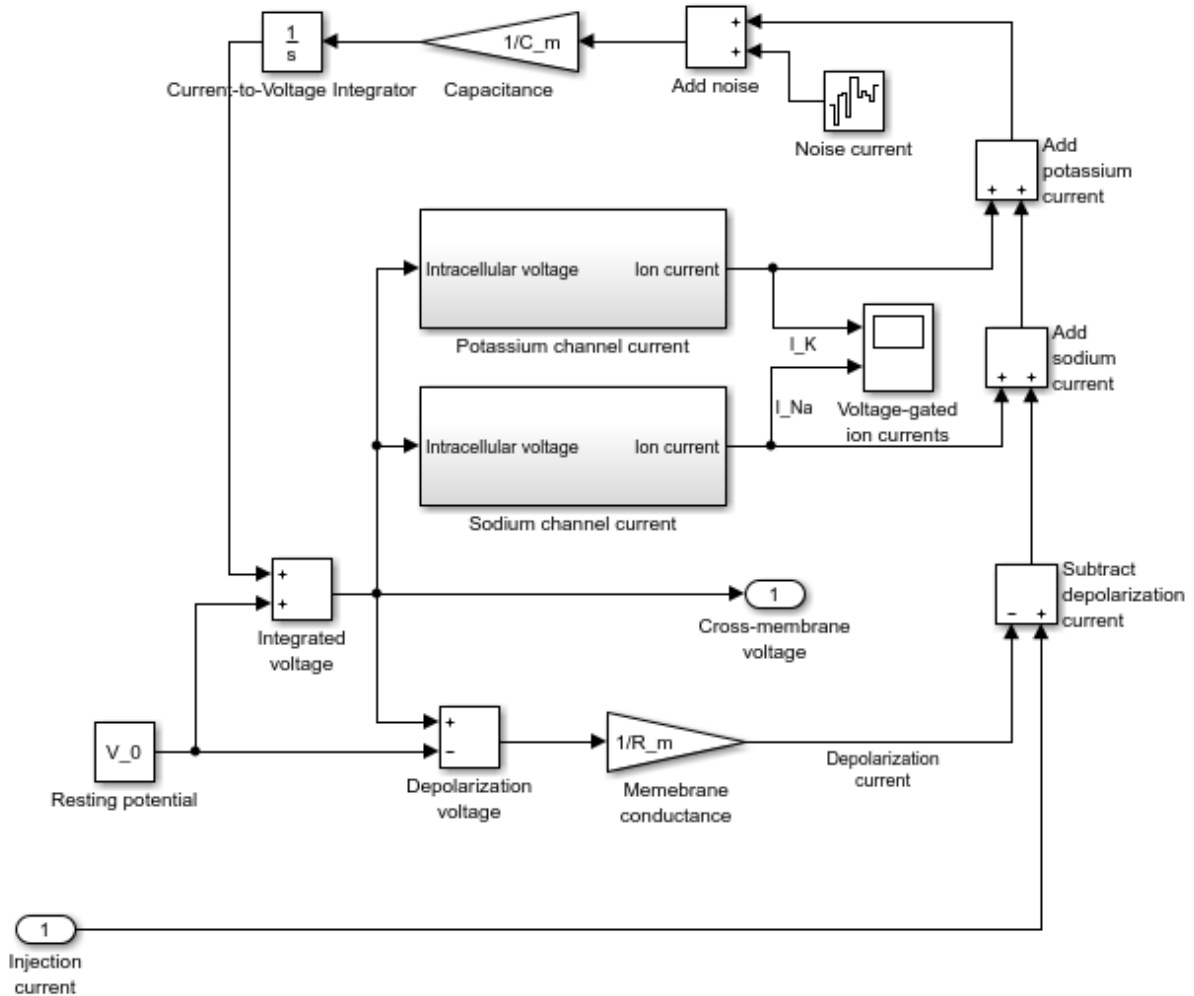
```
open_system('slvndemo_neuron');  
open_system('slvndemo_synaptic_transmission');
```

Modeling Stimulated Spiking in Connected Neural Cells



Copyright 2013 The MathWorks, Inc.

"Integrate-and-Fire" Model of Neural Cell with Voltage-Gated Channels



Copyright 2013 The MathWorks, Inc.

Simulate Model and View Results

Simulate `slvndemo_synaptic_transmission` model. Check Scope for results. The six plots are:

- externally injected electrical current pulse,
- injection-stimulated intracellular voltage spiking of the first neuron,
- post-synaptic current generated in the second neuron,
- synaptically stimulated activity of the second neuron,
- post-synaptic current generated in the third neuron,
- synaptically stimulated activity of the third neuron.

Observe regular spiking of the upstream neuron (plot 2) while stimulation pulse is applied (plot 1). Synaptically induced current in downstream neuron (plot 3) skips some action potentials of the upstream neuron due to short-term neurotransmitter depletion modeled as a temporary turn-off period in Synaptic current function. Downstream neuron is seen to, sometimes, integrate more than one synaptic input to produce a spike (plot 4). The third neuron integrates synaptic inputs from the second neuron (plot 5) and spikes at a later time (plot 6). With our default parameter values, the third neuron may spike 1 or more times, depending on intentionally introduced random noise in the model.

We assign same parameter values for all three neurons and both synapses. Traceability linking is used to justify parameter values and implementation.

```
sim('slvndemo_synaptic_transmission');
```

```
### Successfully updated the model reference SIM target for model: slvndemo_neuron
```

Navigate Between Simulink and Standalone MATLAB Files

The `slvndemo_synaptic_transmission` model runs an external script to load required parameter values into workspace. MATLAB code linking allows you to trace from a dependent block in Simulink, not only to a script file but to the specific line that defines a value used in simulation.

Locate the Stimulation pulse block in the model. Right-click the block and select **Requirements Traceability > 1. "I_inj = 2e-11; % 20 pA"** to follow the link and view the relevant highlighted region in the MATLAB code file. Notice the mismatched value so easily detected via Traceability link. Right-click another highlighted line in the MATLAB

file and navigate back to Simulink by selecting the shortcut in the **Requirements Traceability** context menu.

```
rmi('highlightModel', 'slvsvdemo_synaptic_transmission');  
rmipref('BiDirectionalLinking',true);
```

Create Traceability Link for Lines of MATLAB Code

The Synaptic time constant block in the bottom left corner of the model is not yet linked to its related line in parameters script.

Make sure that bidirectional linking is enabled, so that you can create two-way traceability links in one step. First, in the Simulink Editor, select the Synaptic time constant block. Then, in the MATLAB Editor, select the variable name **Syn_decay_time** at the bottom of the synaptic_params script. Right-click on the selected line and from the context menu, choose **Requirements Traceability > Link to Selection in Simulink**. Test the new links by navigating from MATLAB to Simulink and back to MATLAB.

```
4
5 % Injection current amplitude
6 I_inj = 1.9e-11; % 19 pA
7
8
9
10 % Synapse properties
11
12 V_syn_release = -2e-2; % -20 mV
13
14 Syn_amplitude = 1.5e-11; % 15 pA
15
16 Syn_decay_time = 0.1; % 100 ms
```

The screenshot shows a MATLAB code editor with a context menu open over the variable `Syn_decay_time`. The menu items are as follows:

MenuItem	Shortcut
Evaluate Selection	F9
Open "Syn_decay_time"	Ctrl+D
Help on Selection	F1
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Select All	Ctrl+A
Wrap Comments	Ctrl+J
Comment	Ctrl+R
Uncomment	Ctrl+T
Smart Indent	Ctrl+I
Evaluate Current Section	Ctrl+Enter
Insert Section Breaks Around Selection	
Insert Text Markup	
Function Browser	Shift+F1
Function Hints	Ctrl+F1
Code Folding	
Split Screen	
Requirements Traceability	

The 'Link to Selection in Simulink' option is circled in red. A red arrow points to the 'Requirements Traceability' option at the bottom of the menu.

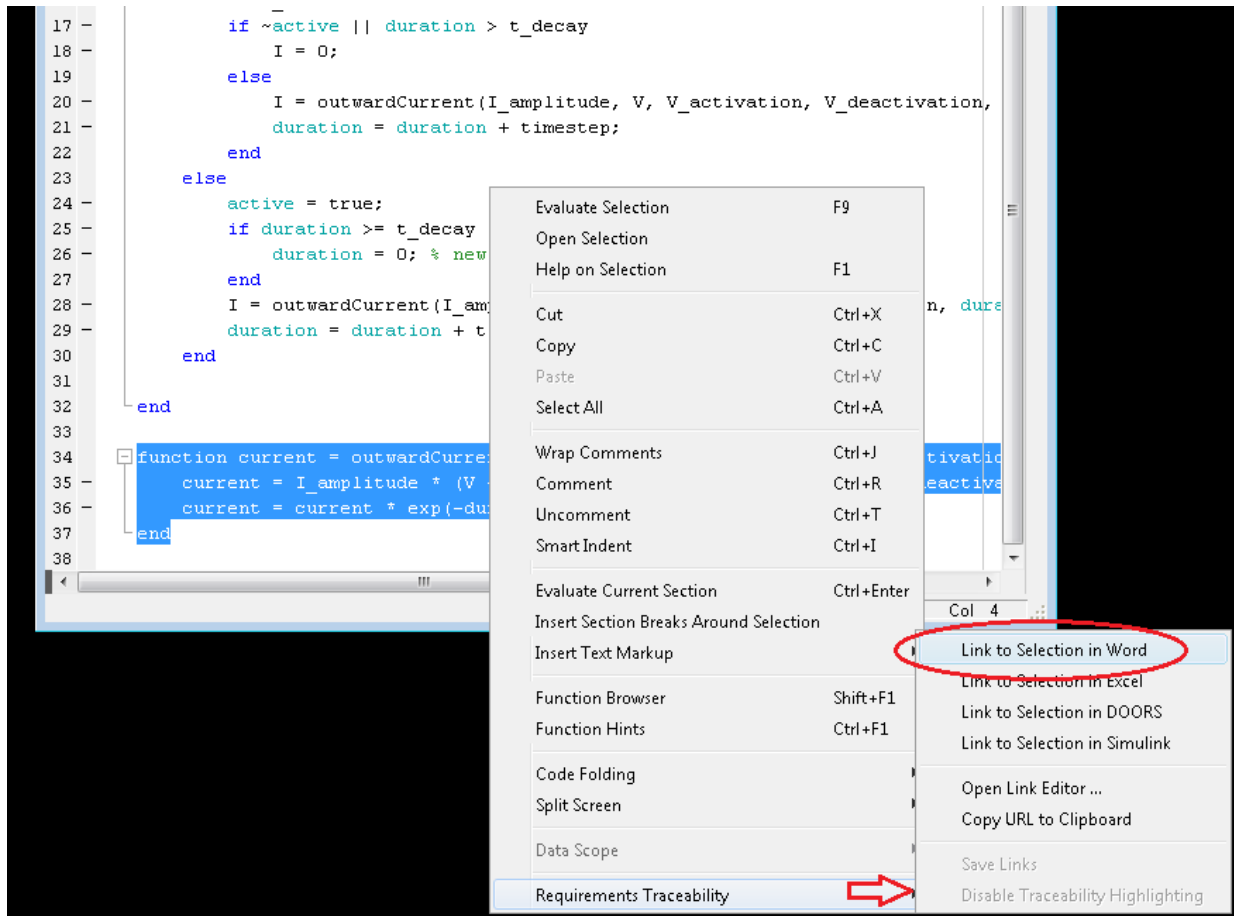
Create Traceability Link for Lines of Code Inside MATLAB Function Blocks


The `slvndemo_synaptic_transmission` model has a MATLAB Function block that we will want to trace to parameter value sources. Instead of linking the block itself, open the MATLAB code of this block and link specific lines. For example, in the Simulink Editor, click on the Synaptic strength constant block to select it for linking. In the MATLAB Editor, find the occurrence of **I_amplitude** on line 33 of the MATLAB Function block code. Right-click the line and from the **Requirements Traceability** context menu, select **Link to Selection in Simulink**.


Create Traceability Link Between Lines in MATLAB Code and Microsoft Word

Until this point we only looked at linking between MATLAB and Simulink. Open the Ion current calculation MATLAB Function block that belongs to the Sodium current calculation subsystem of the referenced `slvndemo_neuron` model. Right-click on a highlighted line or lines and from the **Requirements Traceability** context menu, navigate a link at the top. The associated Word document opens to the associated text.

To create a similar link in another MATLAB Function block of this model, Ion current calculation Function in Potassium channel current subsystem, select the relevant content in the Word document (i.e. **outward current of potassium ions**). Then, open the Potassium channel current MATLAB Function block. In the MATLAB Editor, select the implementation for **outwardCurrent** subfunction (lines 34-37). Right-click the selected lines and in the context menu, select **Requirements Traceability > Link to Selection in Word**. Minimize the Word document, then navigate from the newly highlighted lines at the bottom of the MATLAB code to the related description in Word. When the correct location is highlighted, use the MATLAB icon to navigate back to code.



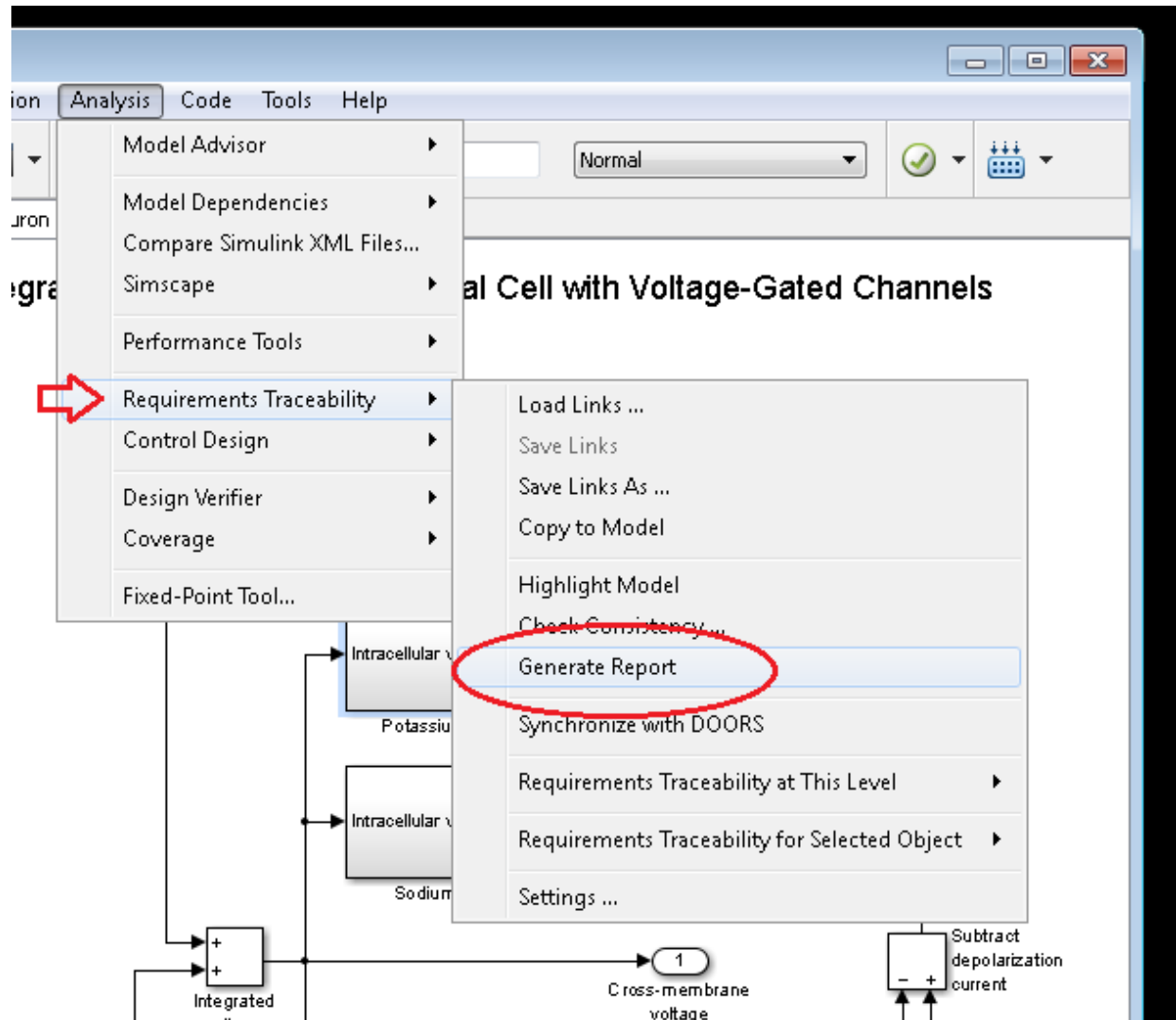
ing the **depolarization** in transmembrane voltage , they
 , which changes the electrochemical gradient, which in turn
 ane potential. This then causes more channels to open,
 and so on. The process proceeds explosively until all of the
 lting in a large upswing in the membrane potential. The
 re polarity of the plasma membrane to reverse, and the ion
 the sodium channels close, sodium ions can no longer enter
 isported out of the plasma membrane.

d, and there is an **outward current** of potassium ions ,
 nt to the resting state. After an action potential has occurred,
 led the afterhyperpolarization or refractory period, due to
 s the mechanism that prevents an action potential from



Report Traceability Links

Traceability links associated with MATLAB code lines of MATLAB Function blocks are included in the **Requirements Traceability Report** generated for the parent Simulink model. Use **Analysis > Requirements Traceability > Generate Report** in slvndemo_neuron model window to generate the report. See MATLAB Function block links information present in the report, including quotations of linked MATLAB Code lines.



Web Browser - Requirements Traceability Report for slvndemo_neuron

Requirements Traceability Report for slvndemo_neuron

Location: file:///L:/Work/tmp/mlink_demo/slvndemo_neuron_requirements.html#aa5668f609ed019cf6615c86744cfa93

Chapter 5. System - Sodium channel current

Table 5.1. Objects in slvndemo_neuron/Sodium channel current that have Requirements Traceability Links

Linked Object	Requirements Traceability Data
Activation threshold	1. "Na_V_activation = -5e-2; % -50 mV" neuron_params_at '735341.1'
Deactivation threshold1	1. "Na_V_deactivation = 5e-2; % +50 mV" neuron_params_at '735341.2'
Intracellular voltage	1. "voltage-gated" _synaptic_doc\neuralspikemodeling.docx_at 'Simulink requirement item 1'
Ion current calculation	+ See below for links from MATLAB Code

Table 5.2. Requirements Traceability Data for code in slvndemo_neuron/Sodium channel current/Ion current calculation

Linked Code	Requirements Traceability Data
Show in Editor 26 I = I_amplitude * (V_deactivation - V) / (V_deactivation - V_activation); 27 I = I * exp(-duration/t_decay);	"depolarization in 1. transmembrane voltage" _synaptic_doc\neuralspikemodeling.docx_at 'Simulink requirement item 2'

Custom Types of Requirements Documents

- “Why Create a Custom Link Type?” on page 10-2
- “Implement Custom Link Types” on page 10-3
- “Custom Link Type Functions” on page 10-4
- “Links and Link Types” on page 10-5
- “Link Type Properties” on page 10-6
- “Custom Link Type Registration” on page 10-10
- “Create a Custom Requirements Link Type” on page 10-11
- “Custom Link Type Synchronization” on page 10-19
- “Navigate to Simulink Objects from External Documents” on page 10-21

Why Create a Custom Link Type?

Abstract

Register custom requirements document types with the Requirements Management Interface (RMI).

In addition to linking to built-in types of requirements documents, you can register custom requirements document types with the Requirements Management Interface (RMI). Then you can create requirement links from your model to these types of documents.

With custom link types, you can:

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

The custom link type API allows you to define MATLAB functions that enable linking between your Simulink model and your custom requirements document type. These functions also enable new link creation and navigation between the model and documents.

For example, navigation involves opening a requirements document and finding the specific requirement record. When you click your custom link in the content menu of a linked object in the model, Simulink uses your custom link type navigation function to open the document and highlight the target requirement based on the implementation provided. The navigation function you implement uses the available API to communicate with your requirements storage application.

Typically, MATLAB runs an operating system shell command or uses ActiveX communication for sending navigation requests to external applications.

Alternatively, if your requirements are stored as custom variants of text or HTML files, you can use the built-in editor or Web browser to open the requirements document.

Implement Custom Link Types

To implement a custom link type:

- 1** Create a MATLAB function file based on the custom link type template, as described in “Custom Link Type Functions” on page 10-4.
- 2** Customize the custom link type file to specify the link type properties and custom callback functions required for the custom link type, as described in “Link Type Properties” on page 10-6.
- 3** Register the custom link type using the `rmi` command `'register'` option, as described in “Custom Link Type Registration” on page 10-10.

Custom Link Type Functions

To create a MATLAB function file, start with the custom link type template, located in:

`matlabroot\toolbox\shared\reqmgt\+linktypes\linktype_TEMPLATE.m`

Your custom link type function:

- Must exist on the MATLAB path with a unique function and file name.
- Cannot require input arguments.
- Must return a single output argument that is an instance of the requirements link type class.

To view similar files for the built-in link types, see the following files in

`matlabroot\toolbox\shared\reqmgt\+linktypes\`:

```
linktype_rmi_doors.m  
linktype_rmi_excel.m  
linktype_rmi_html.m  
linktype_rmi_text.m
```


Links and Link Types

Abstract

Get and set the requirements links on a block.

Requirements links are the data structures, managed by Simulink, that identify a specific location within a document. You get and set the links on a block using the `rmi` command.

Links and link types work together to perform navigation and manage requirements. The `doc` and `id` fields of a link uniquely identify the linked item in the external document. The RMI passes both of these values to the navigation command when you navigate a link from the model.

Link Type Properties

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

Property	Description
Registration	The name of the function that creates the link type. The RMI stores this name in the Simulink model.
Label	A string to identify this link type. In the “Requirements Traceability Link Editor” on page 3-17, this string appears on the Document type drop-down list for a Simulink or Stateflow object.
IsFile	A Boolean property that indicates if the linked documents are files within the computer file system. If a document is a file: <ul style="list-style-type: none"> • The software uses the standard method for resolving the path. • In the Requirements Traceability Link Editor, when you click Browse, the file selection dialog box opens.
Extensions	An array of file extensions. Use these file extensions as filter options in the Requirements Traceability Link Editor when you click Browse . The file extensions infer the link type based on the document name. If you registered more than one link type for the same file extension, the link type that you registered takes first priority.
LocDelimiters	A string containing the list of supported navigation delimiters. The first character in the ID of a requirement specifies the type of identifier. For example, an identifier can refer to a specific page number (#4), a named bookmark (@my_tag), or some searchable text (?search_text). The valid location delimiters determine the possible entries in the Requirements Traceability Link Editor Location drop-down list.
NavigateFcn	The MATLAB callback invoked when you click a link. The function has two input arguments: the document field and the ID field of the link: <pre>feval(LinkType.NavigateFcn, Link.document, Link.id)</pre>

Property	Description
ContentsFcn	<p>The MATLAB callback invoked when you click the Document Index tab in the Requirements Traceability Link Editor. This function has a single input argument that contains the full path of the resolved function or, if the link type is not a file, the Document field contents.</p> <p>The function returns three outputs:</p> <ul style="list-style-type: none"> • Labels • Depths • Locations
BrowseFcn	<p>The MATLAB callback invoked when you click Browse in the Requirements Traceability Link Editor. You do not need this function when the link type is a file. The function takes no input arguments and returns a single output argument that identifies the selected document.</p>
CreateURLFcn	<p>The MATLAB callback that constructs a path name to the requirement. This function uses the document path or URL to create a specific requirement URL. The requirement URL is based on a location identifier specified in the third input argument. The input arguments are:</p> <ul style="list-style-type: none"> • Full path name to the requirements document • Info about creating a URL to the document (if applicable) • Location of the requirement in the document <p>This function returns a single output argument specified as a character vector. Use this argument when navigating to the requirement from the generated report.</p>
IsValidDocFcn	<p>The MATLAB callback invoked when you run a requirements consistency check. The function takes one input argument—the fully qualified name for the requirements document. It returns true if the document can be located; it returns false if the document cannot be found or the document name is invalid.</p>

Property	Description
<p>IsValidIdFcn</p>	<p>The MATLAB callback invoked when you run a requirements consistency check. This function takes two input arguments:</p> <ul style="list-style-type: none"> • Fully qualified name for the requirements document • Location of the requirement in the document <p>IsValidIdFcn returns true if it finds the requirement and false if it cannot find that requirement in the specified document.</p>
<p>IsValidDescFcn</p>	<p>The MATLAB callback invoked when you run a requirements consistency check. This function has three input arguments:</p> <ul style="list-style-type: none"> • Full path to the requirements document • Location of the requirement in the document • Requirement description label as stored in Simulink <p>IsValidDescFcn returns two outputs:</p> <ul style="list-style-type: none"> • True if the description matches the requirement, false otherwise. • The requirement label in the document, if not matched in Simulink.

Property	Description
DetailsFcn	<p>The MATLAB callback invoked when you generate the requirements report with the Include details from linked documents option. This function returns detailed content associated with the requirement and has three input arguments:</p> <ul style="list-style-type: none"> • Full path to the requirements document • Location of the requirement in the document • Level of details to include in report (Unused) <p>The DetailsFcn returns two outputs:</p> <ul style="list-style-type: none"> • Numeric array that describes the hierarchical relationship among the fragments in the cell array • Cell array of formatted fragments (paragraphs, tables, et al.) from the requirement
SelectionLinkFcn	<p>The MATLAB callback invoked when you use the selection-based linking menu option for this document type. This function has two input arguments:</p> <ul style="list-style-type: none"> • Handle to the model object that will have the requirement link • True if a navigation object is inserted into the requirements document, or false if no navigation object is inserted <p>SelectionLinkFcn returns the requirements link structure for the selected requirement.</p>

Custom Link Type Registration

Abstract

Register your custom link type by passing the name of the MATLAB function file to the `rmi` API.

Register your custom link type by passing the name of the MATLAB function file to the `rmi` command as follows:

```
rmi register mytargetfilename
```

Once you register a link type, it appears in the “Requirements Traceability Link Editor” on page 3-17 as an entry in the **Document type** drop-down list. A file in your preference folder contains the list of registered link types, so the custom link type is loaded each time you run MATLAB.

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

You can remove a link type with the following MATLAB command:

```
rmi unregister mytargetfilename
```

Create a Custom Requirements Link Type

Abstract

Implement a custom link type to a hypothetical document type, a text file with the extension `.abc`.

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

You will create a document index listing all the requirement items. When navigating from the Simulink model to the requirements document, the document opens in the MATLAB Editor at the line of the requirement that you want.

To create a custom link requirement type:

- 1 Write a function that implements the custom link type and save it on the MATLAB path.

For this example, the file is `rmicustabcinterface.m`, containing the function, `rmicustabcinterface`, that implements the ABC files shipping with your installation.

- 2 To view this function, at the MATLAB prompt, type:

```
edit rmicustabcinterface
```

The file `rmicustabcinterface.m` opens in the MATLAB Editor. The content of the file is:

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

```

% requirement demo directory to determine the path to the document
% invoke:
%
% >> which demo_req_1.abc

% Copyright 1984-2010 The MathWorks, Inc.

% 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 required

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

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

function openFileToLine(fileName, lineNum)
if lineNum > 0
    if matlab.desktop.editor.isEditorAvailable
        matlab.desktop.editor.openAndGoToLine(fileName, lineNum);
    end
else
    edit(fileName);
end

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

```



```

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

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

% Find all the requirement items
fList1 = 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 = [];

```

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

```
rmi register rmicustabcinterface
```

The ABC file type appears on the “Requirements Traceability Link Editor” on page 3-17 drop-down list of document types.

- 4** Create a text file with the .abc extension containing several requirement items marked by the Requirement:: string.

For your convenience, an example file ships with your installation. The example file is *matlabroot*\toolbox\slvnx\rmidemos\demo_req_1.abc. demo_req_1.abc contains the following content:

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

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

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

Description:

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

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

End of "Altitude Climb Control">

Requirement:: "Altitude Hold"

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

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 of "Altitude Hold"

Requirement:: "Autopilot Disable"

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

Description:

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

ENd of "Autopilot Disable"

Requirement:: "Glide Slope Armed"

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

Units:

Glide Slope Enable - Logical

Glide Slope Signal - Logical

Description:

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

End of "Glide Slope Armed"

Requirement:: "Glide Slope Coupled"

Glide Slope control becomes coupled when the control is armed and (Glide Slope Angle Error > 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 of "Glide Slope Coupled"

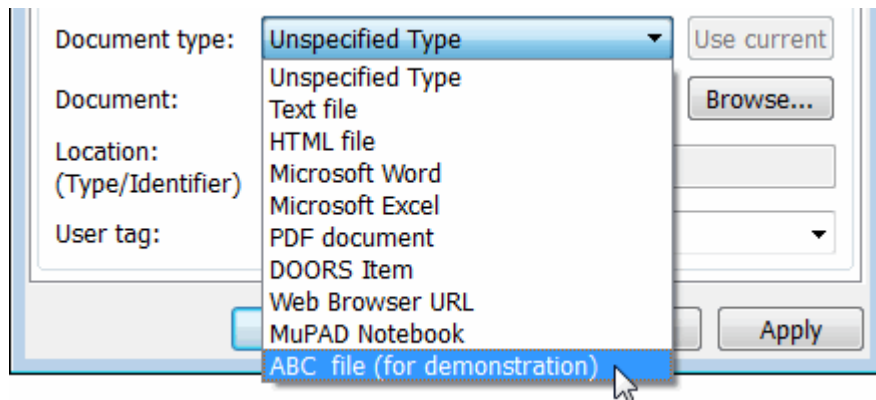
- 5 Open the following example model:

aero_dap3dof

- 6 Right-click the Reaction Jet Control subsystem and select **Requirements Traceability > Open Link Editor**.

The Requirements Traceability Link Editor opens.

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



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

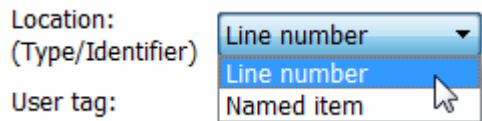
In this example, the *rmicustabcinterface* function specifies two types of location delimiters for your requirements:

- > — Line number in a file

- @ — Named item, such as a bookmark, function, or HTML anchor

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

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



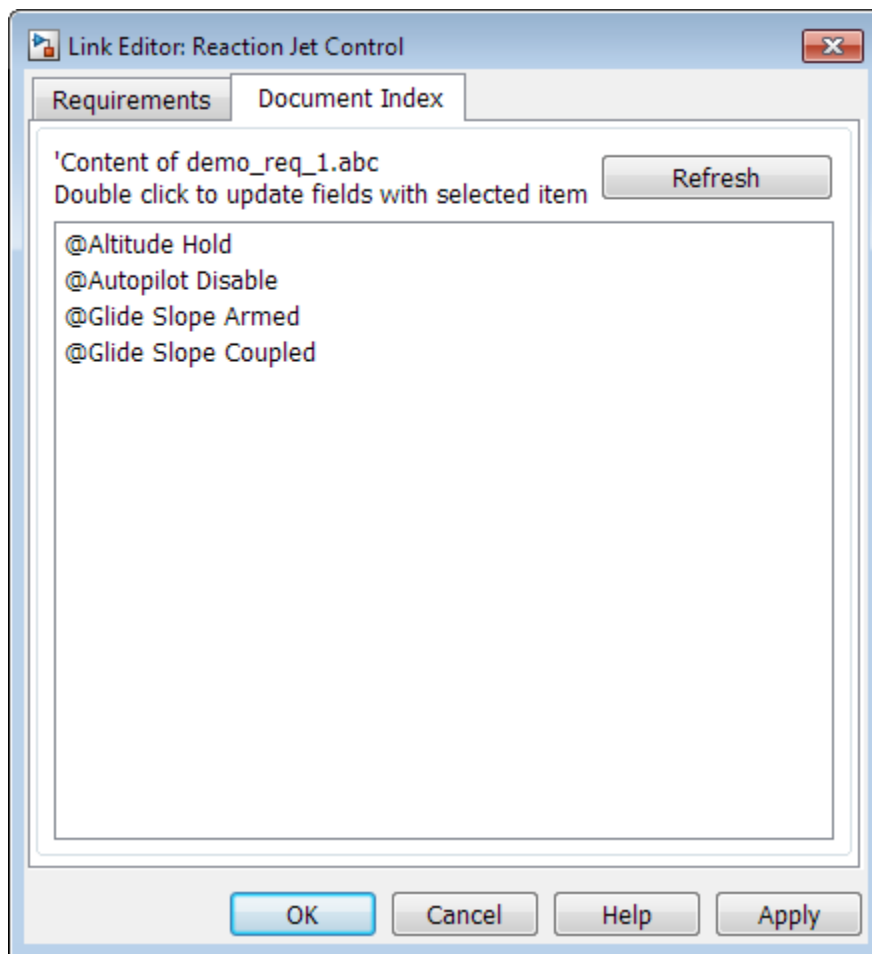
- 10 Select **Line number**. Enter the number 26, which corresponds with the line number for the `Altitude Hold` requirement in `demo_req_1.abc`.
- 11 In the **Description** field, enter `Altitude Hold`, to identify the requirement by name.
- 12 Click **Apply**.
- 13 Verify that the `Altitude Hold` requirement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

Create a Document Index

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

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

For the `demo_req_1.abc` file, in the **Link Editor: Reaction Jet Control** dialog box, click the **Document Index** tab. The `ContentsFcn` function generates the document index automatically.



Custom Link Type Synchronization

After you implement custom link types for RMI that allow you to establish links from Simulink objects to requirements in your requirements management application (RM application), you can implement synchronization of the links between the RM application and Simulink using Simulink Verification and Validation functions. Links can then be reviewed and managed in your RM application environment, while changes made are propagated to Simulink.

You first create the surrogate objects in the RM application to represent Simulink objects of interest. You then automate the process of establishing traceability links between these surrogate objects and other items stored in the RM application, to match links that exist on the Simulink side. After modifying or creating new associations in the RM application, you can propagate the changes back to Simulink. You use Simulink Verification and Validation to implement synchronization of links for custom requirements documents. However, this functionality is dependent upon the automation and inter-process communication APIs available in your RM application. You use the following Simulink Verification and Validation functions to implement synchronization of links between RM applications and Simulink.

To get a complete list of Simulink objects that may be considered for inclusion in the surrogate module:

```
[objHs, parentIdx, isSf, objSIDs] = slrequirements('getObjectsInModel', modelName);
```

This command returns:

- **objHs**, a complete list of numeric handles
- **objSIDs**, a complete list of corresponding session-independent Simulink IDs
- **isSf**, a logical array that indicates which list positions correspond to which Stateflow objects
- **parentIdx**, an array of indices that provides model hierarchy information

When creating surrogate objects in your RM application, you will need to store **objSIDs** values – not **objHs** values – because **objHs** values are not persistent between Simulink sessions.

To get Simulink object Name and Type information that you store on the RM application side:

```
[objName, objType] = slrequirements('getObjLabel', slObjectHandle);
```

To query links for a Simulink object, specified by either numeric handle or SID:

```
linkInfo = slrequirements('getLinks', sLObjectHandle)
linkInfo = slrequirements('getLinks', sigBuilderHandle, m) % Signal Builder group "m"
linkInfo = slrequirements('getLinks', [modelName objSIDs{i}]);
```

`linkInfo` is a MATLAB structure that contains link attributes. See the `rmi` function reference page for more details.

After you retrieve the updated link information from your RM application, populate the fields of `linkData` with the updated values, and propagate the changes to Simulink:

```
slrequirements('setLinks', sLObjectHandle, linkData)
```

For an example MATLAB script implementing synchronization with a Microsoft Excel Workbook, see the following:

```
edit([matlabroot '/toolbox/slrequirements/linktype_examples/slSurrogateInExcel.m'])
```

You can run this MATLAB script on the example model `slvndemo_fuelsys_officereq` to generate the Excel workbook surrogate for the model.

Navigate to Simulink Objects from External Documents

Abstract

Navigate from external documents to Simulink model objects.

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

Provide Unique Object Identifiers

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

Use the `rmiobjnavigate` Function

The `rmiobjnavigate` function identifies the Simulink or Stateflow object, highlights that object, and brings the editor window to the front of the screen. When you navigate to a Simulink model from an external application, invoke this function.

The first time you navigate to an item in a particular model, you might experience a slight delay while the software initializes the communication API and the internal data structures. You do not experience a long delay on subsequent navigation.

Determine the Navigation Command

To create a requirement link for a Simulink or Stateflow object, at the MATLAB prompt, use the following command to find the navigation command, where `obj` is a handle or a uniquely resolved name for the object:

```
[ navCmd, objPath ] = rmi('navCmd', obj);
```

The return values of the `navCmd` method are:

- `navCmd` — A character vector that navigates to the object when evaluated by the MATLAB software.

- `objPath` — A character vector that identifies the model object.

Send `navCmd` to the MATLAB software for evaluation when navigating from the external application to the object `obj` in the Simulink model. Use `objPath` to visually identify the target object in the requirements document.

Use the ActiveX Navigation Control

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

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

Typical Code Sequence for Establishing Navigation Controls

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

To automate the procedure for establishing links:

- 1 Select a Simulink or Stateflow object and an item in the external document.
- 2 Invoke the link creation action either from a Simulink menu or command, or a similar mechanism in the external application.
- 3 Identify the document and current item using the scripting capability of the external tool. Pass this information to the MATLAB software. Create a requirement link on the selected object using the RMI API as follows:
 - a Create an empty link structure using the following command:

```
rmi('createempty')
```
 - b Fill in the link structure fields based on the target location in the requirements document.
 - c Attach the link to the object using the following command:

```
rmi('cat')
```

- 4 Determine the MATLAB navigation command that you must embed in the external tool, using the `navCmd` method:

```
[ navCmd, objPath ] = rmi('navCmd',obj)
```

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

When using ActiveX navigation objects provided by the external tool, set the `MLEvalCmd` property to the `navCmd` and set the `tooltipstring` property to `objPath`.

You define the MATLAB code implementation of this procedure as the `SelectionLinkFcn` function in the link type definition file. The following files in *matlabroot*\toolbox\shared\reqmgt\+linktypes contain examples of how to implement this functionality:

```
linktype_rmi_doors.m  
linktype_rmi_excel.m  
linktype_rmi_html.m  
linktype_rmi_text.m
```


Requirements Information in Generated Code

- “How Requirements Information Is Included in Generated Code” on page 11-2
- “Generate Code for Models with Requirements Links” on page 11-4

How Requirements Information Is Included in Generated Code

After you simulate your model and verify its performance against the requirements, you can generate code from the model for an embedded real-time application. The Embedded Coder software generates code for Embedded Real-Time (ERT) targets.

If the model has any links to requirements, the Embedded Coder software inserts information about the requirements links into the code comments.

For example, if a block has a requirement link, the software generates code for that block. In the code comments for that block, the software inserts:

- Requirement description
- Hyperlink to the requirements document that contains the linked requirement associated with that block

Note:

- You must have a license for Embedded Coder to generate code for an embedded real-time application.
 - If you use an external `.req` file to store your requirement links, to avoid stale comments in generated code, before code generation, you must save any change in your requirement links. For information on how to save, see “Save Requirements Links in External Storage” on page 4-5.
-

Comments for the generated code include requirements descriptions and hyperlinks to the requirements documents in the following locations.

Model Object with Requirement	Location of Code Comments with Requirements Links
Model	In the main header file, <code><model>.h</code>
Nonvirtual subsystem	At the call site for the subsystem
Virtual subsystem	At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem does not have a nonvirtual parent, requirement descriptions appear in the main header file for the model, <code><model>.h</code> .

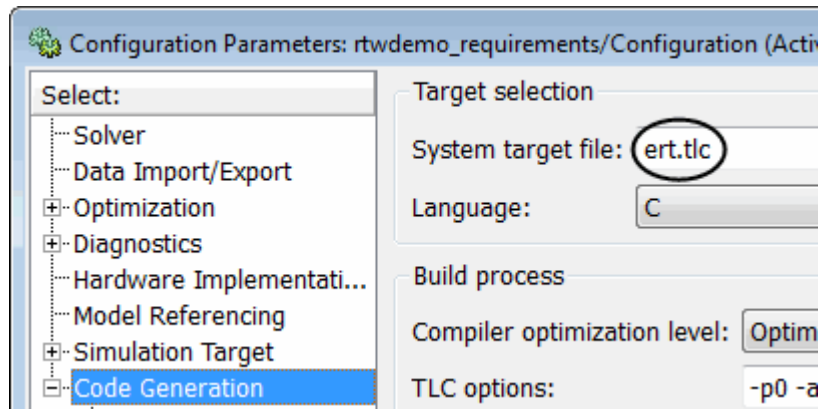
Model Object with Requirement	Location of Code Comments with Requirements Links
Nonsubsystem block	In the generated code for the block
MATLAB code line in MATLAB Function block	In the generated code for the MATLAB code line(s)

Generate Code for Models with Requirements Links

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

- 1 Open the `rtwdemo_requirements` example model.
- 2 Select **Simulation > Model Configuration Parameters**.
- 3 In the **Select** tree of the Configuration Parameters dialog box, select the **Code Generation** node.

The currently configured system target must be an ERT target.



- 4 Under **Code Generation**, select **Comments**.
- 5 In the **Custom comments** section on the right, select the **Requirements in block comments** check box.
- 6 Under **Code Generation**, select **Report**.
- 7 On the **Report** pane, select:
 - **Create code generation report**
 - **Open report automatically**
- 8 Press **Ctrl+B** to build the model.
- 9 In the code-generation report, open `rtwdemo_requirements.c`.
- 10 Scroll to the code for the Pulse Generator block, `clock`. The comments for the code associated with that block include a hyperlink to the requirement linked to that block.


```
rtwdemo_requirements.c 119
rtwdemo_requirements.h 120 /* DiscretePulseGenerator: '<Root>/clock'
rtwdemo_requirements_p 121 *
rtwdemo_requirements_t 122 * Block requirements for '<Root>/clock':
123 * 1. Clock period shall be consistent with chirp tolerance
124 */
```

- 11 Click the link **Clock period shall be consistent with chirp tolerance** to open the HTML requirements document to the associated requirement.

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

Model Component Testing

Overview of Component Verification

- “Component Verification” on page 12-2
- “Basic Approach to Component Verification” on page 12-4
- “Functions for Component Verification” on page 12-8

Component Verification

In this section...
“Component Verification Approaches” on page 12-2
“Simulink Verification and Validation Tools for Component Verification” on page 12-2

Component Verification Approaches

Abstract

Component verification for validating your model

Component verification allows you to test a design component in your model using one of two approaches:

- Within the context of the model that contains the component — Using systematic simulation of closed-loop controllers requires that you verify components within a control system model. Doing so allows you to test the control algorithms with your model. This approach is called *system analysis*.
- As standalone components — For a high level of confidence in the component algorithm, verify the component in isolation from the rest of the system. This approach is called *component analysis*.

Verifying standalone components provides several advantages:

- You can use the analysis to focus on portions of the design that you cannot test because of the physical limitations of the system being controlled.
- You can use this approach for open-loop simulations to test the plant model without feedback control.
- You can use this approach when the model is not yet available or when you need to simulate a control system model in accelerated mode for performance reasons.

Simulink Verification and Validation Tools for Component Verification

By isolating the component to verify and using tools that the Simulink Verification and Validation software provides, you create test cases that allow you to expand the scope of the testing for large models. This expanded testing helps you accomplish the following:

- Achieve 100% model coverage — If certain model components do not record 100% coverage, the top-level model cannot achieve 100% coverage. By verifying these components individually, you can create test cases that fully specify the component interface, allowing the component to record 100% coverage.
- Debug the component — To verify that each model component satisfies the specified design requirements, you can create test cases that verify that specific components perform as designed.
- Test the robustness of the component — To verify that a component handles unexpected inputs and calculations properly, you can create test cases that generate data. Then, test the error-handling capabilities in the component.

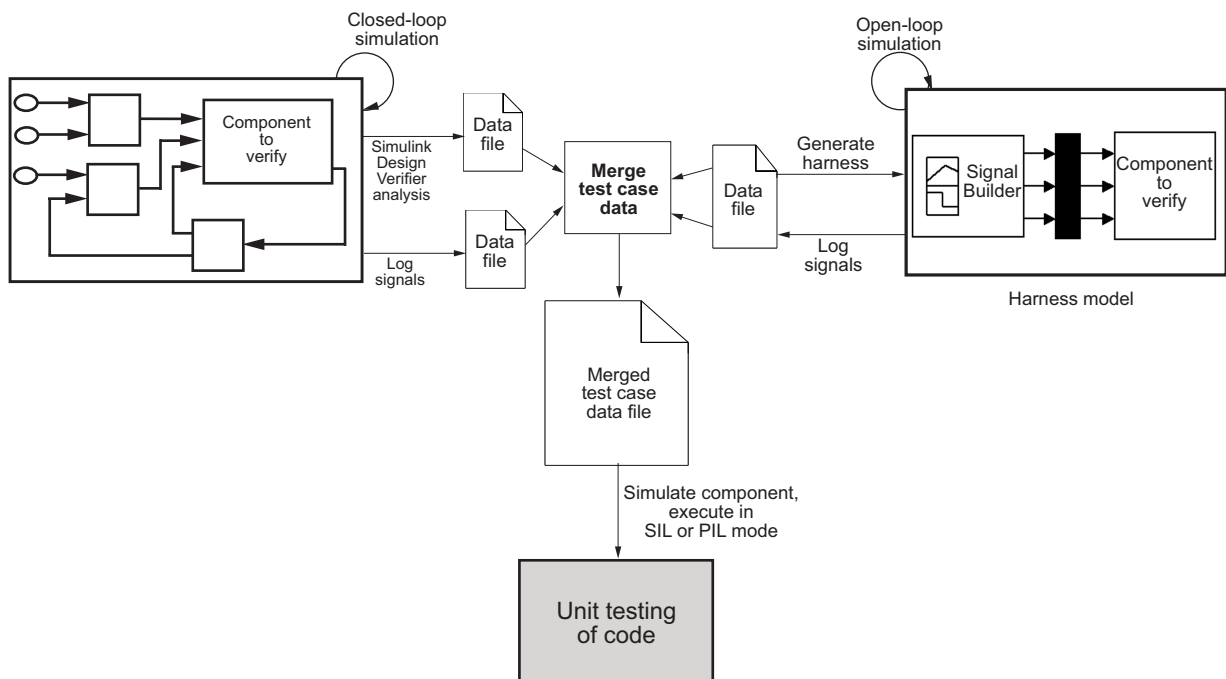
Basic Approach to Component Verification

In this section...

- “Workflow for Component Verification” on page 12-4
- “Verify a Component Independently of the Container Model” on page 12-6
- “Verify a Model Block in the Context of the Container Model” on page 12-7

Workflow for Component Verification

The following graphic illustrates the common workflow for component verification.



This graphic illustrates the two approaches for component verification, described in “Component Verification” on page 12-2:

- 1 Choose your approach for component verification:

- For closed-loop simulations, verify a component within the context of its container model by logging the signals to that component and storing them in a data file. If those signals do not constitute a complete test suite, generate a harness model and add or modify the test cases in the Signal Builder.
 - For open-loop simulations, verify a component independently of the container model by extracting the component from its container model and creating a harness model for the extracted component. Add or modify test cases in the Signal Builder and log the signals to the component in the harness model.
- 2 Prepare component for verification.
 - 3 Create and log test cases. If desired, merge the test case data into a single data file.

The data file contains the test case data for simulating the component. If you cannot achieve the desired results with a certain set of test cases, add new test cases or modify existing test cases in the data file, and merging them into a single data file.

Continue adding or modifying test cases until you achieve a test suite that satisfies the goals of your analysis.

- 4 Execute the test cases in Software-in-the-Loop or Processor-in-the-Loop mode.
- 5 After you have a complete test suite, you can:
 - Simulate the model and execute the test cases to:
 - Record coverage.
 - Record output values to make sure that you get the expected results.
 - Invoke the Code Generation Verification (CGV) API to execute the generated code for the model that contains the component in simulation, Software-in-the-Loop (SIL), or Processor-in-the-Loop (PIL) mode.

Note: To execute a model in different modes of execution, you use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see “Programmatic Code Generation Verification”.

The next sections describe the steps for component verification in more detail:

- “Verify a Component Independently of the Container Model” on page 12-6
- “Verify a Model Block in the Context of the Container Model” on page 12-7

Verify a Component Independently of the Container Model

Use component analysis to verify:

- Model blocks
- Atomic subsystems
- Stateflow atomic subcharts

The recommended steps for verifying a component independently of the container model:

- 1** Depending on the type of component, take one of the following actions:
 - Model blocks — Open the referenced model.
 - Atomic subsystems — Extract the contents of the subsystem into its own Simulink model.
 - Atomic subcharts — Extract the contents of the Stateflow atomic subchart into its own Simulink model.
- 2** Create a harness model for:
 - The referenced model
 - The extracted model that contains the contents of the atomic subsystem or atomic subchart
- 3** Add or modify test cases in the Signal Builder in the harness model.
- 4** Log the input signals from the Signal Builder to the test unit.
- 5** Repeat steps 3 and 4 until you are satisfied with the test suite.
- 6** Merge the test case data into a single file.
- 7** Depending on your goals, take one of the following actions:
 - Execute the test cases to:
 - Record coverage.
 - Record output values and make sure that they equal the expected values.
 - Invoke the Code Generation Verification (CGV) API to execute the test cases in Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL) mode on the generated code for the model that contains the component.

If the test cases do not achieve the desired results, repeat steps 3 through 5.

Verify a Model Block in the Context of the Container Model

Use system analysis to verify a Model block in the context of the block's container model. Use this technique when you analyze a closed-loop controller.

The recommended steps for system analysis:

- 1** Log the input signals to the component by simulating the container model.

or

Analyze the model using the Simulink Design Verifier software.

- 2** If you want to add test cases to your test suite or modify existing test cases, create a harness model using the logged signals.
- 3** Add or modify test cases in the Signal Builder in the harness model.
- 4** Log the input signals from the Signal Builder to the test unit.
- 5** Repeat steps 3 and 4 until you are satisfied with the test suite.
- 6** Merge the test case data into a single file.
- 7** Depending on your goals, do one of the following:
 - Execute the test cases to:
 - Record coverage.
 - Record output values and make sure that they equal the expected values.
 - Invoke the Code Generation Verification (CGV) API to execute the test cases in Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL) mode on the generated code for the model.

If the test cases do not achieve the desired results, repeat steps 3 through 5.

Functions for Component Verification

The Simulink Verification and Validation software provides several functions that facilitate the tasks associated with component verification.

Task	Function
Simulate a Simulink model and log input signals to a Model block in the model. If you modify the test cases in the Signal Builder harness model, use this approach for logging input signals to the harness model itself.	<code>slvnlvlogsignals</code>
Create a harness model for a component, using logged input signals if specified, or using the default signals. A harness model contains four Simulink blocks as described in “Prepare the Component for Verification” on page 13-4 in “Verify Generated Code for a Component” on page 13-2.	<code>slvnlvmakeharness</code>
Merge test case data into a single data structure for batch execution or harness generation.	<code>slvnlvmergedata</code>
Merge test cases from several harness models into a single harness model.	<code>slvnlvmergeharness</code>
Extract an atomic subsystem or atomic subchart into a new model.	<code>slvnlvextract</code>
Simulate a model, executing the specified test cases to record model coverage and output values.	<code>slvnlvruntest</code>
Invoke the Code Generation Verification (CGV) API, and execute the specified test cases on the generated code for the model.	<code>slvnlvruncgvtest</code>

Component verification functions do not support the following Simulink software features:

- Variable-step solvers for `slvnlvruntest`
- Component interfaces that contain:
 - Complex signals
 - Variable-size signals

- Array of buses
- Multiword fixed-point data types

Verifying Generated Code for a Component

Verify Generated Code for a Component

In this section...
“About the Example Model” on page 13-2
“Prepare the Component for Verification” on page 13-4
“Create and Log Test Cases” on page 13-6
“Merge Test Case Data” on page 13-7
“Record Coverage for Component” on page 13-7
“Execute Component in Simulation Mode” on page 13-8
“Execute Component in SIL Mode” on page 13-9

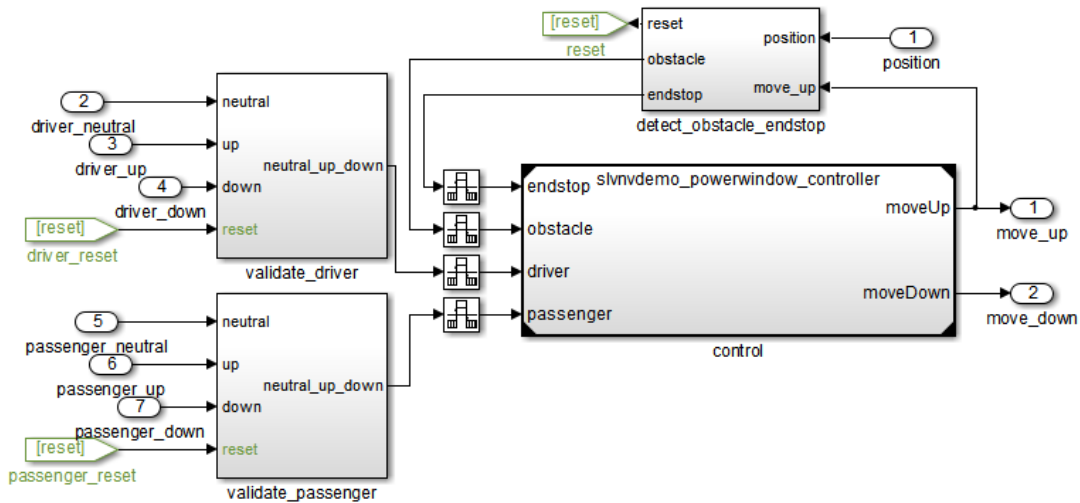
About the Example Model

This example uses the `slvndemo_powerwindow` example model to show how to verify a component in the context of the model that contains that component. As you work through this example, you use the Simulink Verification and Validation component verification functions to create test cases and measure coverage for a referenced model. In addition, you execute the referenced model in both simulation mode and software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode using the Code Generation Verification (CGV) API and then compare the results.

Note: You must have the following product licenses to run this example:

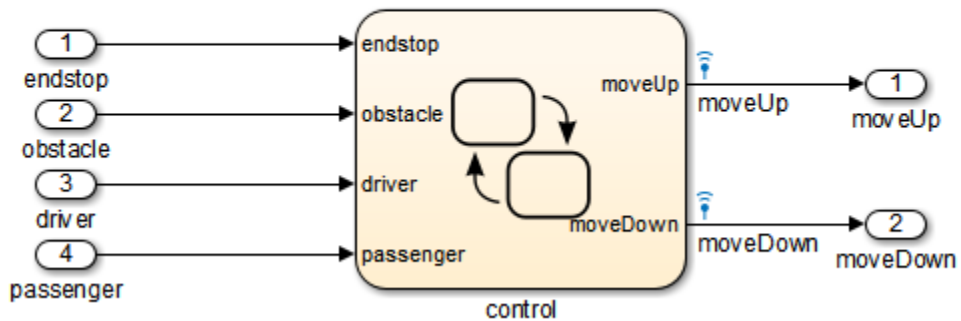
- Stateflow
 - Embedded Coder
 - Simulink Coder™
-

The component you verify is a Model block named `control`. This component resides inside the `power_window_control_system` subsystem in the top level of the `slvndemo_powerwindow` model.

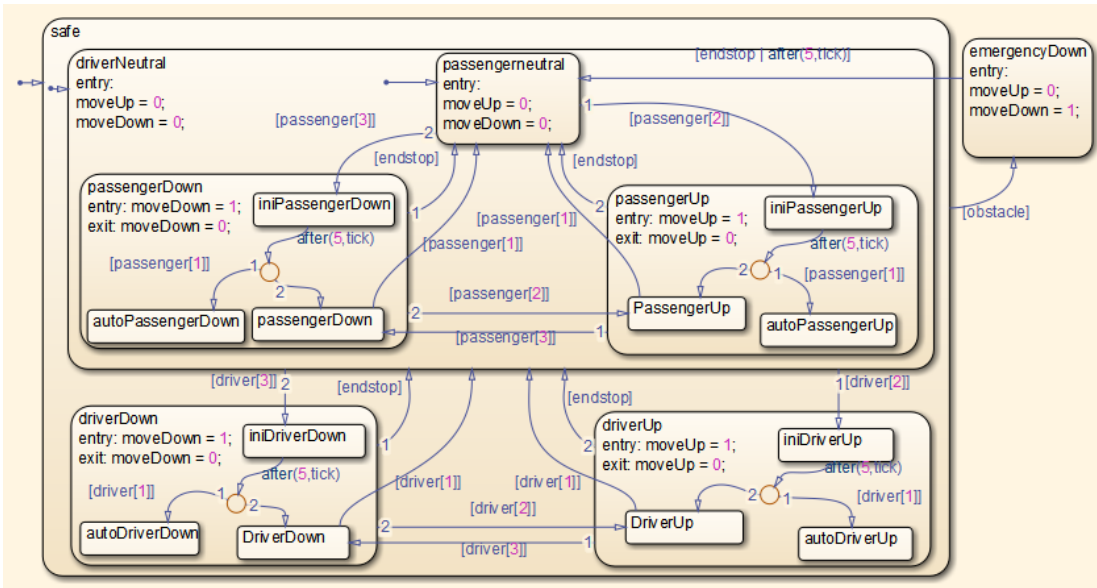


The Model block references the slvndemo_powerwindow_controller model.

Simulink Verification and Validation Power Window Controller



The referenced model contains a Stateflow chart `control`, which implements the logic for the power window controller.



Prepare the Component for Verification

To verify the referenced model `slvndemo_powerwindow_controller`, begin by creating a harness model containing input signals that simulate the controller in the plant model:

- 1 Open the `slvndemo_powerwindow` example model:

```
slvndemo_powerwindow
```

The `slvndemo_powerwindow` example model opens, showing the `power_window_control_system` subsystem.

- 2 The Model block named `control` in the `power_window_control_system` subsystem refers to the component you verify during this example — `slvndemo_powerwindow_controller`. Load the referenced model:

```
load_system('slvndemo_powerwindow_controller');
```

- 3 Simulate the Model block that references `slvndemo_powerwindow_controller` and log the input signals to the Model block:

```
loggedSignalsPlant = ...
```

```
slvnlvlogsignals(...  
    'slvnlvdemo_powerwindow/power_window_control_system/control');
```

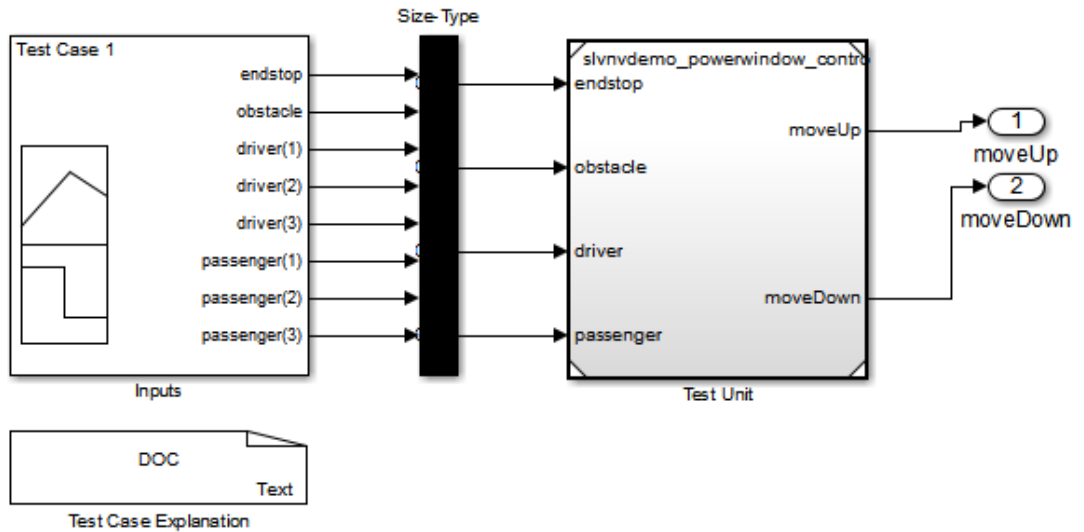
`slvnlvlogsignals` stores the logged signals in `loggedSignalsPlant`.

4 Generate an empty harness model for adding test cases.

```
harnessModelFilePath = ...  
    slvnlvmakeharness('slvnlvdemo_powerwindow_controller');
```

`slvnlvmakeharness` creates a harness model named `slvnlvdemo_powerwindow_controller_harness`. The harness model includes:

- **Test Unit** — A Model block that references the `slvnlvdemo_powerwindow_controller` model.
- **Inputs** — A Signal Builder block that contains one test case. That test case specifies the values of the input signals logged when the model `slvnlvdemo_powerwindow` was simulated.
- **Test Case Explanation** — A DocBlock block that describes the test case.
- **Size-Type** — A Subsystem block that transmits signals from the Inputs block to the Test Unit block. The output signals from this block match the input signals for the Model block you are verifying.
- **moveUp** and **moveDown** — Two output ports that match the output ports from the Model block.



- 5 Get the name of the harness model:

```
[~,harnessModel] = fileparts(harnessModelFilePath);
```

- 6 Leave all models open for the next steps.

Next, create a test case that tests values for input signals to the component.

Create and Log Test Cases

Add a test case for your component to help you get closer to 100% coverage.

Add a test case to the Signal Builder block in the harness model using the `signalbuilder` function. The test case specifies input signals to the component.

- 1 Load the file containing the test case data into the MATLAB workspace:

```
load('slvndemo_powerwindow_controller_newtestcase.mat');
```

The workspace variables `newTestData` and `newTestTime` contain the test case data.

- 2 Add the test case to the Signal Builder block in the harness model.

```

signalBuilderBlock = slvndemo_signalbuilder_block(harnessModel);
signalbuilder(signalBuilderBlock,'Append',...
    newTestTime, newTestData,...
    {'endstop','obstacle','driver(1)','driver(2)','driver(3)',...
    'passenger(1)','passenger(2)','passenger(3)'},'New Test Case');

```

- 3 Simulate the harness model with both test cases, then log the signals to the referenced model and save the results:

```
loggedSignalsHarness = slvnlvsignals(harnessModel);
```

Next, record coverage for the `slvnv_powerwindow_controller` model.

Merge Test Case Data

You have two sets of test case data:

- `loggedSignalsPlant` — Logged signals to the Model block control
- `loggedSignalsHarness` — Logged signals to the test cases you added to the empty harness

To simulate all the test data simultaneously, merge the two data files into a single file:

- 1 Combine the test case data:

```
mergedTestCases = slvnmmergedata(loggedSignalsPlant,...
    loggedSignalsHarness);
```

- 2 View the merged data:

```
disp(mergedTestCases);
```

Next, simulate the referenced model with the merged data and get coverage for the referenced model, `slvnv_powerwindow_controller`.

Record Coverage for Component

To record coverage for the `slvnv_powerwindow_controller` model:

- 1 Create a default options object, required by the `slvnlvruntest` function:

```
runopts = slvnlvruntestopts;
```

- 2 Specify to simulate the model and record coverage:

```
runopts.coverageEnabled = true;
```

- 3 Simulate the model using the logged input signals:

```
[-, covdata] = slvnvrntest('slvnvdemo_powerwindow_controller',...  
    mergedTestCases,runopts);
```

- 4 Display the HTML coverage report:

```
cvhtml('Coverage with Test Cases from Harness', covdata);
```

The `slvnv_powerwindow_controller` model achieved:

- Decision coverage: 44%
- Condition coverage: 45%
- MCDC coverage: 10%

For more information about decision coverage, condition coverage, and MCDC coverage, see “Types of Model Coverage” on page 16-3.

To increase the test coverage, modify or add test cases using the Signal Builder block in the harness model, as described in “Create and Log Test Cases” on page 13-6.

Execute Component in Simulation Mode

To verify that the generated code produces the same results as simulating the model, use the Code Generation Verification (CGV) API methods. When you perform this procedure, the simulation compiles and executes the model code using the merged test cases:

- 1 Create a default options object for `slvnvruncgvtest`:

```
runcgvopts = slvnvrntestopts('cgv');
```

- 2 Specify to execute the model in simulation mode:

```
runcgvopts.cgvConn = 'sim';
```

- 3 Execute the `slvnv_powerwindow_controller` model using the two test cases and the `runopts` object:

```
cgvSim = slvnvruncgvtest('slvnvdemo_powerwindow_controller', ...  
    mergedTestCases, runcgvopts);
```

These steps save the results in the workspace variable `cgvSim`.

Next, execute the same model with the same test cases in software-in-the-loop (SIL) mode and compare the results from both simulations.

For more information about Normal simulation mode, see “Execute the Model”.

Execute Component in SIL Mode

When you execute a model in software-in-the-loop (SIL) mode, the simulation compiles and executes the generated code on your host computer.

To execute a model in SIL mode, you must have an Embedded Coder license.

In this section, you execute the `slvnvdemo_powerwindow_controller` model in SIL mode and compare the results to the previous section, where you executed the model in simulation mode:

- 1 Specify to execute the model in SIL mode:

```
runcgvopts.cgvConn = 'sil';
```

- 2 Execute the `slvnv_powerwindow_controller` model using the merged test cases and the `runopts` object:

```
cgvSil = slvnvruncgvtest('slvnvdemo_powerwindow_controller', ...
    mergedTestCases, runcgvopts);
```

The workspace variable `cgvSil` contains the results of the SIL mode execution.

- 3 Display a comparison of the results in `cgvSil` to the results in `cgvSim` (the results from the simulation mode execution). Use the `compare` (`cgv.CGV`) method to compare the results from the two simulations:

```
for i=1:length(loggedSignalsHarness.TestCases)
    simout = cgvSim.getOutputData(i);
    silout = cgvSil.getOutputData(i);
    [matchNames, ~, mismatchNames, ~] = ...
        cgv.CGV.compare(simout, silout);
    fprintf('\nTest Case(%d): %d Signals match, ...
        %d Signals mismatch', i, length(matchNames), ...
        length(mismatchNames));
end
```

For more information about software-in-the-loop (SIL) simulations, see “What Are SIL and PIL Simulations?”

Signal Monitoring with Model Verification Blocks

Using Model Verification Blocks

- “Model Verification Blocks and the Verification Manager” on page 14-2
- “Use Check Static Lower Bound Block to Check for Out-of-Bounds Signal” on page 14-3
- “Linear System Modeling Blocks in Simulink Control Design” on page 14-6

Model Verification Blocks and the Verification Manager

Simulink Model Verification library blocks monitor time-domain signals in your model during simulation, according to the specifications that you assign to the blocks.

Note: To see a complete description of all Simulink model verification blocks, see the “Model Verification” category in the Simulink documentation.

You set a verification block to assert when its signal leaves the limit or range that you specify. During simulation, when the signal crosses the limit, the verification block can:

- Stop the simulation and bring immediate focus to that part of the model.
- Report the limit encounter with a logical signal output of its own. If the simulation does not encounter the limit, the signal output is true. If the simulation encounters the limit, the signal output is false.

The Verification Manager is a graphical interface in the Signal Builder dialog box. Using this tool, you can manage all the Model Verification blocks in your model from a central location.

If you have Simulink Control Design™ software, you can also monitor frequency-domain characteristics such as:

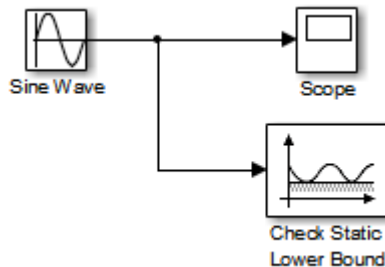
- Gain and phase margins
- Peak magnitude

Note: For more information about the Simulink Control Design model verification blocks, see “Model Verification” in the Simulink Control Design documentation.

Use Check Static Lower Bound Block to Check for Out-of-Bounds Signal

The following example uses a `Check Static Lower Bound` block to stop simulation when a signal from a `Sine Wave` block crosses its lower bound limit.

- 1 Attach a `Check Static Lower Bound` block to the signal from a `Sine Wave` block.



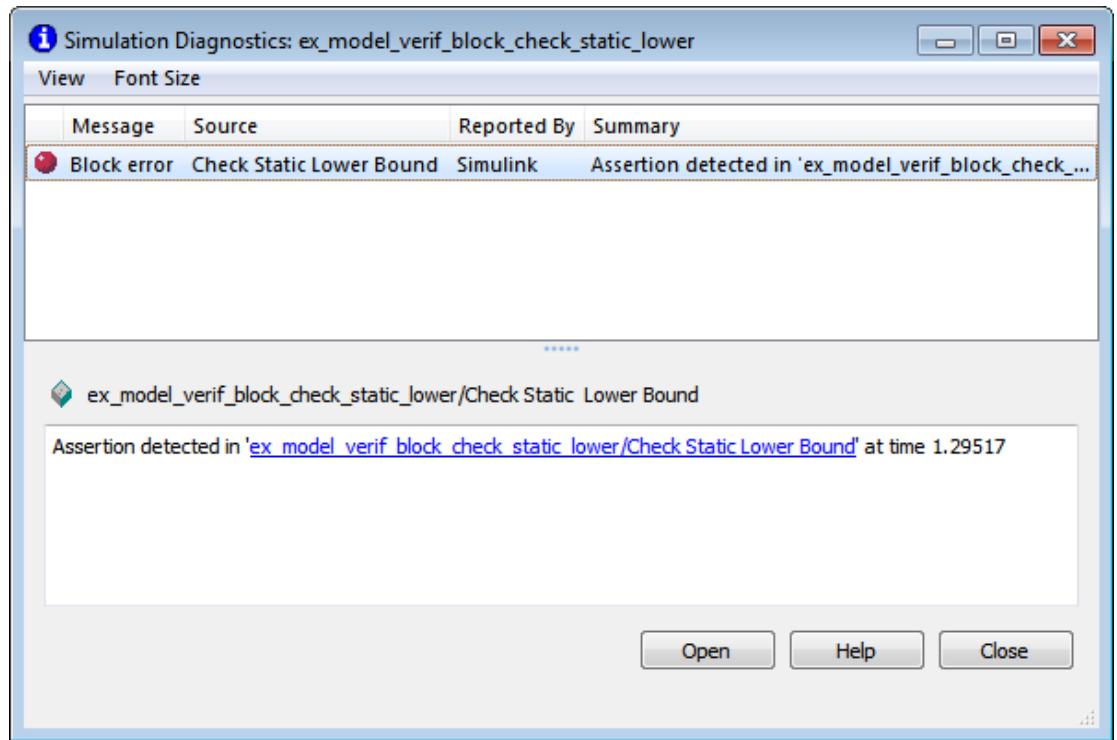
- 2 Set the Simulation stop time to 2 seconds.
- 3 Double-click the `Sine Wave` block and set the following parameters:
 - Set the **Amplitude** to 1.
 - Set the **Frequency** to π radians per second.
- 4 Double-click the `Check Static Lower Bound` block and set the **Lower bound** parameter to `-0.8`.

Enable assertion is the default. This parameter enables a verification block for an assertion. You set the `Check Static Lower Bound` block to detect a signal value of `-0.8` or lower. If the signal value reaches that value or falls below it, the simulation stops.

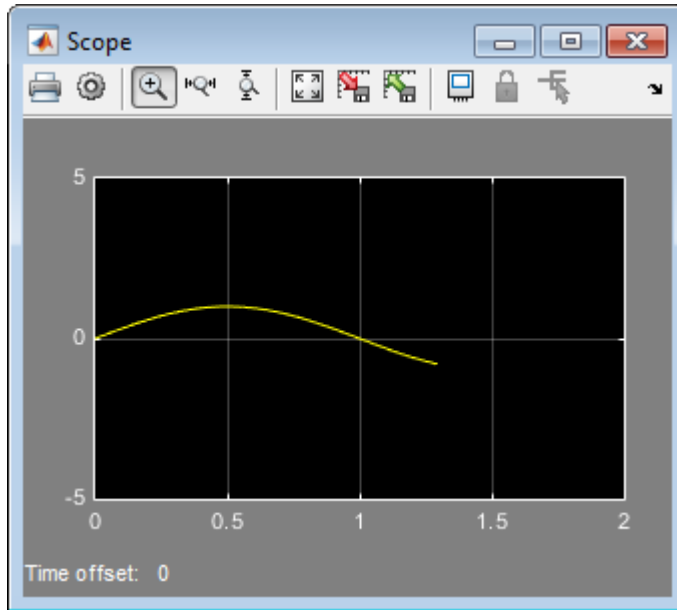
- 5 Run the simulation.

The model stops simulating after 1.295 seconds, when the output is `-0.8`. The software highlights the `Check Static Lower Bound` block.

When the simulation stops, you see the following diagnostic message.

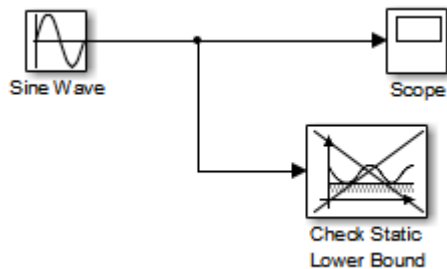


- 6 To verify the signal value, double-click the Scope block.



- 7 To disable the Check Static Lower Bound block from asserting its limit, clear the **Enable assertion** check box.

The block is crossed out in the model, as shown.



Linear System Modeling Blocks in Simulink Control Design

If you have Simulink Control Design software, you can:

- Specify bounds on linear system characteristics.
- Check that the bounds are satisfied during simulation.

For example, you can check if the linearized behavior of your model satisfied upper and lower magnitude bounds on a Bode plot or gain and phase margins. For more information, see the individual block reference pages in the “Model Verification” category of the Simulink Control Design documentation.

Constructing Simulation Tests Using the Verification Manager

- “What Is the Verification Manager?” on page 15-2
- “Construct Simulation Tests Using the Verification Manager” on page 15-3

What Is the Verification Manager?

The Verification Manager is a graphical interface in the Signal Builder dialog box. Using this tool, you can manage all the Model Verification blocks in your model from a central location.

Construct Simulation Tests Using the Verification Manager

In this section...

“View Model Verification Blocks” on page 15-3

“Enable and Disable Model Verification Blocks in a Model” on page 15-9

“Enable and Disable Model Verification Blocks in a Subsystem” on page 15-12

“Use Check Static Lower Bound Block to Check for Out-of-Bounds Signal” on page 15-16

“Link Test Cases to Requirements Documents Using the Verification Manager” on page 15-19

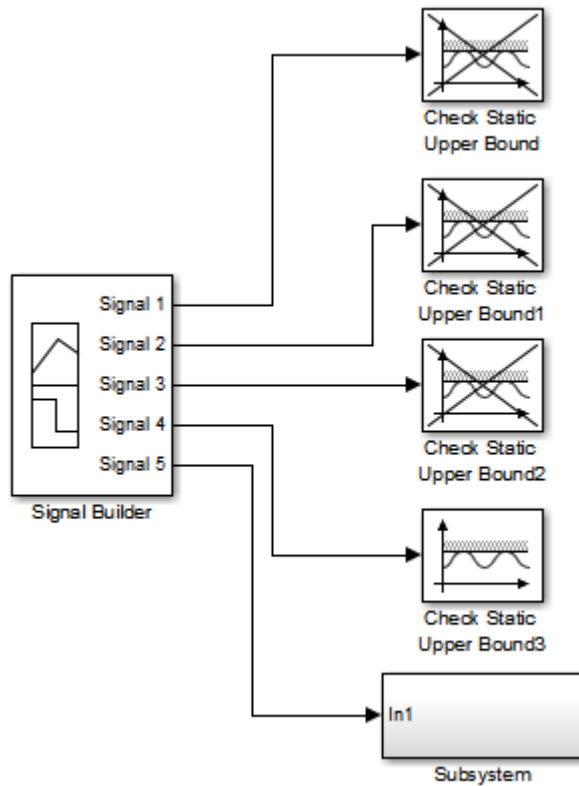
View Model Verification Blocks

Abstract

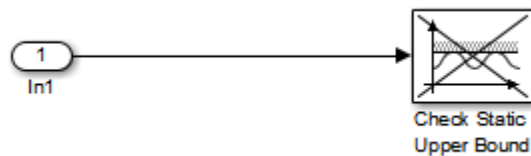
Create a Simulink model to examine the Verification Manager.

Create a Simulink model that you can use to examine the Verification Manager.

1 In the Simulink software, create the following example model.



In the example model, the contents of the subsystem are as follows.



- a In the Signal Builder block, create a signal group with five signals in the group.

- b** Make two copies of the signal group, so that you have three signal groups: **Group 1, Group 2, Group 3.**

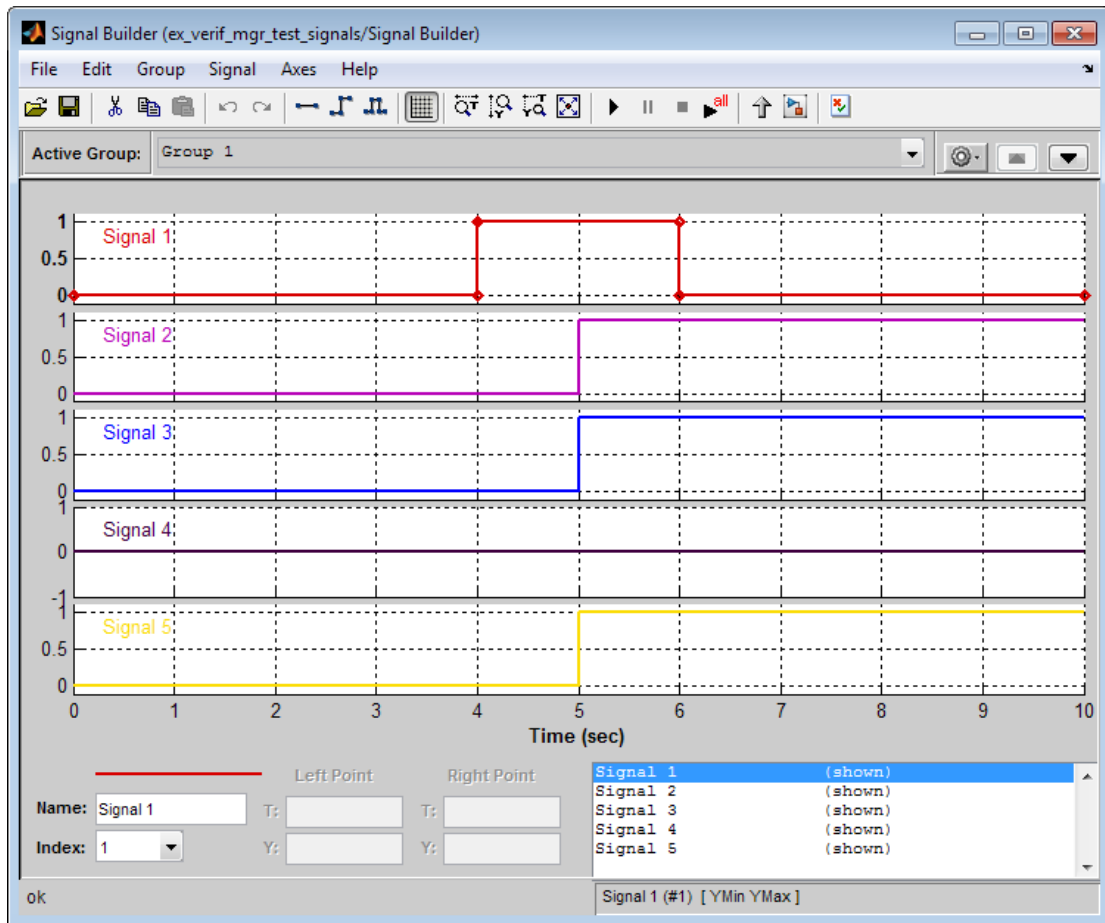
Note: A Signal Builder block provides test signals for an entire model from one location. This model contains a Signal Builder block that feeds five test signals to the Model Verification blocks. The model sends the first four signals directly to Check Static Upper Bound blocks. The model sends the fifth signal to a subsystem that contains a Check Static Upper Bound block.

For more information on the Signal Builder block, see “Signal Groups” in the Simulink documentation.

- c** To set each Check Static Upper Bound verification block to assert for an upper bound of 1, set the **Upper bound** parameter to 1.
- d** For the following blocks, disable the assertion by clearing the **Enable assertion** parameter:
- Check Static Upper Bound
 - Check Static Upper Bound1
 - Check Static Upper Bound2
 - Check Static Upper Bound in the subsystem

These blocks are crossed out in the model.

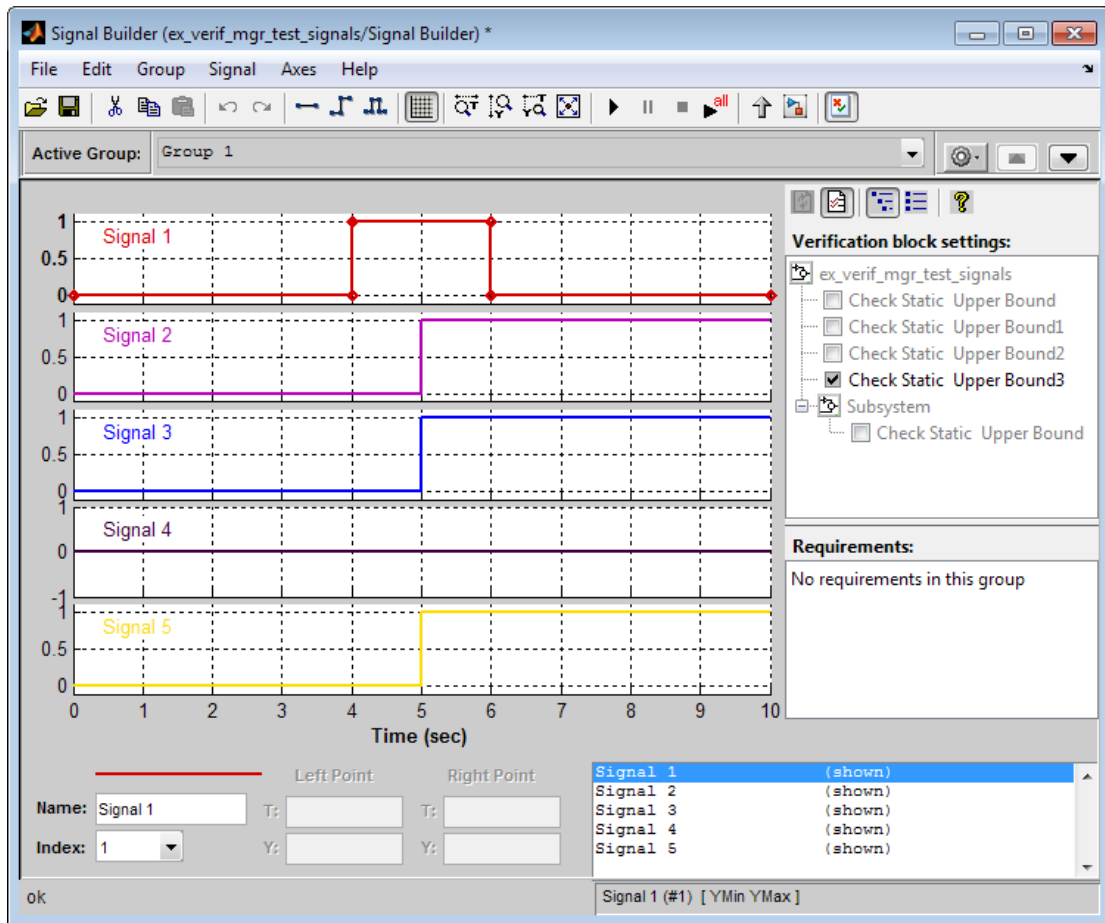
- e** To enable the Check Static Upper Bound3 block, select the **Enable assertion** parameter.
- 2** Save this model and name it `ex_verif_mgr_test_signals`.
- 3** To open the model Signal Builder dialog box, double-click the Signal Builder block. The signals in the first group (**Group 1** in this example) are displayed.



- 4 On the Signal Builder dialog box toolbar, select the Show Verification Settings tool



The **Verification block settings** pane and the **Requirements** pane are displayed.



The **Verification block settings** pane lists all Model Verification blocks in the model, grouped by subsystem. If you right-click in this pane, you can select one of three options for viewing Model Verification blocks in this window:



- **Display > Tree format** — If enabled, lists the blocks as they appear in the model hierarchy.
- **Display > Overridden blocks only** — If enabled, lists only the blocks that have been disabled.

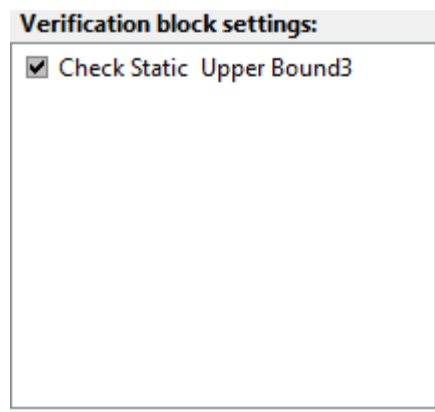
- **Display > Active blocks only** — If enabled, lists only the blocks that are enabled.


Note: If both **Overridden blocks only** and **Active blocks only** are enabled, no Model Verification blocks appear. If both **Overridden blocks only** and **Active blocks only** are disabled, all Model Verification blocks appear.

In this example, the **Verification block settings** pane displays five Check Static Upper Bound blocks. Four are in the top level of the model, and one is in a subsystem.

The **Requirements** pane lists the requirements document links for the current signal group. For details on adding requirement document links in the Signal Builder dialog box, see “Link Test Cases to Requirements Documents Using the Verification Manager” on page 15-19.

- 5 For this example, select  to close the **Requirements** pane.
- 6 To display only the enabled Model Verification blocks for the current signal group, in the **Verification block settings** toolbar, select the List Enabled Verifications tool  .



- 7 To redisplay all Model Verification blocks for the current group, click the Show Verification Block Hierarchy tool .

Enable and Disable Model Verification Blocks in a Model

Use the Verification Manager to enable and disable individual Model Verification blocks in signal groups. To open the Verification Manager in the Signal Builder dialog box, click

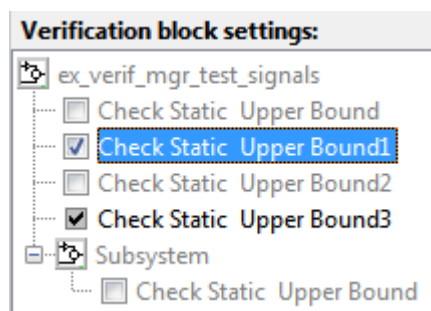


The **Verification block settings** pane lists the Model Verification blocks in the model. Each verification block has a status node that indicates whether its assertion is enabled or disabled. Each verification block's status node also indicates whether the enabled or disabled setting applies universally or to the active group. The following table describes the different types of status nodes.

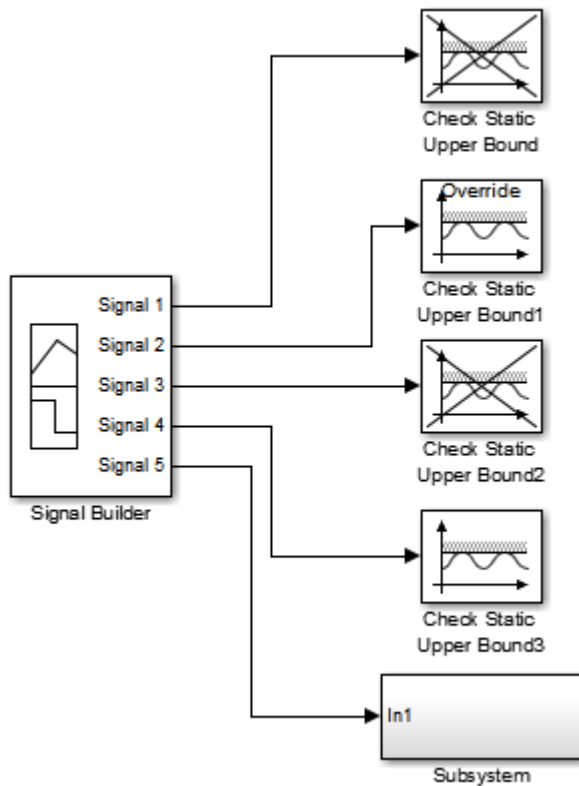
Node	Status
<input type="checkbox"/>	Verification block is disabled for this group. Click to enable for the current group.
<input checked="" type="checkbox"/>	Verification block is enabled for the current group. Click to disable for the current group.
<input checked="" type="checkbox"/>	Verification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

Use the Verification Manager to enable or disable model verification blocks in the `ex_verif_mgr_test_signals` model that you created in “View Model Verification Blocks” on page 15-3.

- 1 In the Verification Manager, click the empty check box next to the Check Static Upper Bound1 node to enable that node for the current active group (**Group 1**).

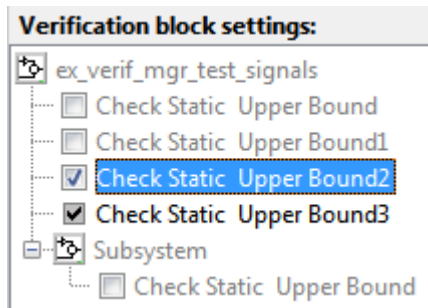


In the **Verification block settings** pane, when you enable a disabled block, you see the following change in how the block is displayed in the model.

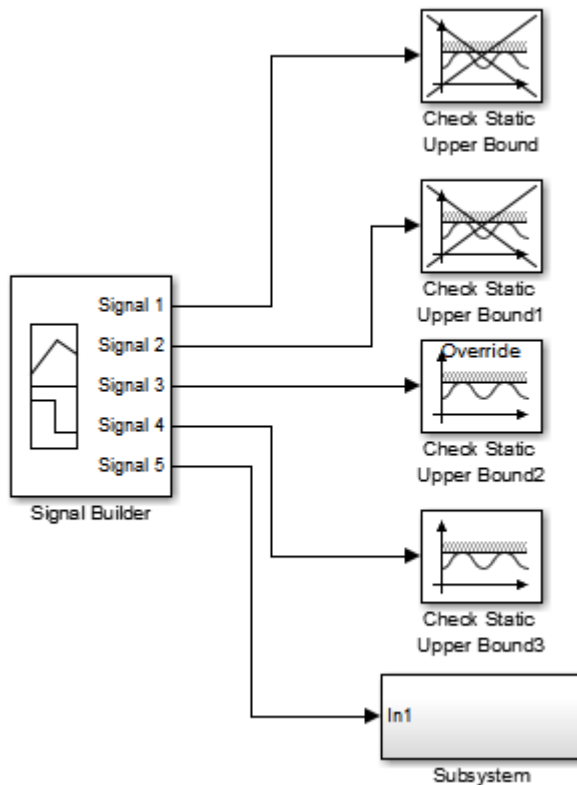


Because you enabled the Check Static Upper Bound1 block in the current group, an **Override** label is applied to the block and it is no longer crossed out.

- 2 In the Signal Builder, from the **Active Group** list, select **Group 2**.
- 3 Select the empty check box next to the Check Static Upper Bound2 node to enable that block for the current group (**Group 2**).




The Check Static Upper Bound2 block is no longer crossed out, indicating that the block is enabled for the current group. Check Static Upper Bound1, however, is crossed out because it is enabled in a different group.



- 4 Save the model with these changes.

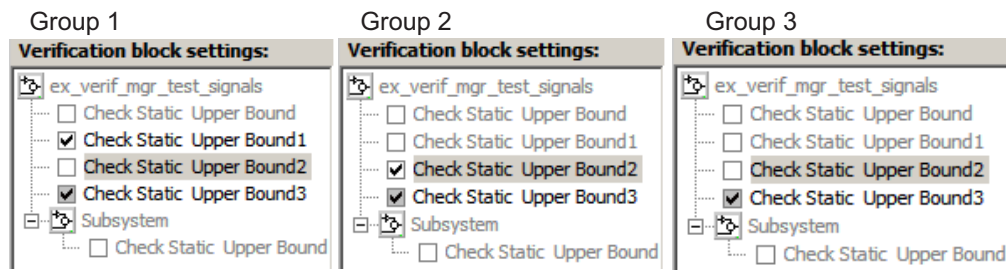
Enable and Disable Model Verification Blocks in a Subsystem

If you have a lot of verification blocks, it is tedious to enable and disable blocks individually. Using the Verification Manager, you can enable and disable blocks from context menu options. Depending on the status of the node, you have the following options.

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

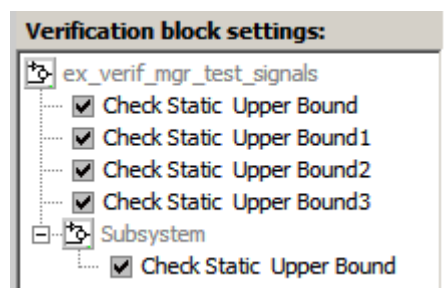
Node Status	Context Menu Options
	<ul style="list-style-type: none"> • Contents group enable • Contents group disable
<input checked="" type="checkbox"/>	<ul style="list-style-type: none"> • Block enable by group
<input type="checkbox"/>	<ul style="list-style-type: none"> • Block enable for all groups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

For example, assume that you define the following groups in the Verification Manager for a model with five Model Verification blocks.



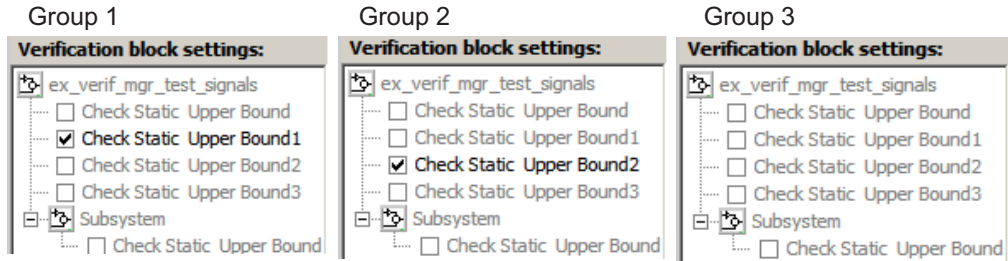
- 1 In the Verification Manager window, right-click the `ex_verif_mgr_test_signals` node and select **Contents enable for all groups**.

This option enables all verification blocks, for all test groups, in all subsystems; the settings for all groups look as follows:



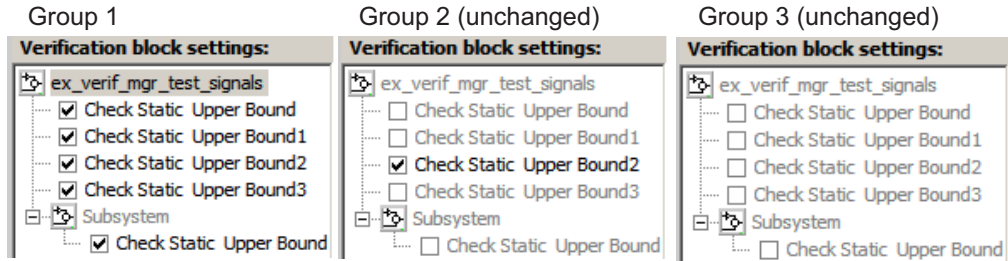
- 2 Right-click `ex_verif_mgr_test_signals` and select **Contents enable by group**.

This option restores the individually enabled/disabled settings for each verification block in each group.



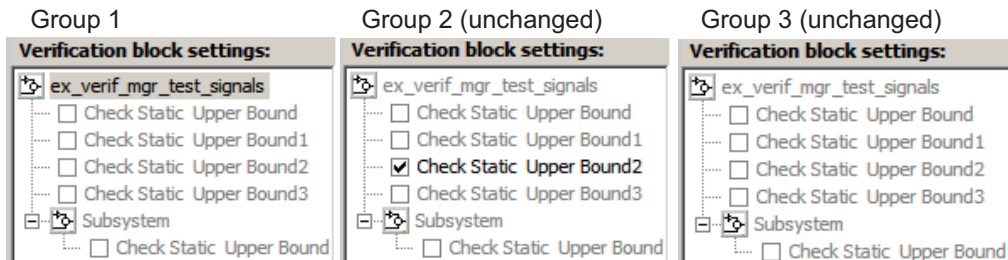
- From the **Active Group** list, select **Group 1**. Right-click `ex_verif_mgr_test_signals`, and select **Contents group enable**.

This option individually enables all contained blocks for only **Group 1**.



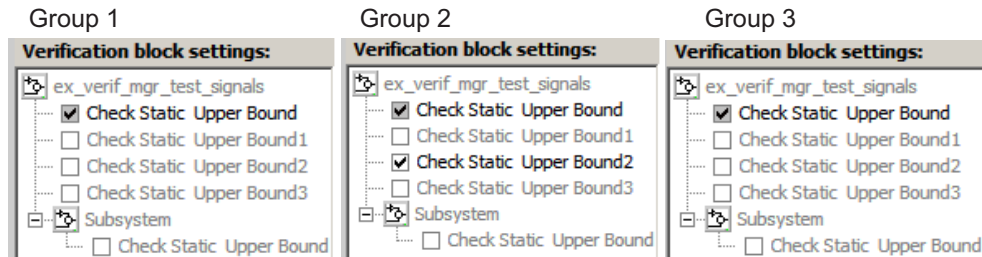
- From the **Active Group** list, select **Group 1**. Right-click `ex_verif_mgr_test_signals` and select **Contents group disable**.

This option individually disables all contained blocks for only **Group 1**.



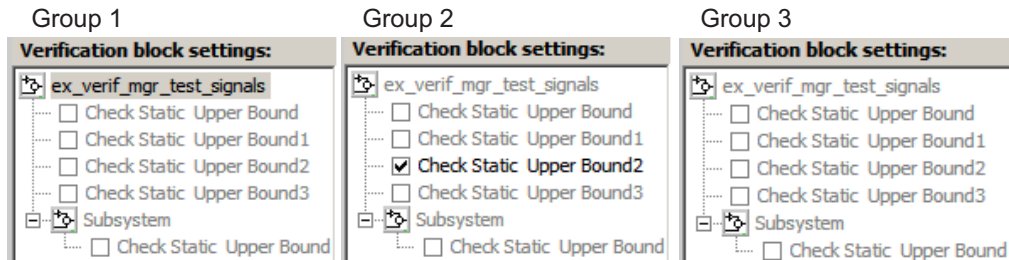
- From the **Active Group** list, select Group 1. Right-click the Check Static Upper Bound node, and select **Block enable for all groups**.

This option enables the Check Static Upper Bound block for all groups.



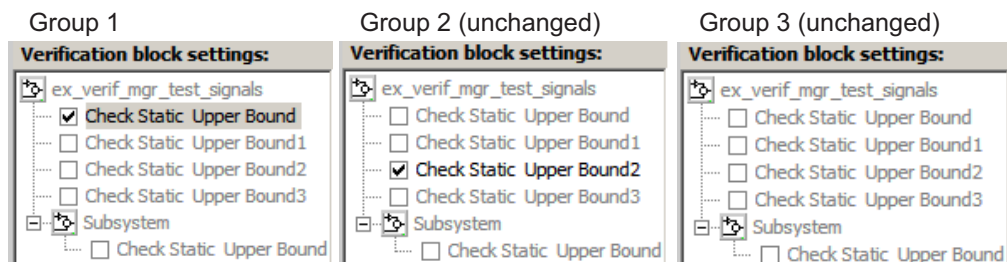
- From the **Active Group** list, select Group 1. Right-click the Check Static Upper Bound node, and select **Block enable by group**.

This option restores the individually enabled/disabled state to this block for all groups. The **Block enable by group** option lets you enable or disable this node individually for each group.



- From the **Active Group** list, select Group 1. Right-click the Check Static Upper Bound node, and select **Block group enable**.

This option enables the Check Static Upper Bound block for this group only.

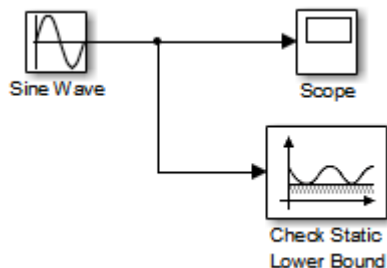


Selecting **Block group disable** disables the specified block for this group only.

Use Check Static Lower Bound Block to Check for Out-of-Bounds Signal

The following example uses a **Check Static Lower Bound** block to stop simulation when a signal from a **Sine Wave** block crosses its lower bound limit.

- 1 Attach a **Check Static Lower Bound** block to the signal from a **Sine Wave** block.



- 2 Set the Simulation stop time to 2 seconds.
- 3 Double-click the **Sine Wave** block and set the following parameters:
 - Set the **Amplitude** to 1.
 - Set the **Frequency** to π radians per second.
- 4 Double-click the **Check Static Lower Bound** block and set the **Lower bound** parameter to -0.8.

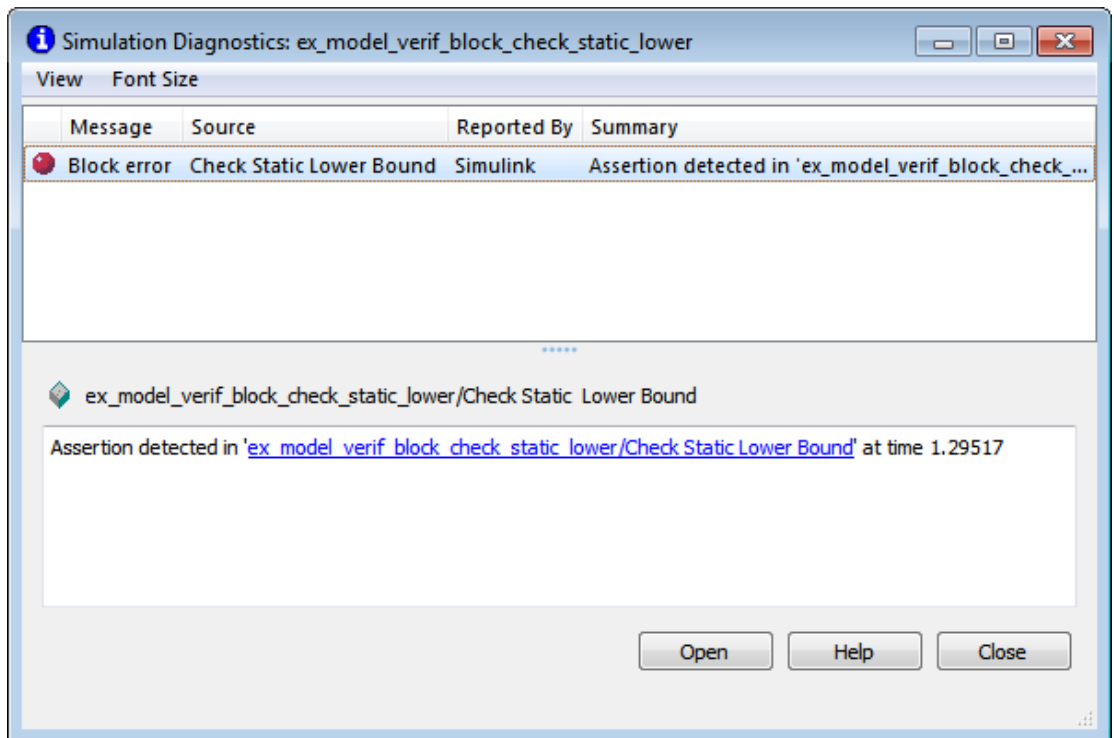
Enable assertion is the default. This parameter enables a verification block for an assertion. You set the **Check Static Lower Bound** block to detect a signal value of –

0.8 or lower. If the signal value reaches that value or falls below it, the simulation stops.

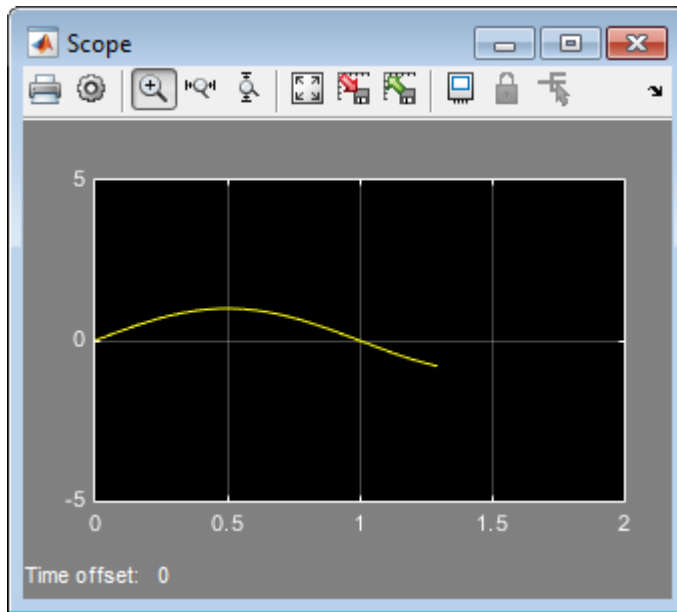
5 Run the simulation.

The model stops simulating after 1.295 seconds, when the output is -0.8 . The software highlights the Check Static Lower Bound block.

When the simulation stops, you see the following diagnostic message.

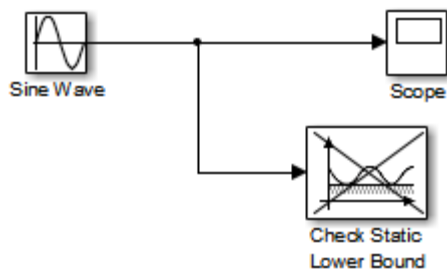


6 To verify the signal value, double-click the Scope block.





- 7 To disable the Check Static Lower Bound block from asserting its limit, clear the **Enable assertion** check box.

The block is crossed out in the model, as shown.



Link Test Cases to Requirements Documents Using the Verification Manager

You can link requirements documents to test cases and their corresponding Model Verification blocks through the Verification Manager **Requirements** pane in the Signal Builder.

- 1 To display the **Requirements** pane in the Signal Builder dialog box:
 - a Click the **Show verification settings** button (.
 - b Click the **Requirements display** button (.
- 2 In the **Requirements** pane, right-click anywhere.
- 3 From the context menu, select **Link Editor**.

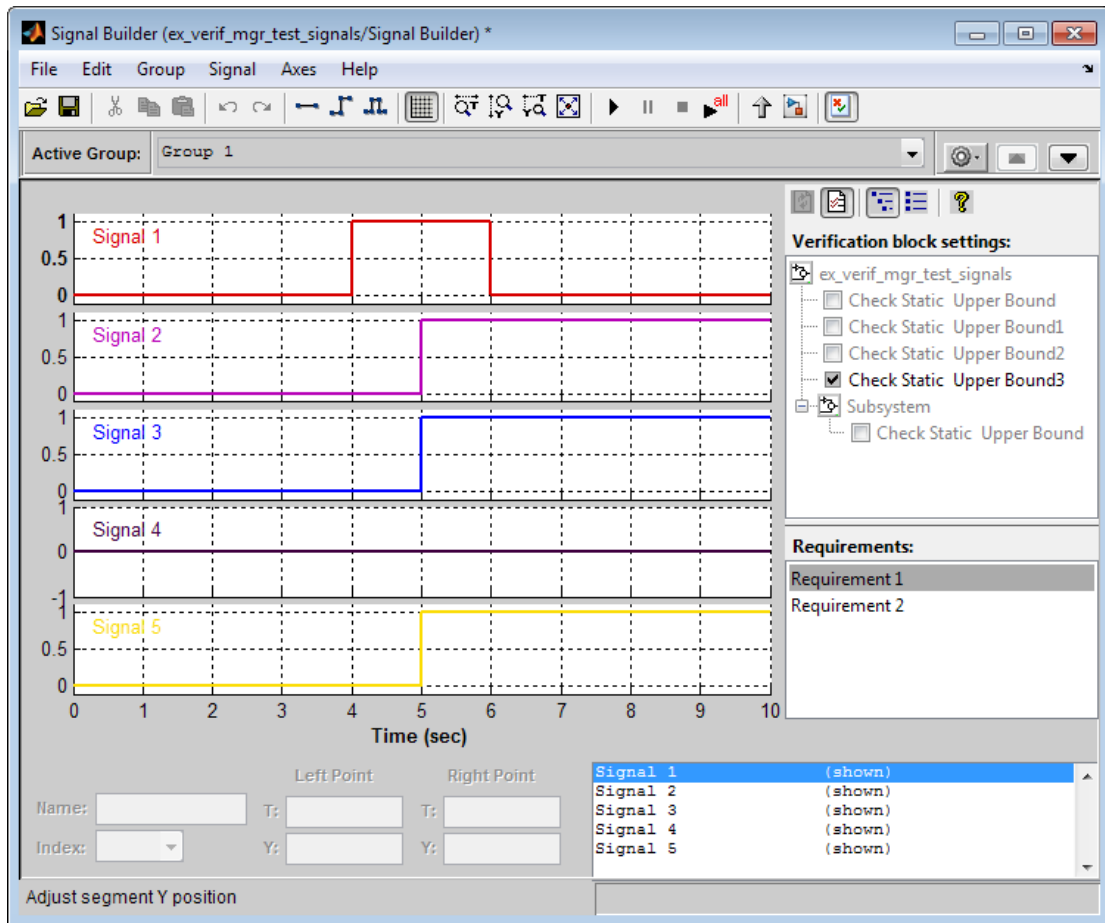
The Requirements dialog box opens.

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

For information about which setting to use for your working environment, see “Document Path Storage” on page 6-14.

- 5 Add links to requirements documents, as described in “Link to Requirements Document Using Selection-Based Linking” on page 3-11.

The names of the linked requirements appear in the **Requirements** pane.



- 6 To view the requirements document in its native editor, right-click a requirement link and select **View**.
- 7 Optionally, to delete a requirement link, right-click the link and select **Delete**.

Model Coverage Analysis

Model Coverage Definition

- “Model Coverage” on page 16-2
- “Types of Model Coverage” on page 16-3
- “Simulink Optimizations and Model Coverage” on page 16-11

Model Coverage

Abstract

Validate your model tests by measuring how thoroughly the model objects are tested.

Model coverage helps you validate your model tests by measuring how thoroughly the model objects are tested. Model coverage calculates how much a model test case exercises simulation pathways through a model. Model coverage is a measure of how thoroughly a test case tests a model and the percentage of pathways that a test case exercises. Model coverage helps you validate your model tests.

Model coverage analyzes the execution of the following types of model objects that directly or indirectly determine simulation pathways through your model:

- Simulink blocks
- Models referenced in Model blocks
- The states and transitions of Stateflow charts

During a simulation run, the tool records the behavior of the covered objects, states, and transitions. At the end of the simulation, the tool reports the extent to which the run exercised potential simulation pathways through each covered object in the model.

The Simulink Verification and Validation software can only collect model coverage for a model if its simulation mode is set to **Normal**, **SIL**, or **PIL**. If the simulation mode is set to any other mode, model coverage is not measured during simulation.

For the types of coverage that model coverage performs, see “Types of Model Coverage” on page 16-3. For an example of a model coverage report, see “Top-Level Model Coverage Report” on page 21-12.

If you have an Embedded Coder license, you can also measure code coverage for code generated from models in software-in-the-loop (SIL) mode or processor-in-the-loop (PIL) mode. For the types of coverage that code coverage performs, see “Types of Code Coverage” on page 19-2. For an example of how to enable code coverage, see “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode” on page 19-6

Types of Model Coverage

Simulink Verification and Validation can perform several types of coverage analysis.

In this section...

“Execution Coverage (EC)” on page 16-3
“Decision Coverage (DC)” on page 16-3
“Condition Coverage (CC)” on page 16-3
“Modified Condition/Decision Coverage (MCDC)” on page 16-4
“Cyclomatic Complexity” on page 16-5
“Lookup Table Coverage” on page 16-5
“Signal Range Coverage” on page 16-6
“Signal Size Coverage” on page 16-6
“Objectives and Constraints Coverage” on page 16-7
“Saturate on Integer Overflow Coverage” on page 16-8
“Relational Boundary Coverage” on page 16-8

Execution Coverage (EC)

Execution coverage is the most basic form of coverage. For each item, execution coverage determines whether the item is executed during simulation.

Decision Coverage (DC)

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

For an example of decision coverage data in a model coverage report, see “Decisions Analyzed” on page 21-24.

Condition Coverage (CC)

Condition coverage analyzes blocks that output the logical combination of their inputs (for example, the Logical Operator block) and Stateflow transitions. A test case achieves

full coverage when it causes each input to each instance of a logic block in the model and each condition on a transition to be true at least once during the simulation, and false at least once during the simulation. Condition coverage analysis reports whether the test case fully covered the block for each block in the model.

When you collect coverage for a model, you may not be able to achieve 100% condition coverage. For example, if you specify to short-circuit logic blocks, by selecting **Treat Simulink Logic blocks as short-circuited** in the **Coverage** pane of the Configuration Parameters, you might not be able to achieve 100% condition coverage for that block. See “Treat Simulink logic blocks as short-circuited” on page 18-8 for more information.

For an example of condition coverage data in a model coverage report, see “Conditions Analyzed” on page 21-25.

Modified Condition/Decision Coverage (MCDC)

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

- A test case achieves full coverage for a block when a change in one input, independent of any other inputs, causes a change in the block output.
- A test case achieves full coverage for a Stateflow transition when there is at least one time when a change in the condition triggers the transition for each condition.

If your model contains blocks that define expressions that have different types of logical operators and more than 12 conditions, the software cannot record MCDC coverage.

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

Some Simulink objects support MCDC coverage, some objects support only condition coverage, and some objects support only decision coverage. The table in “Model Objects That Receive Coverage” on page 17-2 lists which objects receive which types of model coverage. For example, the Combinatorial Logic block can receive decision coverage and condition coverage, but not MCDC coverage.

To achieve 100% MCDC coverage for your model, as defined by the DO-178C/DO-331 standard, in the **Coverage** pane of the Configuration Parameters, select “Modified

Condition/Decision Coverage (MCDC)” on page 16-4 as the **Structural coverage level**.

When you collect coverage for a model, you may not be able to achieve 100% MCDC coverage. For example, if you specify to short-circuit logic blocks, you may not be able to achieve 100% MCDC coverage for that block.

If you run the test cases independently and accumulate all the coverage results, you can determine if your model adheres to the modified condition and decision coverage standard. For more information about the DO-178C/DO-331 standard, see “DO-178C/DO-331 Checks”.

For an example of MCDC coverage data in a model coverage report, see “MCDC Analysis” on page 21-26. For an example of accumulated coverage results, see “Cumulative Coverage” on page 21-27.

Cyclomatic Complexity

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

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

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

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

For an example of cyclomatic complexity data in a model coverage report, see “Cyclomatic Complexity” on page 21-22.

Lookup Table Coverage

Lookup table coverage (LUT) examines blocks, such as the 1-D Lookup Table block, that output information from inputs in a table of inputs and outputs, interpolating between or extrapolating from table entries. Lookup table coverage records the frequency that

table lookups use each interpolation interval. A test case achieves full coverage when it executes each interpolation and extrapolation interval at least once. For each lookup table block in the model, the coverage report displays a colored map of the lookup table, indicating each interpolation. If the total number of breakpoints of an n-D Lookup Table block exceeds 1,500,000, the software cannot record coverage for that block.

For an example of lookup table coverage data in a model coverage report, see “N-Dimensional Lookup Table” on page 21-30.

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

Signal Range Coverage

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

The software does not record signal range coverage for control signals, signals used by one block to initiate execution of another block. See “Control Signals”.

If the total number of signals in your model exceeds 65535, or your model contains a signal whose width exceeds 65535, the software cannot record signal range coverage.

For an example of signal range coverage data in a model coverage report, see “Signal Range Analysis” on page 21-42.

Note: When you create cumulative coverage for reusable subsystems or Stateflow constructs with single range coverage, the cumulative coverage has the largest possible range of signal values. For more information, see “Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs” on page 23-8.

Signal Size Coverage

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

If the total number of signals in your model exceeds 65535, or your model contains a signal whose width exceeds 65535, the software cannot record signal size coverage.

For an example of signal size coverage data in a model coverage report, see “Signal Size Coverage for Variable-Dimension Signals” on page 21-44.

For more information about variable-size signals, see “Variable-Size Signal Basics”.

Objectives and Constraints Coverage

The Simulink Verification and Validation software collects model coverage data for the following Simulink Design Verifier blocks and MATLAB for code generation functions:

Simulink Design Verifier blocks	MATLAB for code generation functions
Test Condition	sldv.condition
Test Objective	sldv.test
Proof Assumption	sldv.assume
Proof Objective	sldv.prove

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

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

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

If you specify to collect Simulink Design Verifier coverage:

- The software collects coverage for the Simulink Design Verifier blocks and functions.
- The software checks the data type of the signal that links to each Simulink Design Verifier block. If the signal data type is fixed point, the block parameter must also be fixed point. If the signal data type is not fixed point, the software tries to convert the block parameter data type. If the software cannot convert the block parameter data type, the software reports an error and you must explicitly assign the block parameter data type to match the signal.
- If your model contains a Verification Subsystem block, the software only records coverage for Simulink Design Verifier blocks in the **Verification Subsystem** block; it does not record coverage for any other blocks in the Verification Subsystem.

If you do not specify to collect Simulink Design Verifier coverage, the software does not check the data types for any Simulink Design Verifier blocks and functions in your model and does not collect coverage.

For an example of coverage data for Simulink Design Verifier blocks or functions in a model coverage report, see “Simulink Design Verifier Coverage” on page 21-45.

Saturate on Integer Overflow Coverage

Saturate on integer overflow coverage examines blocks, such as the Abs block, with the **Saturate on integer overflow** parameter selected. Only blocks with this parameter selected receive saturate on integer overflow coverage.

Saturate on integer overflow coverage records the number of times the block saturates on integer overflow.

A test case achieves full coverage when the blocks saturate on integer overflow at least once and does not saturate at least once.

For an example of saturate on integer overflow coverage data in a model coverage report, see “Saturate on Integer Overflow Analysis” on page 21-41.

Relational Boundary Coverage

Relational boundary coverage examines blocks, Stateflow charts, and MATLAB function blocks that have an explicit or implicit relational operation.

- Blocks such as **Relational Operator** and **If** have an explicit relational operation.

- Blocks such as `Abs` and `Saturation` have an implicit relational operation.

For these model objects, the metric records whether a simulation tests the relational operation with:

- Equal operand values.

This part of relational boundary coverage applies only if both operands are integers or fixed-point numbers.

- Operand values that differ by a certain tolerance.

This part of relational boundary coverage applies to all operands. For integer and fixed-point operands, the tolerance is fixed. For floating-point operands, you can either use a predefined tolerance or you can specify your own tolerance.

The tolerance value depends on the data type of both the operands. If both operands have the same type, the tolerance follows the following rules:

Data Type of Operand	Tolerance
Floating point, such as <code>single</code> or <code>double</code>	$\max(\text{absTol}, \text{relTol} * \max(\text{lhs} , \text{rhs}))$ <ul style="list-style-type: none"> • <code>absTol</code> is an absolute tolerance value you specify. Default is <code>1e-05</code>. • <code>relTol</code> is a relative tolerance value you specify. Default is <code>0.01</code>. • <code>lhs</code> is the left operand and <code>rhs</code> the right operand. • <code>max(x, y)</code> returns <code>x</code> or <code>y</code>, whichever is greater.
Fixed point	Value corresponding to least significant bit. For more information, see “Precision”. To find the precision value, use the <code>lsb</code> function.
Integer	1
Boolean	N/A
Enum	N/A

If the two operands have different types, the tolerance follows the rules for the stricter type. If one of the operands is boolean, the tolerance follows the rules for the other operand. The strictness decreases in this order:

- 1** Floating point
- 2** Fixed point
- 3** Integer

If both operands are fixed point but have different precision, the smaller value of precision is used as tolerance.

You specify the value of absolute and relative tolerances for relational boundary coverage of floating point inputs when you select this metric in the **Coverage metrics** section in the “Coverage Pane” on page 18-2 of the Configuration Parameters dialog box.

For more information on:

- How this coverage metric appears in reports, see “Relational Boundary” on page 21-37.
- Which model objects receive this coverage, see the table in “Model Objects That Receive Coverage” on page 17-2.
- How to obtain coverage results from the MATLAB command-line, see “Collect Relational Boundary Coverage for Supported Block in Model”.

Simulink Optimizations and Model Coverage

Abstract

Learn how inlined parameters, block reduction, and conditional input branch execution can affect your model coverage data.

In the Configuration Parameters dialog box, there are three Simulink optimization parameters that can affect your model coverage data:

In this section...

“Inlined parameters” on page 16-11

“Block reduction” on page 16-11

“Conditional input branch execution” on page 16-12

Inlined parameters

To transform tunable model parameters into constant values for code generation, in the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, set **Default parameter behavior** to **Inlined**.

When the parameters are transformed into constants, Simulink may eliminate certain decisions in your model. You cannot achieve coverage for eliminated decision, so the coverage report displays 0/0 for those decisions.

Block reduction

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

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

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

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

Conditional input branch execution

To improve model execution when the model contains Switch and Multiport Switch blocks, in the Configuration Parameters dialog box, on the **All Parameters** tab, select **Conditional input branch execution**. If you select this parameter, the simulation executes only blocks that are required to compute the control input and the data input selected by the control input.

When Conditional input branch execution is enabled, instead of executing all blocks driving the Switch block input ports at each time step, only the blocks required to compute the control input and the data input selected by the control input execute.

Several considerations affect or limit Switch block optimization:

- Only blocks with `-1` (inherited) or `inf` (Constant) sample time can be optimized.
- Blocks with outputs flagged as test points cannot be optimized.
- Multirate blocks cannot be optimized.
- Blocks with states cannot be optimized.
- Only S-functions with the `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` option enabled can be optimized.

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

Conditional input branch execution does not apply to Stateflow charts.

Model Objects That Receive Model Coverage

Model Objects That Receive Coverage

Certain Simulink objects can receive any type of model coverage. Other Simulink objects can receive only certain types of coverage, as the following table shows. Click a link in the first column to get more detailed information about coverage for specific model objects.

All Simulink objects can receive Execution coverage, except blocks that are not instrumented in model coverage:

- Merge Blocks
- Scope Blocks
- Outport Blocks
- Inport Blocks
- Width Blocks
- Display Blocks

For Stateflow states, events, and state temporal logic decisions, model coverage provides only decision coverage. For Stateflow transitions, model coverage provides decision, condition, and MCDC coverage. For more information, see “Model Coverage for Stateflow Charts” on page 20-48.

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
“Abs” on page 17-8	•					•	•
“Bias” on page 17-9						•	
“Combinatorial Logic” on page 17-9	•	•					
“Compare to Constant” on page 17-9		•					•
“Compare to Zero” on page 17-10		•					•

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
“Data Type Conversion” on page 17-10						•	
“Dead Zone” on page 17-10	•					•	•
“Direct Lookup Table (n-D)” on page 17-11				•			
“Discrete Filter” on page 17-12						•	
“Discrete FIR Filter” on page 17-12						•	
“Discrete-Time Integrator” on page 17-12 (when saturation limits are enabled or reset)	•					•	
“Discrete Transfer Fcn” on page 17-13						•	
“Dot Product” on page 17-13						•	
“Enabled Subsystem” on page 17-13	•	•	•				
“Enabled and Triggered Subsystem” on page 17-14	•	•	•				

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
“Fcn” on page 17-15		•	•				•
“For Iterator, For Iterator Subsystem” on page 17-16	•						
“Gain” on page 17-16						•	
“If, If Action Subsystem” on page 17-16	•	•	•				•
“Interpolation Using Prelookup” on page 17-17				•		•	
“Library-Linked Objects” on page 17-18	•	•	•	•	•		
“Logical Operator” on page 17-18		•	•				
“1-D Lookup Table” on page 17-18				•		•	
“2-D Lookup Table” on page 17-19				•		•	
“n-D Lookup Table” on page 17-20				•		•	

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
“Math Function” on page 17-20						•	
“MATLAB Function” on page 17-20	•	•	•				•
“MATLAB System” on page 17-21	•	•	•				
“MinMax” on page 17-21	•					•	
“Model” on page 17-21 See also “Triggered Models” on page 17-30.	•	•	•	•	•	•	•
“Multiport Switch” on page 17-22	•					•	
“PID Controller, PID Controller (2 DOF)” on page 17-22						•	
“Product” on page 17-23						•	
“Proof Assumption” on page 17-23					•		
“Proof Objective” on page 17-23					•		

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
“Rate Limiter” on page 17-23	● (Relative to slew rates)						●
“Relational Operator” on page 17-24		●					●
“Relay” on page 17-25	●						●
“C/C++ S-Function” on page 17-25	●	●	●				
“Saturation” on page 17-27	●						●
“Saturation Dynamic” on page 17-27						●	
“Simulink Design Verifier Functions in MATLAB Function Blocks” on page 17-28					●		
Stateflow charts	●	●	●				●
Stateflow state transition tables	●	●	●				●
“Sqrt, Signed Sqrt, Reciprocal Sqrt” on page 17-28						●	

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
“Sum, Add, Subtract, Sum of Elements” on page 17-28						•	
“Switch” on page 17-28	•					•	•
“SwitchCase, SwitchCase Action Subsystem” on page 17-29	•						
“Test Condition” on page 17-29					•		
“Test Objective” on page 17-30					•		
“Triggered Models” on page 17-30	•	•	•				
“Triggered Subsystem” on page 17-31	•	•	•				
“Truth Table” on page 17-32	•	•	•				
“Unary Minus” on page 17-32						•	
“Weighted Sample Time Math” on page 17-32						•	

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
“While Iterator, While Iterator Subsystem” on page 17-32	•						

Abs

The Abs block receives decision coverage. Decision coverage is based on:

- Input to the block being less than zero.
- Data type of the input signal.

For input to the block being less than zero, the decision coverage measures:

- The number of time steps that the block input is less than zero, indicating a true decision.
- The number of time steps the block input is not less than zero, indicating a false decision.

If you select the **Saturate on integer overflow** coverage metric, the Abs block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

If the input data type to the Abs block is `uint8`, `uint16`, or `uint32`, the Simulink Verification and Validation software reports no coverage for the block. The software sets the block output equal to the block input without making a decision. If the input data type to the Abs block is Boolean, an error occurs.

The Abs block contains an implicit comparison of the input with zero. Therefore, if you select the **Relational Boundary** coverage metric, the Abs block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Bias

If you select the **Saturate on integer overflow** coverage metric, the Bias block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

Combinatorial Logic

The Combinatorial Logic block receives decision and condition coverage. Decision coverage is based on achieving each output row of the truth table. The decision coverage measures the number of time steps that each output row of the truth table is set to the block output.

The condition coverage measures the number of time steps that each input is false (equal to zero) and the number of times each input is true (not equal to zero). If the Combinatorial Logic block has a single input element, the Simulink Verification and Validation software reports only decision coverage, because decision and condition coverage are equivalent.

If all truth table values are set to the block output for at least one time step, decision coverage is 100%. Otherwise, the software reports the coverage as the number of truth table values output during at least one time step, divided by the total number of truth table values. Because this block always has at least one value in the truth table as output, the minimum coverage reported is one divided by the total number of truth table values.

If all block inputs are false for at least one time step and true for at least one time step, condition coverage is 100%. Otherwise, the software reports the coverage as achieving a false value at each input for at least one time step, plus achieving a true value for at least one time step, divided by two raised to the power of the total number of inputs (i.e., $2^{\text{number_of_inputs}}$). The minimum coverage reported is the total number of inputs divided by two raised to the power of the total number of inputs.

Compare to Constant

The Compare to Constant block receives condition coverage.

Condition coverage measures:

- the number of times that the comparison between the input and the specified constant was true.

- the number of times that the comparison between the input and the specified constant was false.

The Compare to Constant block contains a comparison of the input with a constant. Therefore, if you select the **Relational Boundary** coverage metric, the Compare to Constant block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Compare to Zero

The Compare to Zero block receives condition coverage.

Condition coverage measures:

- the number of times that the comparison between the input and zero was true.
- the number of times that the comparison between the input and zero was false.

The Compare to Zero block contains a comparison of the input with zero. Therefore, if you select the **Relational Boundary** coverage metric, the Compare to Zero block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Data Type Conversion

If you select the **Saturate on integer overflow** coverage metric, the Data Type Conversion block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

Dead Zone

The Dead Zone block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for these parameters:

- **Start of dead zone**
- **End of dead zone**

The **Start of dead zone** parameter specifies the lower limit of the dead zone. For the **Start of dead zone** parameter, decision coverage measures:

- The number of time steps that the block input is greater than or equal to the lower limit, indicating a true decision.

- The number of time steps that the block input is less than the lower limit, indicating a false decision.

The **End of dead zone** parameter specifies the upper limit of the dead zone. For the **End of dead zone**, decision coverage measures:

- The number of time steps that the block input is greater than the upper limit, indicating a true decision.
- The number of time steps that the block input is less than or equal to the upper limit, indicating a false decision.

When the upper limit is true, the software does not measure **Start of dead zone** coverage for that time step. Therefore, the total number of **Start of dead zone** decisions equals the number of time steps that the **End of dead zone** is false.

If you select the **Saturate on integer overflow** coverage metric, the Dead Zone block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

The Dead Zone block contains an implicit comparison of the input with an upper and lower limit value. Therefore, if you select the **Relational Boundary** coverage metric, the Dead Zone block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Direct Lookup Table (n-D)

The Direct Lookup Table (n-D) block receives lookup table coverage. For an n -dimensional lookup table, the number of output break points is the product of all the number of break points for each table dimension.

Lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.
- The number of times during simulation that each combination of dimension input values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for an n -dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with

zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

Discrete Filter

If you select the **Saturate on integer overflow** coverage metric, the Discrete Filter block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

Discrete FIR Filter

If you select the **Saturate on integer overflow** coverage metric, the Discrete FIR Filter block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

Discrete-Time Integrator

The Discrete-Time Integrator block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for these parameters:

- **External reset**
- **Limit output**

If you set **External reset** to none, the Simulink Verification and Validation software does not report decision coverage for the reset decision. Otherwise, the decision coverage measures:

- The number of time steps that the block output is reset, indicating a true decision.
- The number of time steps that the block output is not reset, indicating a false decision.

If you do not select **Limit output**, the software does not report decision coverage for that decision. Otherwise, the software reports decision coverage for the **Lower saturation limit** and the **Upper saturation limit**.

For the **Upper saturation limit**, decision coverage measures:

- The number of time steps that the integration result is greater than or equal to the upper limit, indicating a true decision.
- The number of time steps that the integration result is less than the upper limit, indicating a false decision.

For the **Lower saturation limit**, decision coverage measures

- The number of time steps that the integration result is less than or equal to the lower limit, indicating a true decision.
- The number of time steps that the integration result is greater than the lower limit, indicating a false decision.

For a time step when the upper limit is true, the software does not measure **Lower saturation limit** coverage. Therefore, the total number of lower limit decisions equals the number of time steps that the upper limit is false.

If you select the **Saturate on integer overflow** coverage metric, the Discrete-Time Integrator block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

Discrete Transfer Fcn

If you select the **Saturate on integer overflow** coverage metric, the Discrete Transfer Fcn block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

Dot Product

If you select the **Saturate on integer overflow** coverage metric, the Dot Product block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

Enabled Subsystem

The Enabled Subsystem block receives decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the block is enabled, indicating a true decision.

- The number of time steps that the block is disabled, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The Simulink Verification and Validation software measures condition coverage for the enable input only if the enable input is a vector. For the enable input, condition coverage measures the number of time steps each element of the enable input is true and the number of time steps each element of the enable input is false. The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

The software measures MCDC coverage for the enable input only if the enable input is a vector. Because the enable of the subsystem is an OR of the vector inputs, MCDC coverage is 100% if, during at least one time step, each vector enable input is exclusively true and if, during at least one time step, all vector enable inputs are false. For MCDC coverage measurement, the software treats each element of the vector as a separate condition.

Enabled and Triggered Subsystem

The Enabled and Triggered Subsystem block receives decision, condition, and MCDC coverage. Decision coverage measures:

- The number of time steps that a trigger edge occurs while the block is enabled, indicating a true decision.
- The number of time steps that a trigger edge does not occur while the block is enabled, or the block is disabled, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The software measures condition coverage for the enable input and for the trigger input separately:

- For the enable input, condition coverage measures the number of time steps the enable input is true and the number of time steps the enable input is false.
- For the trigger input, condition coverage measures the number of time steps the trigger edge occurs, indicating true, and the number of time steps the trigger edge does not occur, indicating false.

The software reports condition coverage based on the total number of possible conditions and how many conditions are true for at least one time step and how many are false for at least one time step. The software treats each element of a vector as a separate condition coverage measurement.

The software measures MCDC coverage for the enable input and for the trigger input in combination. Because the enable input of the subsystem is an AND of these two inputs, MCDC coverage is 100% if all of the following occur:

- During at least one time step, both inputs are true.
- During at least one time step, the enable input is true and the trigger edge is false.
- During one time step, the enable input is false and the trigger edge is true.

The software treats each vector element as a separate MCDC coverage measurement. It measures each trigger edge element against each enable input element. However, if the number of elements in both the trigger and enable inputs exceeds 12, the software does not report MCDC coverage.

Fcn

The Fcn block receives condition and MCDC coverage. The Simulink Verification and Validation software reports condition or MCDC coverage for Fcn blocks only if the top-level operator is Boolean (&&, ||, or !).

Condition coverage is based on input values or arithmetic expressions that are inputs to Boolean operators in the block. The condition coverage measures:

- The number of time steps that each input to a Boolean operator is true (not equal to zero).
- The number of time steps that each input to a Boolean operator is false (equal to zero).

If all Boolean operator inputs are false for at least one time step and true for at least one time step, condition coverage is 100%. Otherwise, the software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

The software measures MCDC coverage for Boolean expressions within the Fcn block. If, during at least one time step, each condition independently sets the output of the

expression to true and if, during at least one time step, each condition independently sets the output of the expression to false, MCDC coverage is 100%. Otherwise, the software reports MCDC coverage based on the total number of possible conditions and how many times each condition independently sets the output to true during at least one time step and how many conditions independently set the output to false during at least one time step.

If the Fcn block contains a relational operation and you select the **Relational Boundary** coverage metric, the Fcn block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

For Iterator, For Iterator Subsystem

The For Iterator block and For Iterator Subsystem receive decision coverage. The Simulink Verification and Validation software measures decision coverage for the loop condition value, which is determined by one of the following:

- The iteration value being at or below the iteration limit, indicated as true.
- The iteration value being above the iteration limit, indicated as false.

The software reports the total number of times that each loop condition evaluates to true and to false. If the loop condition evaluates to true at least once and false at least once, decision coverage is 100%. If no loop conditions are true, or if no loop conditions are false, decision coverage is 50%.

Gain

If you select the **Saturate on integer overflow** coverage metric, the Gain block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8.

If, If Action Subsystem

The If block that causes an If Action Subsystem to execute receives condition, decision, and MCDC coverage:

- The software measures decision coverage for the `if` condition and all `elseif` conditions defined in the If block.

- If the `if` condition or any of the `elseif` conditions contains a logical expression with multiple conditions, such as `u1 & u2 & u3`, the software also measures condition and MCDC coverage for each condition in the expression, `u1`, `u2`, and `u3` in the preceding example.

The software does not directly measure the `else` condition. When there are no `elseif` conditions, the `else` condition is the direct complement of the `if` condition, or the `else` condition is the direct complement of the last `elseif` condition.

The software reports the total number of time steps that each `if` and `elseif` condition evaluates to true and to false. If the `if` or `elseif` condition evaluates to true at least once, and evaluates to false at least once, decision coverage is 100%. If no `if` or `elseif` conditions are true, or if no `if` or `elseif` conditions are false, decision coverage is 50%. If the previous `if` or `elseif` condition never evaluates as false, an `elseif` condition can have 0% decision coverage.

The `If` block contains a comparison between its inputs. Therefore, if you select the **Relational Boundary** coverage metric, the `If` block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Interpolation Using Prelookup

The Interpolation Using Prelookup block receives lookup table coverage. For an n -D lookup table, the number of output break points equals the product of all the number of break points for each table dimension. The lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.
- The number of times during simulation that each combination of dimension input values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for an n -dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

If you select the **Saturate on integer overflow**, the Interpolation Using Prelookup block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Library-Linked Objects

Simulink blocks and Stateflow charts that are linked to library objects receive the same coverage that they would receive if they were not linked to library objects. The Simulink Verification and Validation software records coverage individually for each library object in the model. If your model contains multiple instances of the same library object, each instance receives its own coverage data.

Logical Operator

The Logical Operator block receives condition and MCDC coverage. The Simulink Verification and Validation software measures condition coverage for each input to the block. The condition coverage measures:

- The number of time steps that each input is true (not equal to zero).
- The number of time steps that each input is false (equal to zero).

If all block inputs are false for at least one time step and true for at least one time step, the software condition coverage is 100%. Otherwise, the software reports the condition coverage based on the total number of possible conditions and how many are true at least one time step and how many are false at least one time step.

The software measures MCDC coverage for all inputs to the block. If, during at least one time step, each condition independently sets the output of the block to true and if, during at least one time step, each condition independently sets the output of the block to false, MCDC coverage is 100%. Otherwise, the software reports the MCDC coverage based on the total number of possible conditions and how many times each one of them independently set the output to true for at least one time step and how many independently set the output to false for at least one time step.

1-D Lookup Table

The 1-D Lookup Table block receives lookup table coverage; for a one-dimensional lookup table, the number of input and output break points is equal. Lookup table coverage measures:

- The number of times during simulation that the input and output values are between each of the break points.
- The number of times during simulation that the input and output values are below the lowest break point and above the highest break point.

The total number of coverage points for a one-dimensional lookup table is the number of break points in the table plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

If you select the **Saturate on integer overflow** coverage metric, the 1-D Lookup Table block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

2-D Lookup Table

The 2-D Lookup Table block receives lookup table coverage. For a two-dimensional lookup table, the number of output break points equals the number of row break points multiplied by the number of column inputs. Lookup table coverage measures:

- The number of times during simulation that each combination of row input and column input values is between each of the break points.
- The number of times during simulation that each combination of row input and column input values is below the lowest break point and above the highest break point for each row and column.

The total number of coverage points for a two-dimensional lookup table is the number of row break points in the table plus one, multiplied by the number of column break points in the table plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

If you select the **Saturate on integer overflow** coverage metric, the 2-D Lookup Table block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

n-D Lookup Table

The n-D Lookup Table block receives lookup table coverage. For an n -dimensional lookup table, the number of output break points equals the product of all the number of break points for each table dimension. Lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.
- The number of times during simulation that each combination of dimension output values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for an n -dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

If you select the **Saturate on integer overflow** coverage metric, the n-D Lookup Table block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Math Function

If you select the **Saturate on integer overflow** coverage metric, the Math Function block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

MATLAB Function

For information about the type of coverage that the Simulink Verification and Validation software reports for the MATLAB Function block, see “Model Coverage for MATLAB Functions” on page 20-23.

MATLAB System

Simulink Verification and Validation records only Decision, Condition, and MCDC coverage for MATLAB System blocks.

MinMax

The MinMax block receives decision coverage based on passing each input to the output of the block.

For decision coverage based on passing each input to the output of the block, the coverage measures the number of time steps that the simulation passes each input to the block output. The number of decision points is based on the number of inputs to the block and whether they are scalar, vector, or matrix.

If all inputs are passed to the block output for at least one time step, the Simulink Verification and Validation software reports the decision coverage as 100%. Otherwise, the software reports the coverage as the number of inputs passed to the output during at least one time step, divided by the total number of inputs.

If you select the **Saturate on integer overflow** coverage metric, the MinMax block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Model

The Model block does not receive coverage directly; the model that the block references receives coverage. If the simulation mode for the referenced model is set to **Normal**, the Simulink Verification and Validation software reports coverage for all objects within the referenced model that receive coverage. . If the simulation mode for the referenced model is set to **SIL** or **PIL** and you have Embedded Coder installed, the Simulink Verification and Validation software reports coverage for the code generated from your model .If the simulation mode is set to a value other than **Normal**, **SIL**, or **PIL**, the software cannot measure coverage for the referenced model.

In the **Coverage** pane of the Configuration Parameters dialog box, select the referenced models for which you want to report coverage. The software generates a coverage report for each referenced model you select.

If your model contains multiple instances of the same referenced model, the software records coverage for all instances of that model where the simulation mode of the Model block is set to **Normal**. The coverage report for that referenced model combines the coverage data for all Normal mode instances of that model.

The coverage reports for referenced models are linked from a summary report for the parent model.

Note: For details on how to select referenced models to report coverage, see “Referenced Models” on page 18-4.

Multiport Switch

The Multiport Switch block receives decision coverage based on passing each input, excluding the first control input, to the output of the block.

For decision coverage based on passing each input, excluding the first control input, to the output of the block, the coverage measures the number of time steps that each input is passed to the block output. The number of decision points is based on the number of inputs to the block and whether the control input is scalar or vector.

If all inputs, excluding the first control input, are passed to the block output for at least one time step, decision coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of inputs passed to the output during at least one time step, divided by the total number of inputs minus one.

If you select the **Saturate on integer overflow** coverage metric, the Multiport Switch block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

PID Controller, PID Controller (2 DOF)

If you select the **Saturate on integer overflow** coverage metric, the PID Controller and PID Controller (2 DOF) blocks receive saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Product

If you select the **Saturate on integer overflow** coverage metric, the Product block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Proof Assumption

The Proof Assumption block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Proof Assumption block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Proof Objective

The Proof Objective block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Proof Objective block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Rate Limiter

The Rate Limiter block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for the **Rising slew rate** and **Falling slew rate** parameters.

For the **Rising slew rate**, decision coverage measures:

- The number of time steps that the block input changes more than or equal to the rising rate, indicating a true decision.
- The number of time steps that the block input changes less than the rising rate, indicating a false decision.

For the **Falling slew rate**, decision coverage measures:

- The number of time steps that the block input changes less than or equal to the falling rate, indicating a true decision.
- The number of time steps that the block input changes more than the falling rate, indicating a false decision.

The software does not measure **Falling slew rate** coverage for a time step when the **Rising slew rate** is true. Therefore, the total number of **Falling slew rate** decisions equals the number of time steps that the **Rising slew rate** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

The Rate Limiter block implicitly compares the derivative of the input signal with an upper and lower limit value. Therefore, if you select the **Relational Boundary** coverage metric, the Rate Limiter block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Relational Operator

The Relational Operator block receives condition coverage.

Condition coverage measures:

- the number of times that the specified relational operation was true.
- the number of times that the specified relational operation was false.

The Relational Operator block contains a comparison between its inputs. Therefore, if you select the **Relational Boundary** coverage metric, the Relational Operator block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Relay

The Relay block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for the **Switch on point** and the **Switch off point** parameters.

For the **Switch on point**, decision coverage measures:

- The number of consecutive time steps that the block input is greater than or equal to the **Switch on point**, indicating a true decision.
- The number of consecutive time steps that the block input is less than the **Switch on point**, indicating a false decision.

For the **Switch off point**, decision coverage measures:

- The number of consecutive time steps that the block input is less than or equal to the **Switch off point**, indicating a true decision.
- The number of consecutive time steps that the block input is greater than the **Switch off point**, indicating a false decision.

The software does not measure **Switch off point** coverage for a time step when the switch on threshold is true. Therefore, the total number of **Switch off point** decisions equals the number of time steps that the **Switch on point** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

The Relay block contains an implicit comparison of its second input with a threshold value. Therefore, if you select the **Relational Boundary** coverage metric, the Relay block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

C/C++ S-Function

Model coverage is supported for C/C++ S-Functions. The coverage report for the model contains results for each instance of an **S-Function** block in the model. The results for an S-Function block link to a separate coverage report for the C/C++ code in the block.

To generate coverage report for S-Functions:

- 1 When creating the S-Functions, enable support for coverage. For more information, see “Make S-Function Compatible with Model Coverage” on page 20-40.
- 2 When generating the coverage report, enable support for S-Functions. For more information, see “Generate Coverage Report for S-Function” on page 20-41.

The following coverage types are reported for S-Functions:

- “Cyclomatic Complexity for Code Coverage” on page 19-4
- “Condition Coverage for Code Coverage” on page 19-3
- “Decision Coverage for Code Coverage” on page 19-3
- “Modified Condition/Decision Coverage (MCDC) for Code Coverage” on page 19-4
- “Relational Boundary for Code Coverage” on page 19-5
- Percentage of statements covered

The coverage data for S-Function blocks is obtained in the following way:

- The coverage result for a block is a weighted average of the result over all files in the block.

For instance, an S-Function block has two files, `file1.c` and `file2.c`. The decision coverage for `file1.c` is 75% (3/4 outcomes covered) and that for `file2.c` is 50% (10/20 outcomes covered). The decision coverage for the block is $13/24 \approx 54\%$.

- For each file, the coverage result is a weighted average of the result over all functions in the file.
- For each function, the coverage result is a weighted average of the result over all statements in the function that receive that coverage.

Note: Model coverage for S-Functions have the following restrictions:

- Only Level-2 C/C++ S-Functions are supported for coverage. For an example of a level-2 C S-Function, see “Basic C MEX S-Function”.
 - C++ class templates are not instrumented for coverage.
-

Saturation

The Saturation block receives decision coverage. The Simulink Verification and Validation software reports decision coverage for the **Lower limit** and **Upper limit** parameters.

For the **Upper limit**, decision coverage measures:

- The number of time steps that the block input is greater than or equal to the upper limit, indicating a true decision.
- The number of time steps that the block input is less than the upper limit, indicating a false decision.

For the **Lower limit**, decision coverage measures:

- The number of time steps that the block input is greater than the lower limit, indicating a true decision.
- The number of time steps that the block input is less than or equal to the lower limit, indicating a false decision.

The software does not measure **Lower limit** coverage for a time step when the upper limit is true. Therefore, the total number of **Lower limit** decisions equals the number of time steps that the **Upper limit** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the Saturation block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

The Saturation block contains an implicit comparison of the input with an upper and lower limit value. Therefore, if you select the **Relational Boundary** coverage metric, the Saturation block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Saturation Dynamic

If you select the **Saturate on integer overflow** coverage metric, the Saturation Dynamic block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Simulink Design Verifier Functions in MATLAB Function Blocks

The following functions in MATLAB Function blocks receive Simulink Design Verifier coverage:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

Each of these functions evaluates an expression *expr*, for example, `sldv.test(expr)`, where *expr* is any valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to true.

If *expr* is true for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Verification and Validation software reports coverage for that function as 0%.

Sqrt, Signed Sqrt, Reciprocal Sqrt

If you select the **Saturate on integer overflow** coverage metric, the Sqrt, Signed Sqrt, and Reciprocal Sqrt blocks receive saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Sum, Add, Subtract, Sum of Elements

If you select the **Saturate on integer overflow** coverage metric, the Sum, Add, Subtract, and Sum of Elements blocks receive saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Switch

The Switch block receives decision coverage based on the control input to the block. Decision coverage measures:

- The number of time steps that the control input evaluates to true.

- The number of time steps the control input evaluates to false.

The number of decision points is based on whether the control input is scalar or vector.

If you select the **Saturate on integer overflow** coverage metric, the Switch block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

The Switch block contains an implicit comparison of its second input with a threshold value. Therefore, if you select the **Relational Boundary** coverage metric, the Switch block receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

SwitchCase, SwitchCase Action Subsystem

The SwitchCase block and SwitchCase Action Subsystem receive decision coverage. The Simulink Verification and Validation software measures decision coverage individually for each switch case defined in the block and also for the default case. The number of decision outcomes is equal to the number of case conditions plus one for the `default` case, if one is defined.

The software reports the total number of time steps that each case evaluates to true. If each case, including the default case, evaluates to true at least once, decision coverage is 100%. The software determines the decision coverage by the number of cases that evaluate true for at least one time step divided by the total number of cases.

If the SwitchCase block does not contain a `default` case, the software measures decision coverage for the number of time steps in which none of the cases evaluated to true. In the coverage report, this coverage is reported as **implicit-default**.

Test Condition

The Test Condition block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Test Condition block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and

Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Test Objective

The Test Objective block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Test Objective block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Verification and Validation software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Triggered Models

A Model block can reference a model that contains edge-based trigger ports at the root level of the model. Triggered models receive decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the referenced model is triggered, indicating a true decision.
- The number of time steps that the referenced model is not triggered, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the Model block that references the triggered model is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

Only if the trigger input is a vector, the Simulink Verification and Validation software measures condition coverage for the trigger port in the referenced model. For the trigger port, condition coverage measures:

- The number of time steps that each element of the trigger port is true.
- The number of time steps that each element of the trigger port is false.

The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

If the trigger port is a vector, the software measures MCDC coverage for the trigger port only. Because the trigger port of the referenced model is an **OR** of the vector inputs, if, during at least one time step, each vector trigger port is exclusively true and if, during at least one time step, all vector trigger port inputs are false, MCDC coverage is 100%. The software treats each element of the vector as a separate condition for MCDC coverage measurement.

Triggered Subsystem

The Triggered Subsystem block receives decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the block is triggered, indicating a true decision.
- The number of time steps that the block is not triggered, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The Simulink Verification and Validation software measures condition coverage for the trigger input only if the trigger input is a vector. For the trigger input, condition coverage measures:

- The number of time steps that each element of the trigger edge is true.
- The number of time steps that each element of the trigger edge is false.

The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

If the trigger input is a vector, the software measures MCDC coverage for the trigger input only. Because the trigger edge of the subsystem is an **OR** of the vector inputs, if, during at least one time step, each vector trigger edge input is exclusively true and if, during at least one time step, all vector trigger edge inputs are false, MCDC coverage is 100%. The software treats each element of the vector as a separate condition for MCDC coverage measurement.

Truth Table

The Truth Table block is a Stateflow block that enables you to use truth table logic directly in a Simulink model. The Truth Table block receives condition, decision, and MCDC coverage. For more information on model coverage with Stateflow truth tables, see “Model Coverage for Stateflow Truth Tables” on page 20-69.

Unary Minus

If you select the **Saturate on integer overflow** coverage metric, the Unary Minus block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

Weighted Sample Time Math

If you select the **Saturate on integer overflow** coverage metric, the Weighted Sample Time Math block receives saturate on integer overflow coverage. For more information, see “Saturate on Integer Overflow Coverage” on page 16-8. The software treats each element of a vector or matrix as a separate coverage measurement.

While Iterator, While Iterator Subsystem

The While Iterator block and While Iterator Subsystem receive decision coverage. Decision coverage is measured for the `while` condition value, which is determined by the `while` condition being satisfied (true), or the `while` condition not being satisfied (false). Simulink Verification and Validation software reports the total number of times that each `while` condition evaluates to true and to false. If the `while` condition evaluates to true at least once, and false at least once, decision coverage for the `while` condition is 100%. If no `while` conditions are true, or if no `while` conditions are false, decision coverage is 50%.

If the iteration limit is exceeded (true) or is not exceeded (false), the software measures decision coverage independently. If the iteration limit evaluates to true at least once, and false at least once, decision coverage for the iteration limit is 100%. If no iteration limits are true, or if no iteration limits are false, decision coverage is 50%. If you set **Maximum number of iterations** to -1 (no limit), the decision coverage for the iteration limit is true for all iterations and false for zero iterations, and decision coverage is 50%.

Model Objects That Do Not Receive Coverage

The Simulink Verification and Validation software does not record Decision, Condition, or MCDC coverage for blocks that are not listed in “Model Objects That Receive Coverage” on page 17-2.

Note: The software only records model coverage when the **Simulation mode** parameter is set to **Normal**. If you have Embedded Coder installed, the software can measure the coverage of code generated from models in SIL or PIL mode. For more information, see “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode” on page 19-6.

The following table identifies specific model objects that do not receive coverage in certain conditions.

Model object	Does not receive coverage...
Logical Operator block	When the Operator parameter specifies XOR or NXOR and there are two or more inputs.
Model block	When the Simulation mode parameter specifies Accelerator . Coverage for Model blocks is the sum of the coverage data for the contents of the referenced model.
Subsystem block	When the Read/Write Permissions parameter is set to NoReadOrWrite .
Stateflow chart MATLAB Function block	When debugging/animation is not enabled for the model or object.
Virtual Blocks	Virtual blocks do not receive model coverage. For more information, see “Nonvirtual and Virtual Blocks”.

Setting Model Coverage Options

- “Specify Model Coverage Options” on page 18-2
- “Access, Manage, and Accumulate Coverage Results” on page 18-12
- “Cumulative Coverage Data” on page 18-22

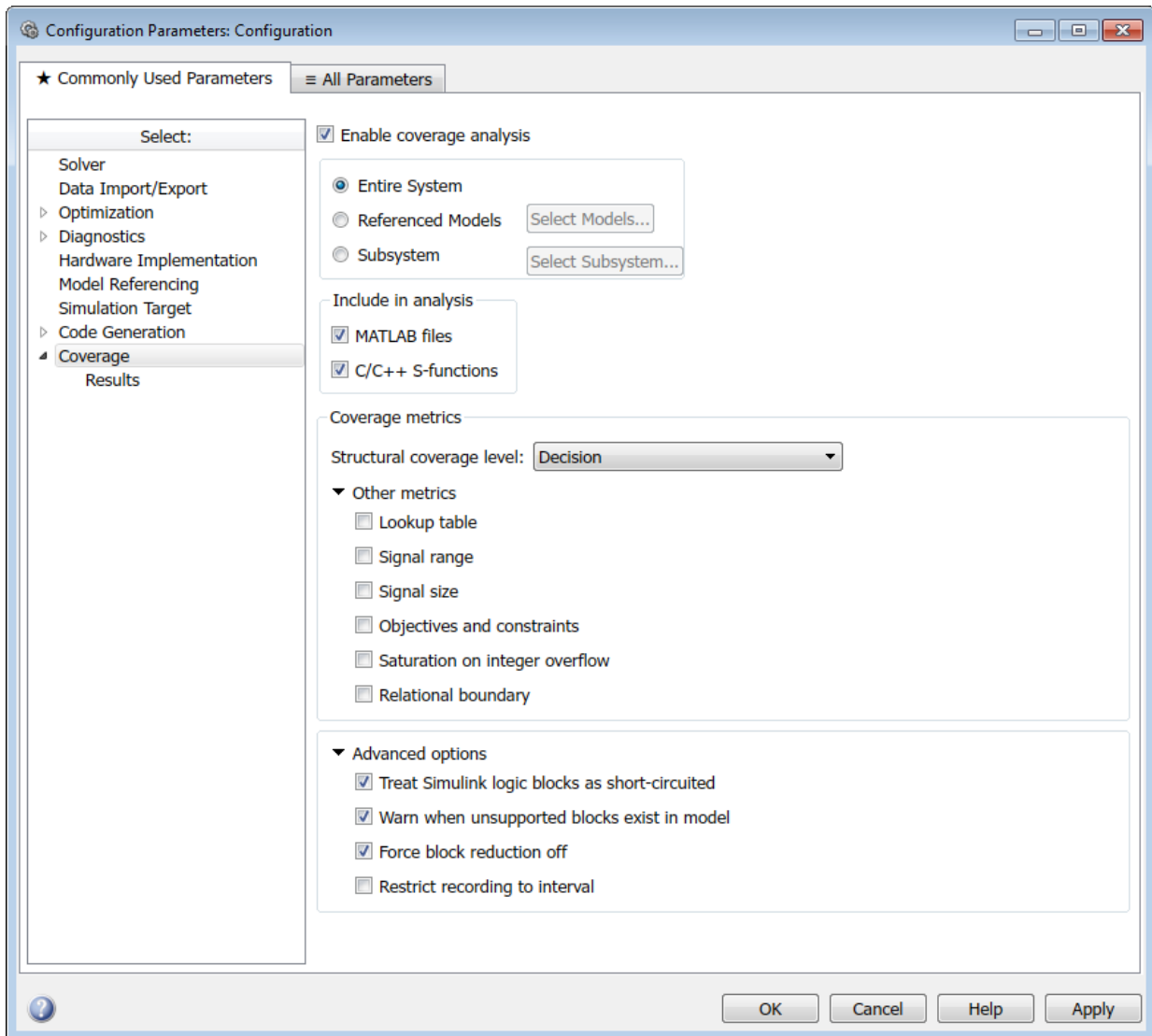
Specify Model Coverage Options

Before starting a model coverage analysis, you specify several model coverage recording options. In the Simulink Editor, select **Analysis > Coverage > Settings**.

In this section...
“Coverage Pane” on page 18-2
“Results Pane” on page 18-9

Coverage Pane

On the **Coverage** pane in the Configuration Parameters dialog box, set the options for the model coverage calculated during simulation.



Enable coverage analysis

Gather specified coverage results during simulation and report the model coverage. When you select **Enable coverage analysis**, these sections become available:

- “Scope of analysis” on page 18-4
- “Include in analysis” on page 18-6
- “Coverage metrics” on page 18-7
- “Advanced options” on page 18-7

Scope of analysis

Specifies the systems for which the software gathers and reports coverage data. The options are:

- “Entire System” on page 18-4
- “Referenced Models” on page 18-4
- “Subsystem” on page 18-5

You must select **Enable coverage analysis** to specify the scope of analysis.

Entire System

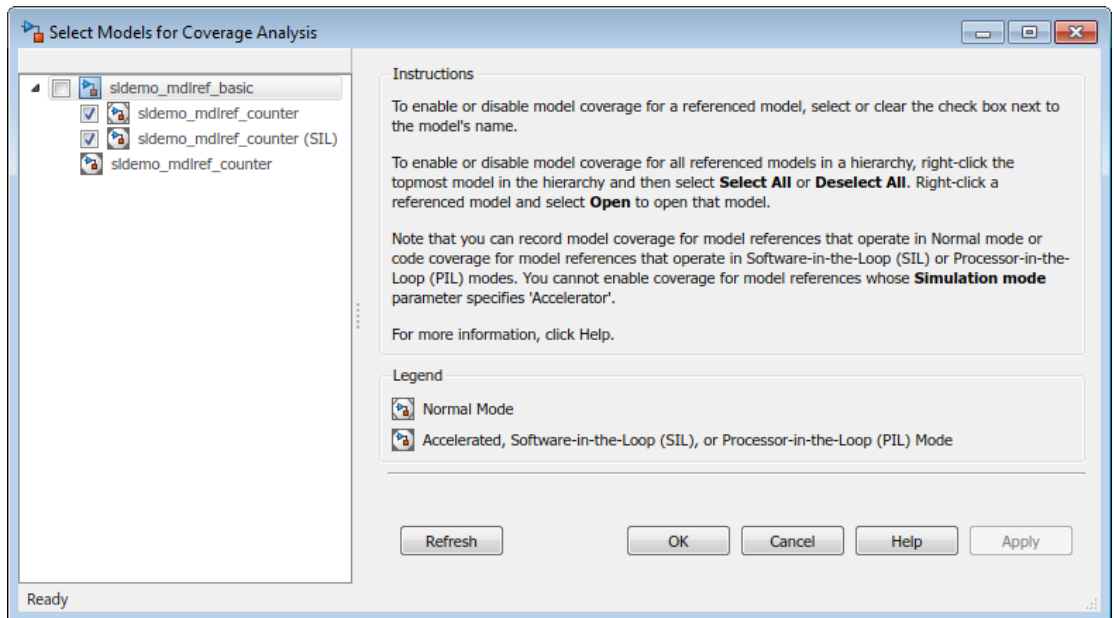
By default, generates coverage data for the entire system. The coverage results include the top-level model and all supported subsystems and model references.

Referenced Models

Coverage analysis records the model coverages for the referenced models that you select. By default, generates coverage data for all referenced models where the simulation mode of the Model block is set to Normal, Software-in-the-loop (SIL), or Processor-in-the-loop (PIL).

To specify the referenced models for which the Simulink Verification and Validation software records coverage data:

- 1 In the Configuration Parameters dialog box, on the **Coverage** pane, select **Enable coverage analysis**.
- 2 Click **Select Models**.



- 3 In the Select Models for Coverage Analysis dialog box, select the referenced models for which you want to record coverage. You can also select the top-level model.

The icon next to the model name indicates the simulation mode for that referenced model. You can select only referenced models whose simulation mode is set to Normal, SIL, or PIL.

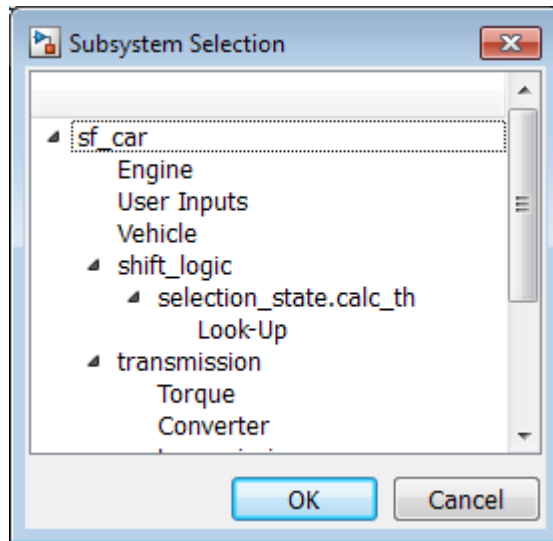
If you have multiple Model blocks that reference the same model and whose simulation modes are the same, selecting the check box for that model selects the check boxes for all instances of that model with the same simulation mode.

- 4 To close the Select Models for Coverage Analysis dialog box and return to the Configuration Parameters dialog box, click **OK**.

Subsystem

Coverage analysis records model coverage during simulation for the subsystem that you select. By default, generates coverage data for the entire model. To restrict coverage reporting to a particular subsystem:

- 1 In the Configuration Parameters dialog box, on the **Coverage** pane, select **Enable coverage analysis**.
- 2 Click **Select Subsystem**.



- 3 In the Subsystem Selection dialog box, select the subsystem for which you want to enable coverage reporting and click **OK**.

Include in analysis

The **Include in analysis** section contains two options:

- **MATLAB files** enables coverage for any external functions called by MATLAB functions in your model. You can define MATLAB functions in MATLAB Function blocks or in Stateflow charts.

To select the **Coverage for MATLAB files** option, you must select **Enable coverage analysis**.

- **C/C++ S-functions** enables coverage for C/C++ S-Function blocks in your model. Coverage metrics are reported for the S-Function blocks and the C/C++ code in those blocks. For more information, see “Generate Coverage Report for S-Function” on page 20-41.

You must select **Enable coverage analysis** to select the **Coverage for S-Functions** option.

Coverage metrics

Select the structural coverage level and other types of test case coverage analysis that you want the tool to perform (see “Types of Model Coverage” on page 16-3). The Simulink Verification and Validation software gathers and reports those types of coverage for the subsystems, models, and referenced models that you specify.

The structural coverage levels are listed in order of strictness of test case coverage analysis:

- **Block Execution** — Enables “Execution Coverage (EC)” on page 16-3
- **Decision** — Enables “Execution Coverage (EC)” on page 16-3 and “Decision Coverage (DC)” on page 16-3
- **Condition Decision** — Enables “Execution Coverage (EC)” on page 16-3, “Decision Coverage (DC)” on page 16-3, and “Condition Coverage (CC)” on page 16-3
- **Modified Condition Decision (MCDC)** — enables “Execution Coverage (EC)” on page 16-3, “Decision Coverage (DC)” on page 16-3, “Condition Coverage (CC)” on page 16-3, and “Modified Condition/Decision Coverage (MCDC)” on page 16-4

Coverage metrics also includes **Other metrics**:

- “Lookup Table Coverage” on page 16-5
- “Signal Range Coverage” on page 16-6
- “Signal Size Coverage” on page 16-6
- “Objectives and Constraints Coverage” on page 16-7
- “Saturate on Integer Overflow Coverage” on page 16-8
- “Relational Boundary Coverage” on page 16-8

You must select **Enable coverage analysis** to select the coverage metrics.

Advanced options

Advanced options for model coverage collection:

- “Treat Simulink logic blocks as short-circuited” on page 18-8
- “Warn when unsupported blocks exist in model” on page 18-8
- “Force block reduction off” on page 18-8

- “Restrict recording to interval” on page 18-9

Treat Simulink logic blocks as short-circuited

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

Note: If you disable this option, Simulink Verification and Validation does not combine logic block cascades for MCDC coverage. Instead, Simulink Verification and Validation analyses each block individually for MCDC coverage. For more information, see “Analyzing MCDC for Cascaded Logic Blocks”.

If you select this option and logic blocks are short-circuited while collecting model coverage, it is possible that you are not able to achieve 100% coverage for that block.

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

Note: Some test cases that do not achieve full MCDC coverage for non-short-circuited logic expressions do achieve full coverage for short-circuited expressions.

Warn when unsupported blocks exist in model

If the model contains blocks that require coverage analysis but are not currently covered by the tool, provides a warning at the end of the simulation.

Force block reduction off

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

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

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

If you select the **Block reduction** parameter and do not set **Force block reduction off**, the coverage report lists the reduced blocks that do not receive coverage analysis.

The model coverage report identifies any reduced blocks. For an example of a reduced blocks report, see “Block Reduction” on page 21-36.

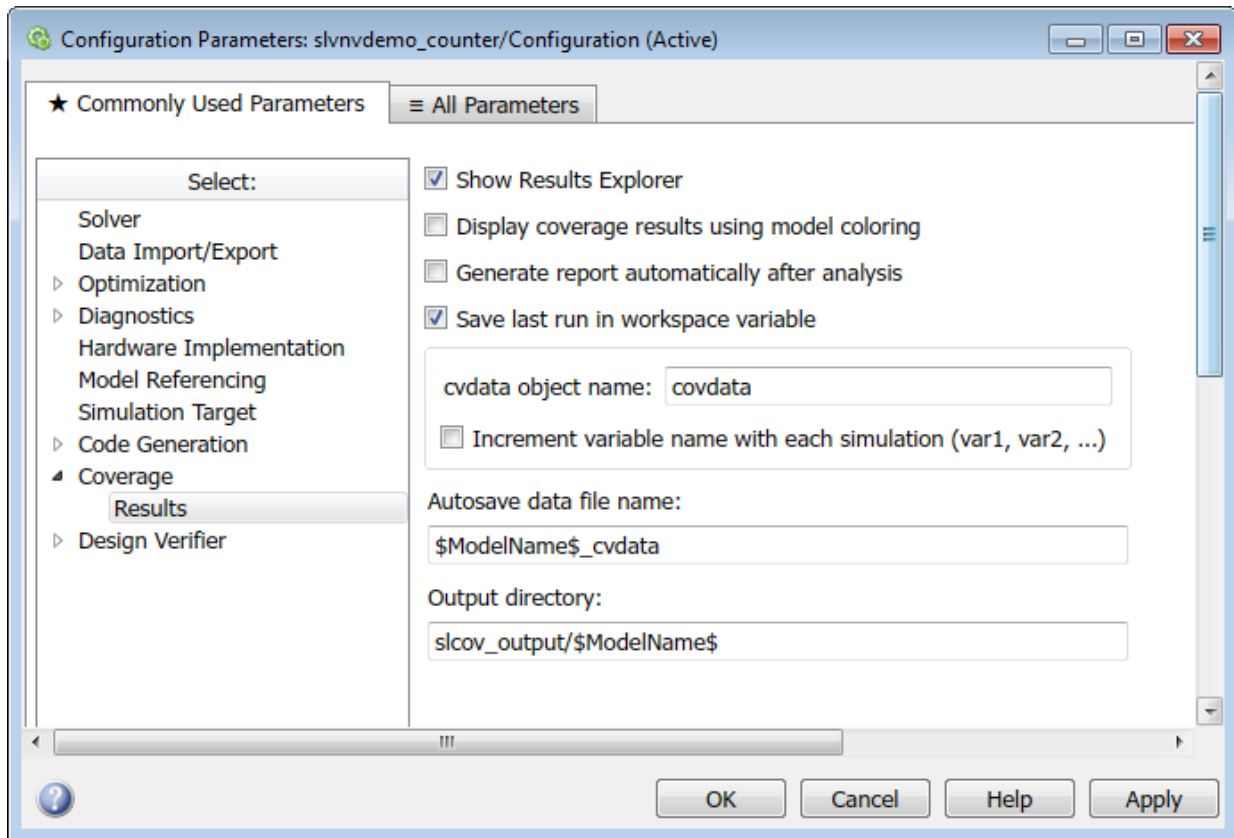
Restrict recording to interval

To record model coverage only inside a specified simulation time interval, select **Restrict recording to interval** and define a **Start time** and **Stop time**. Model coverage is not recorded for simulation times outside **Start time** and **Stop time**. If your simulation starts at a time greater than or equal to **Stop time**, model coverage is not recorded.

For example, you might want to restrict model coverage recording if your model has transient effects early in simulation, or if you need model coverage reported only for a particular model operation.

Results Pane

On the **Coverage > Results** pane in the Configuration Parameters dialog box, select the destination for model coverage results. You must select **Enable coverage analysis** on the **Coverage** pane to set the **Coverage > Results** pane options.



Show Results Explorer

After simulation, shows the results explorer.

Display coverage results using model coloring

After simulation, colors model objects according to their level of coverage. Objects highlighted in light green receive full coverage during testing. Objects highlighted in light red receive incomplete coverage. See “View Coverage Results in a Model” on page 20-7.

Generate report automatically after analysis

Specifies whether to open a generated HTML coverage report in a MATLAB browser window at the end of model simulation.

Save last run in workspace variable

Saves the results of the last simulation run in a `cvdata` object in the workspace. Specify the workspace variable name in **cvdata object name**.

cvdata object name

Name of the workspace variable where the results of the last simulation run are saved. You must select **Save last run in workspace variable** to specify the `cvdata` object name.

Increment variable name with each simulation (var1, var2, ...)

Appends numerals to the workspace variable names for each new result so that earlier results are not overwritten. You must select **Save last run in workspace variable** to enable this option.

Autosave data file name

Name of file to which coverage data results are saved. The default name is `$ModelName$_cvdata`. `$ModelName$` is the name of the model.

Output directory

The folder where the coverage data is saved. The default location is `slcov_output/$ModelName$` in the current folder. `$ModelName$` is the name of the model.

Related Examples

- “Access, Manage, and Accumulate Coverage Results” on page 18-12

Access, Manage, and Accumulate Coverage Results

After you “Specify Model Coverage Options” on page 18-2 and record coverage results for your model, you can use the Results Explorer to access, manage, and accumulate the coverage data that you record. After you accumulate the coverage results you need, you can then create a “Top-Level Model Coverage Report” on page 21-12 or “Export Model Coverage Web View” on page 21-47 using your accumulated coverage data.

In this section...
“Accessing Coverage Data from the Results Explorer” on page 18-12
“Managing Coverage Data from the Results Explorer” on page 18-19
“Accumulating Coverage Data from the Results Explorer” on page 18-19

Accessing Coverage Data from the Results Explorer

In the Configuration Parameters dialog box, on the **Coverage** > “Results Pane” on page 18-9, you can specify whether to show the Results Explorer after each simulation. You can also specify whether to generate an HTML report after each simulation. If you do not specify to show the Results Explorer or generate an HTML report, you can access the Results Explorer by selecting **Analysis > Coverage > Open Results Explorer** after you record coverage for a model. The Coverage Results Explorer opens to show the most recent coverage run:

The screenshot shows a software window titled "Coverage Results: sf_car". On the left is a tree view with "sf_car" expanded, showing "Settings", "Current Cumulative Data", "Run 1" (selected), and "Data Repository". The main area is titled "Coverage Data" and contains the following information:

- Model version: 1.120
- Author: The MathWorks, Inc.
- Started execution: 28-Jun-2016 14:03:56
- File name: sf_car_cvdata
- Description: (empty text box)
- Tag: Run 1
- Summary: (empty text box)

Below the summary is a table titled "Model Hierarchy/Complexity":

		Decision	Condition	MCDC	TBL	Execution
1. sf_car	32	79%	75%	50%	27%	100%
2. . . . Engine		NA	NA	NA	11%	100%
3. . . . Vehicle		NA	NA	NA	NA	100%
4. . . . shift_logic	26	78%	75%	50%	17%	100%
5. SF: shift_logic	25	78%	75%	50%	17%	100%
6. SF: gear_state	9	69%	NA	NA	NA	NA
7. SF:	16	86%	75%	50%	17%	100%

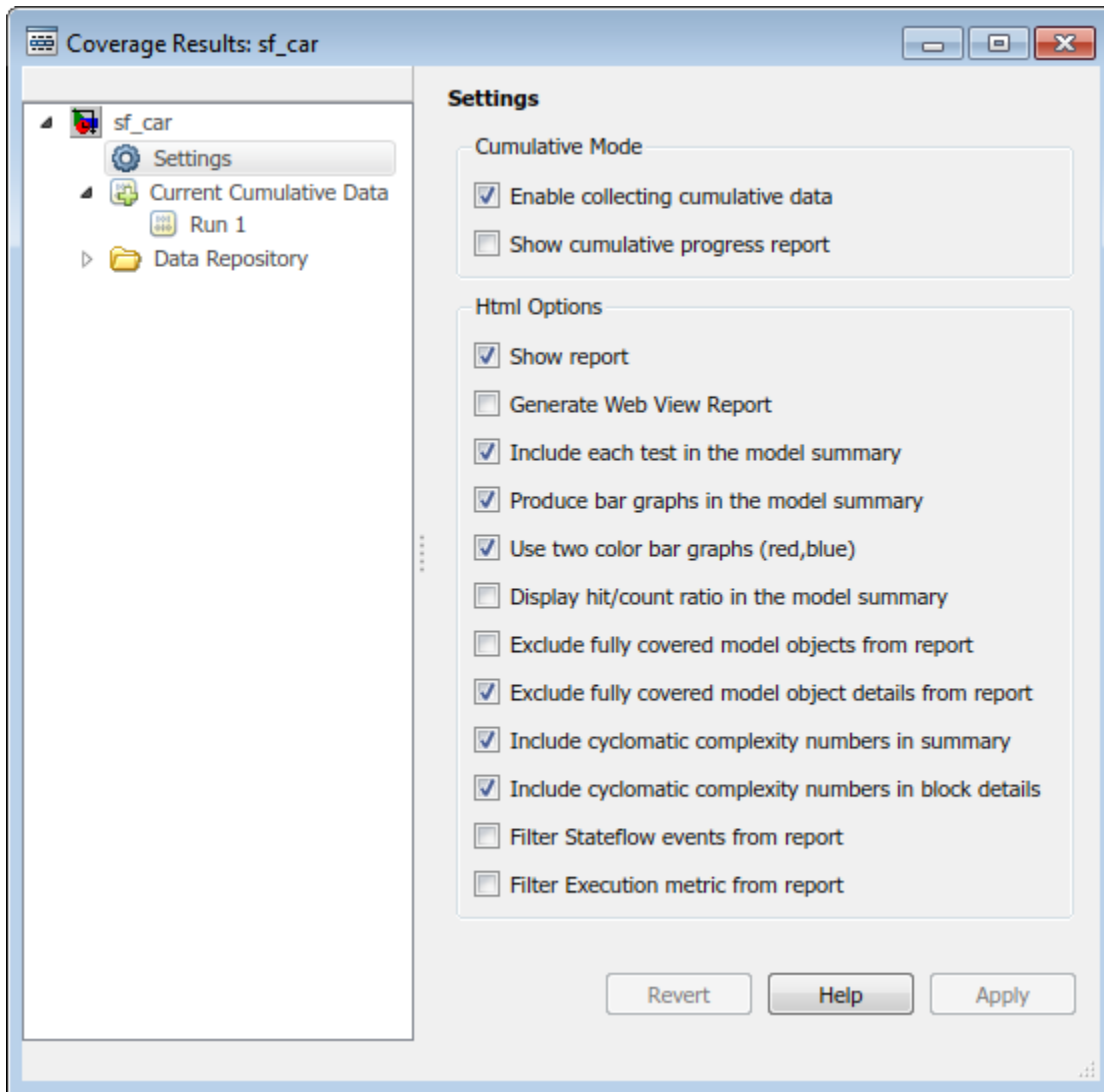
At the bottom of the window, there are two links: "Generate report" and "Highlight model with coverage results", and three buttons: "Revert", "Help", and "Apply".

You can view the current data results summary from within the Results Explorer or click **Generate Report** to create a full coverage report. If you do not make any changes to your model after you record coverage, you do not need to resimulate the model to generate a new coverage report. For more information on model coverage reports, see “Top-Level Model Coverage Report” on page 21-12.

Click **Highlight model with coverage results** to provide highlighted results in your model that allow you to quickly see coverage results for model objects. For more information, see “Overview of Model Coverage Highlighting” on page 20-7.

Settings

In the coverage Results Explorer, you can access the data and reporting settings for your coverage data. To access these settings, click **Settings**.



Option	Description
Enable collecting cumulative data	Accumulates model coverage results from successive simulations, by default. You specify the name and output folder

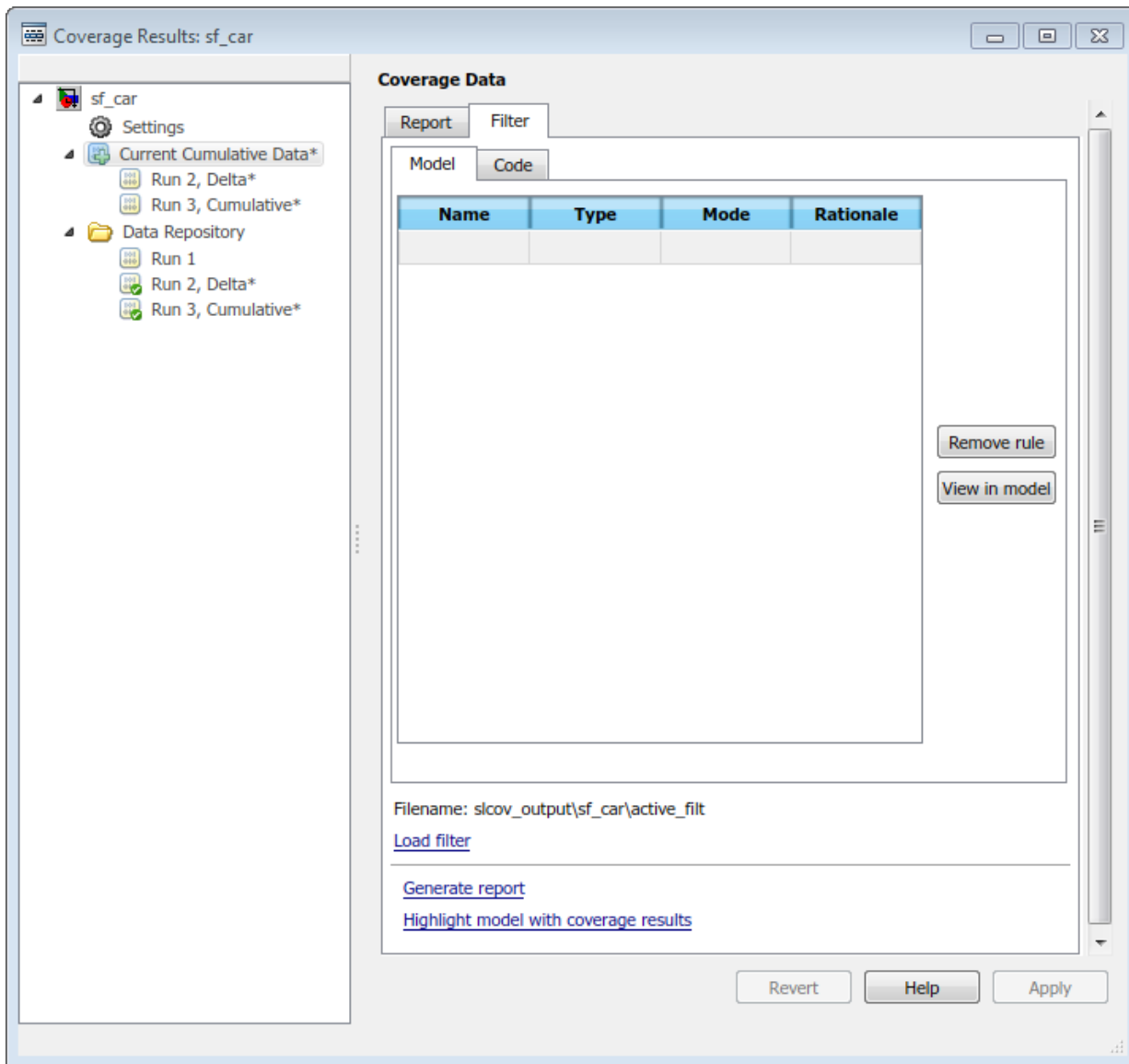
Option	Description
	of the .cvt file in the Configuration Parameters dialog box, on the “Results Pane” on page 18-9. For more information, see “.Cumulative Coverage Data” on page 18-22
Show cumulative progress report	Shows the Current Run coverage results, the Delta of coverage compared to the previous cumulative data, and the total Cumulative data from all current cumulative data separately in the coverage reports. If you do not select this option, only the total Cumulative data from all current cumulative data are shown.
Show report	Opens a generated HTML coverage report in a MATLAB browser window at the end of model simulation. For more information, see “Top-Level Model Coverage Report” on page 21-12.
Generate Web View Report	Opens a generated Model Coverage Web View in a MATLAB browser window at the end of model simulation. For more information, see “Export Model Coverage Web View” on page 21-47.
Include each test in the model summary	At the top of the HTML report, the model hierarchy table includes columns listing the coverage metrics for each test. If you do not select this option, the model summary reports only the total coverage.
Produce bar graphs in the model summary	Causes the model summary to include a bar graph for each coverage result for a visual representation of the coverage.
Use two color bar graphs (red, blue)	Red and blue bar graphs are displayed in the report instead of black and white bar graphs.

Option	Description
Display hit/count ratio in the model summary	Reports coverage numbers as both a percentage and a ratio, for example, 67% (8/12).
Exclude fully covered model objects from report	The coverage report includes only model objects that the simulation does not cover fully, useful when developing tests, because it reduces the size of the generated reports.
Exclude fully covered model object details from report	If you choose to include fully covered model objects in the report, the report does not include the details of the fully covered model objects
Include cyclomatic complexity numbers in summary	Includes the cyclomatic complexity (see “Types of Model Coverage” on page 16-3) of the model and its top-level subsystems and charts in the report summary. A cyclomatic complexity number shown in boldface indicates that the analysis considered the subsystem itself to be an object when computing its complexity. Boldface text can occur for atomic and conditionally executed subsystems and Stateflow Chart blocks.
Include cyclomatic complexity numbers in block details	Includes the cyclomatic complexity metric in the block details section of the report.
Filter Stateflow events from report	Excludes coverage data on Stateflow events.
Filter Execution metric from report	Excludes coverage data on Execution metrics

Creating and Managing Filters

You can create, load, or edit filters for the current coverage data from within the Results Explorer.

- 1 Open the Results Explorer.
- 2 Click the **Current Cumulative Data**.
- 3 Click the **Filter** tab.



For more information on filtering model objects, see “Creating and Using Coverage Filters”.

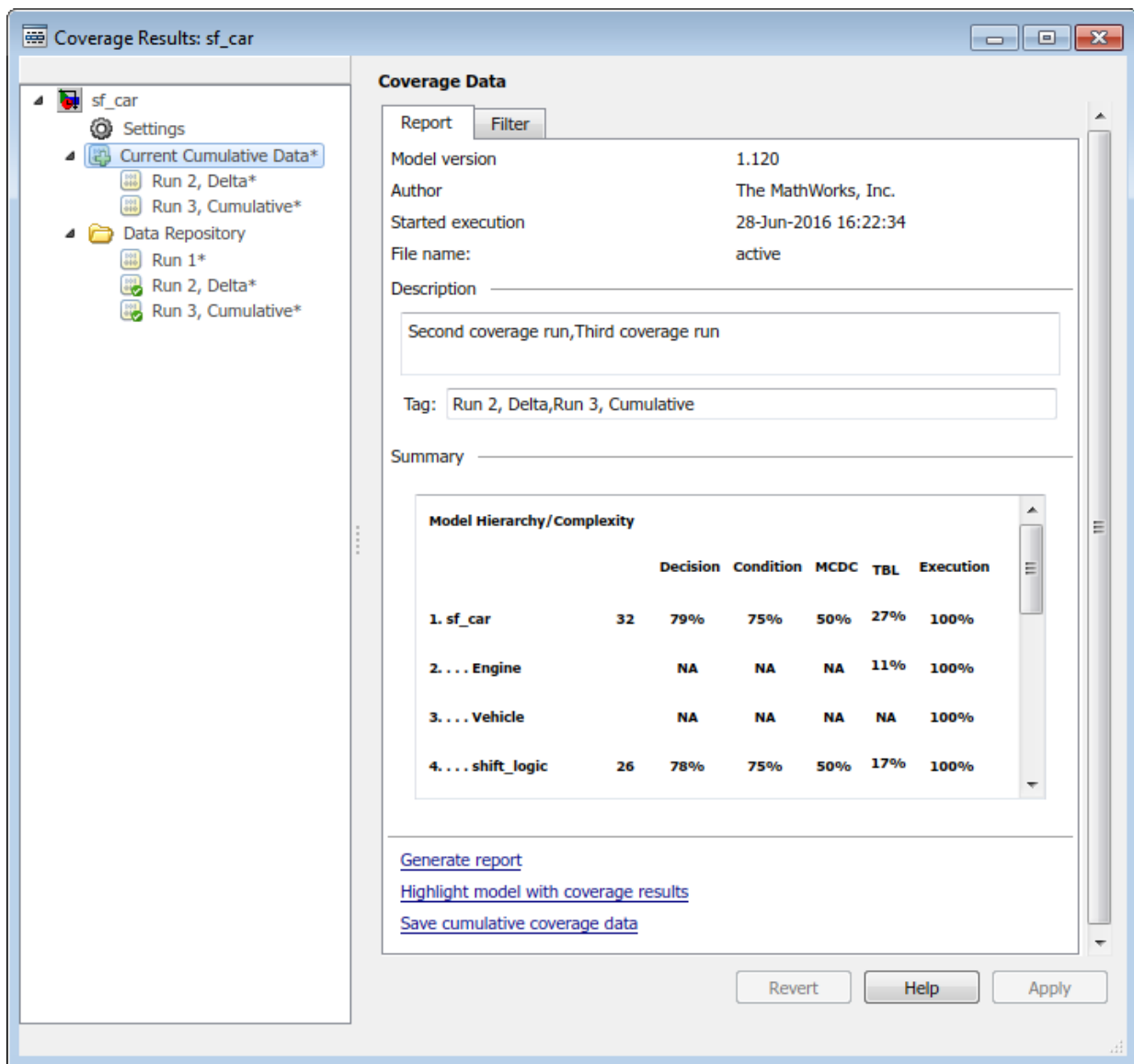
Managing Coverage Data from the Results Explorer

After you record coverage for a model, you can manage the coverage data from the Results Explorer. To view coverage data details, under **Current Cumulative Data**, click the coverage data of interest. You can edit the description and tags for each run. Before you leave the coverage data details view, click **Apply** to apply your changes. Otherwise, the changes are reverted.

When you apply changes to coverage data, such as adding descriptions and tags, the data shows an asterisk next to its icon. To save these changes, right-click the data and click **Save modified coverage data**.

Accumulating Coverage Data from the Results Explorer

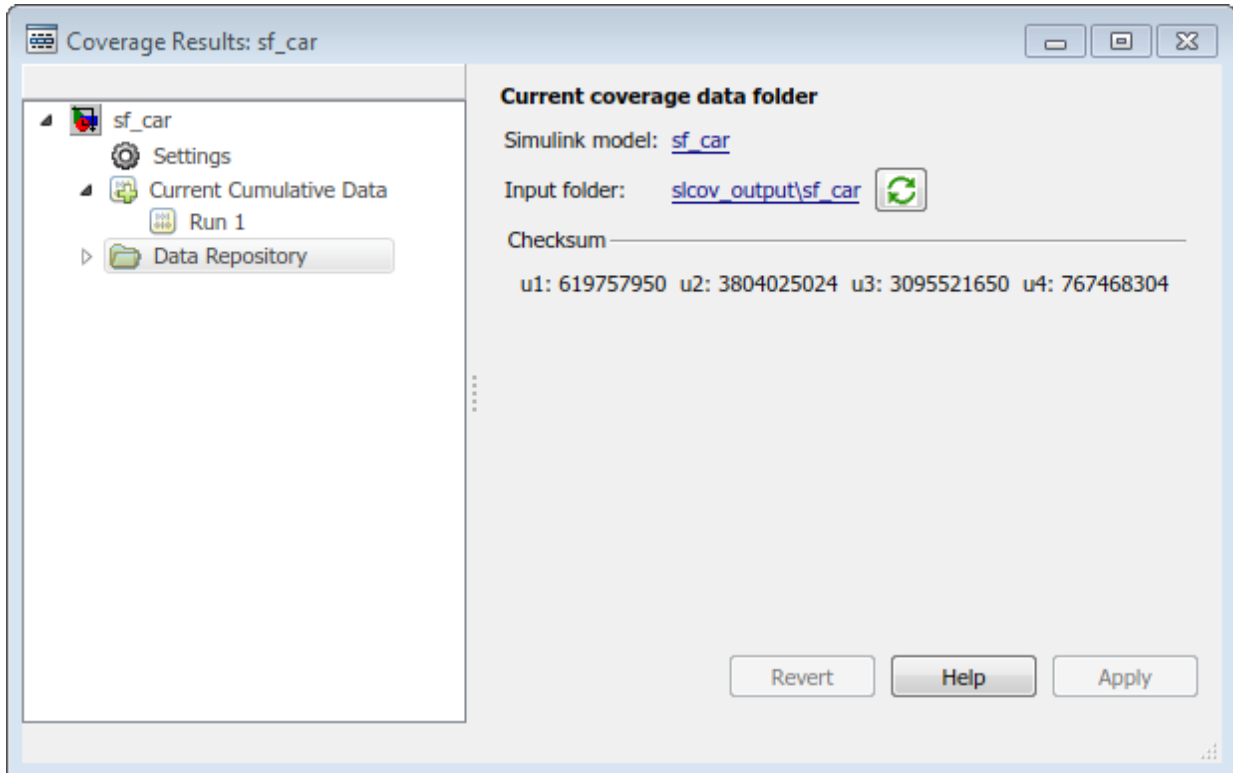
If you record multiple coverage runs, each run is listed separately in the Data Repository. You can drag and drop runs from the Data Repository to the Current Cumulative Data to manage which runs to include in the cumulative coverage data. Alternatively, right-click runs in the Data Repository or the Current Cumulative Data to include or exclude them in the cumulative coverage data.




To save the current cumulative data set to a .cvt file, click **Save cumulative coverage data**. Alternatively, you can right-click the **Current Cumulative Data** and select **Save cumulative coverage data**.

Load Existing Coverage Data

The Data Repository contains the coverage data, which is saved to the Input folder. You specify the Input folder on the **Configuration Parameters dialog box > Coverage > “Results Pane”** on page 18-9, in the **Output directory** field.



To synchronize the data in the input folder and the data in the Data Repository, click **Synchronize with the current coverage data folder** .

To load existing coverage data to the Data Repository:

- 1 Right-click the **Data Repository**.
- 2 Select **Load coverage data**.
- 3 Select existing coverage data for the current model and click **Open**.

Cumulative Coverage Data

On the **Coverage > Results** pane in the Configuration Parameters dialog box, if you select **Enable cumulative data collection** and **Save cumulative results in workspace variable**, 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 data resets.

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

Whenever you report on more than one single simulation, the coverage displayed for truth tables and lookup-table maps is based on the total coverage of all the reported runs. For cumulative reports, this information includes all the simulations where cumulative results are stored. For more information about managing cumulative results, see “Access, Manage, and Accumulate Coverage Results” on page 18-12.

You can make cumulative coverage results persist between MATLAB sessions. The `cvload` parameter `RESTORETOTAL` must be 1 to restore cumulative results. At the end of the sessions, use `cvsave` to save results to a file. At the beginning of the next session, use `cvload` to load the results.

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

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

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

Code Coverage

Types of Code Coverage

If you have Embedded Coder, Simulink Verification and Validation can perform several types of code coverage analysis for models in software-in-the-loop (SIL) mode, processor-in-the-loop (PIL) mode, and for the code within supported S-Function blocks.

In this section...

“Statement Coverage for Code Coverage” on page 19-2

“Condition Coverage for Code Coverage” on page 19-3

“Decision Coverage for Code Coverage” on page 19-3

“Modified Condition/Decision Coverage (MCDC) for Code Coverage” on page 19-4

“Cyclomatic Complexity for Code Coverage” on page 19-4

“Relational Boundary for Code Coverage” on page 19-5

Statement Coverage for Code Coverage

Statement coverage determines the number of source code statements that execute when the code runs. Use this type of coverage to determine whether every statement in the program has been invoked at least once.

Statement coverage = (Number of executed statements / Total number of statements) * 100

Statement Coverage Example

This code snippet contains five statements. To achieve 100% statement coverage, you need at least three test cases. Specifically, tests with positive x values, negative x values, and x values of zero.

```
if (x > 0)
    printf( "x is positive" );
else if (x < 0)
    printf( "x is negative" );
else
    printf( "x is 0" );
```

Condition Coverage for Code Coverage

Condition coverage analyzes statements that include conditions in source code. Conditions are C/C++ Boolean expressions that contain relation operators (<, >, <=, or >=), equation operators (!= or ==), or logical negation operators (!), but that do not contain logical operators (&& or | |). This type of coverage determines whether every condition has been evaluated to all possible outcomes at least once.

Condition coverage = (Number of executed condition outcomes / Total number of condition outcomes) *100

Condition Coverage Example

In this expression:

```
y = x<=5 && x!=7;
```

there are these conditions:

```
x<=5
```

```
x!=7
```

Decision Coverage for Code Coverage

Decision coverage analyzes statements that represent decisions in source code. Decisions are Boolean expressions composed of conditions and one or more of the logical C/C++ operators && or | |. Conditions within branching constructs (if/else, while, do-while) are decisions. Decision coverage determines the percentage of the total number of decision outcomes the code exercises during execution. Use this type of coverage to determine whether all decisions, including branches, in your code are tested.

Decision coverage = (Number of executed decision outcomes / Total number of decision outcomes) *100

Decision Coverage Example

This code snippet contains three decisions:

```
y = x<=5 && x!=7;    // decision #1

if( x > 0 )          // decision #2
    printf( "decision #2 is true" );
else if( x < 0 && y ) // decision #3
```

```

    printf( "decision #3 is true" );
else
    printf( "decisions #2 and #3 are false" );

```

Modified Condition/Decision Coverage (MCDC) for Code Coverage

Modified condition/decision coverage (MCDC) is the extent to which the conditions within decisions are independently exercised during code execution.

- All conditions within decisions have been evaluated to all possible outcomes at least once.
- Every condition within a decision independently affects the outcome of the decision.

MCDC coverage = (Number of conditions evaluated to all possible outcomes affecting the outcome of the decision / Total number of conditions within the decisions) *100

Modified Condition/Decision Coverage Example

For this decision:

```
X || ( Y && Z )
```

the following set of test cases delivers 100% MCDC coverage.

	X	Y	Z
Test case #1	0	0	1
Test case #2	0	1	0
Test case #3	0	1	1
Test case #4	1	0	1

Cyclomatic Complexity for Code Coverage

Cyclomatic complexity is a measure of the structural complexity of code that uses the McCabe complexity measure. To compute the cyclomatic complexity of code, code coverage uses this formula:

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

N is the number of decisions in the code. o_n is the number of outcomes for the n^{th} decision point. Code coverage adds 1 to the complexity number for each C/C++ function.

Coverage Example

For this code snippet:

```
void evalNum( int x ){  
    if (x > 0)  
        printf( "x is positive" );  
    else if (x < 0)  
        printf( "x is negative" );  
    else  
        printf( "x is 0" );  
}
```

the cyclomatic complexity is 3.

Relational Boundary for Code Coverage

Relational boundary code coverage examines code that has relational operations. Relational boundary code coverage metrics align with those for model coverage, as described in “Relational Boundary Coverage” on page 16-8. Fixed-point values in your model are integers during code coverage.

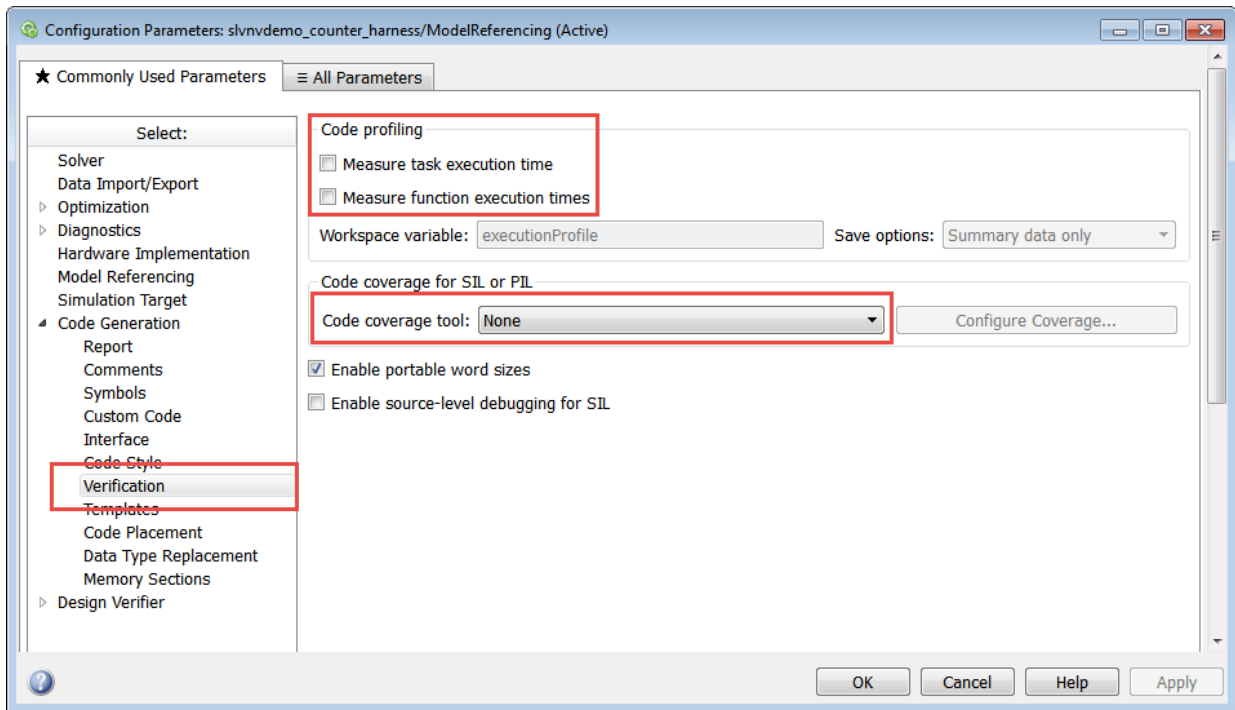
Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode

In this section...
“Requirements to Enable SIL or PIL Code Coverage for a Model” on page 19-6
“Conditions for Simulink Verification and Validation Code Coverage Measurement” on page 19-7
“Reviewing the Coverage Results for Models in SIL or PIL Mode” on page 19-7
“Limitations” on page 19-9

Requirements to Enable SIL or PIL Code Coverage for a Model

If you have both an Embedded Coder license and a Simulink Verification and Validation license, you can measure coverage for code generated from models in software-in-the-loop (SIL) mode or processor-in-the-loop (PIL) mode. The following configurations must apply for the parameters in **Code Generation Verification**:

- Under **Code profiling**, clear **Measure function execution times**.
- Under **Code coverage for SIL or PIL**, the selected **Code coverage tool** must be **None**.



Conditions for Simulink Verification and Validation Code Coverage Measurement

There are two workflows for measuring code coverage:

- The top model is in SIL mode or PIL mode. Measures code coverage for the top model depending on RecordCoverage. Also measures code coverage for referenced models, depending on CovModelRefEnable.
- The top model is in Normal mode and contains at least one reference model in SIL or PIL mode. Measures code coverage for the referenced model if CovModelRefEnable is 'on', 'all', or 'filtered' and RecordCoverage is 'off'.

Reviewing the Coverage Results for Models in SIL or PIL Mode

In the code coverage report, each hyperlink opens a report with more details on the coverage analysis for the model. The code coverage results in these reports are similar to

the coverage results for C/C++ code in S-function blocks, as described in “View Coverage Results for C/C++ Code in S-Function Blocks” on page 20-43. You can navigate from code coverage results to the associated model blocks by using the links within the detailed code coverage reports.

Link to model element

Logic block "[And](#)"

Metric	Coverage
Condition (C1)	100% (4/4) condition outcomes
MCDC (C1)	100% (2/2) conditions reversed the outcome

Code coverage summary

Covered expressions: [\(*rtu_upper >= rtb_input\) && rtb_inputGElower](#) (line 39) ← Link to code

Each detailed code coverage report also contains syntax highlighted code with coverage information.

Link to model element

```

34  /* Switch: '<Root>/Switch' incorporates:
35  * Logic: '<Root>/And'
36  * RelationalOperator: '<Root>/upper GE input'
37  * Switch: '<Root>/limit'
38  */
39  if ((*rtu_upper >= rtb_input) && rtb_inputGElower) {
40      *rty_output = rtb_input;
41  } else if (rtb_inputGElower) {

```

Link to code coverage result details

Decisions analyzed:

rtb_inputGElower	50%
false	5/5
true	0/5

Tooltip with code coverage results

```

    *limit' */
    per;
    wer;
    t>/Switch' */
    : '<Root>/Previous Output' */
51  localIDW->previousoutput_DSTATE = *rty_output;
52  }

```

Limitations

Coverage for models in SIL and PIL mode has these limitations:

- The model must meet the requirements listed in “Requirements to Enable SIL or PIL Code Coverage for a Model” on page 19-6.
- Code coverage results must not include external C/C++ files in read-only folders.

Related Examples

- “Software-in-the-Loop Code Coverage”

Coverage Collection During Simulation

- “Model Coverage Collection Workflow” on page 20-2
- “Create and Run Test Cases” on page 20-3
- “Modified Condition and Decision Coverage in Simulink Design Verifier” on page 20-4
- “View Coverage Results in a Model” on page 20-7
- “Model Coverage for Multiple Instances of a Referenced Model” on page 20-13
- “Model Coverage for MATLAB Functions” on page 20-23
- “Model Coverage for C and C++ S-Functions” on page 20-40
- “View Coverage Results for C/C++ Code in S-Function Blocks” on page 20-43
- “Model Coverage for Stateflow Charts” on page 20-48

Model Coverage Collection Workflow

Abstract

Develop effective tests with model coverage.

To develop effective tests with model coverage:

- 1 Develop one or more test cases for your model. (See “Create and Run Test Cases” on page 20-3.)
- 2 Run the test cases to verify model behavior.
- 3 Analyze the coverage reports produced by the Simulink Verification and Validation software.
- 4 Using the information in the coverage reports, modify the test cases to increase their coverage or add new test cases to cover areas not currently covered.
- 5 Repeat the preceding steps until you are satisfied with the coverage of your test suite.

Note: The Simulink Verification and Validation software comes with an example of model coverage to validate model tests. To step through the example, at the MATLAB command prompt, enter `simcovdemo`.

Create and Run Test Cases

Abstract

Create and run test cases using model coverage MATLAB commands `cvtest` and `cvsim`.

To create and run test cases, model coverage provides two MATLAB commands, `cvtest` and `cvsim`. The `cvtest` command creates test cases that the `cvsim` command runs. (See “Run Tests with `cvsim`” on page 23-5.)

You can also run the coverage tool interactively:

- 1 Open the `sldemo_fuelsys` model.
- 2 In the Simulink model window, select **Analysis > Coverage > Settings**.

The “Coverage Pane” on page 18-2 of the Configuration Parameters dialog box opens.

- 3 Select **Enable coverage analysis**, which enables the coverage settings.
- 4 Under **Coverage metrics**, select the types of coverage that you want to record in the coverage report.
- 5 Click **OK**.
- 6 In the Simulink Editor, select **Simulation > Run** to start simulating the model.

If you specify to report model coverage in the “Results Pane” on page 18-9 of the Configuration Parameters dialog box, Simulink Verification and Validation saves coverage data for the current run in the workspace object `covdata` and cumulative coverage data in `covCumulativeData`, by default. Simulink Verification and Validation also saves these results to a `.cvt` file by default. At the end of the simulation, the data appears in an HTML report that opens in a browser window. For more information on coverage data settings, see “Specify Model Coverage Options” on page 18-2.

Note: You cannot run simulations if you select both model coverage reporting and acceleration options. If you select **Simulation > Mode > Accelerator** in the Simulink Editor, Simulink does not record coverage.

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

Modified Condition and Decision Coverage in Simulink Design Verifier

There are some differences in modified condition and decision coverage (MCDC) reporting and test generation between Simulink Verification and Validation and Simulink Design Verifier .

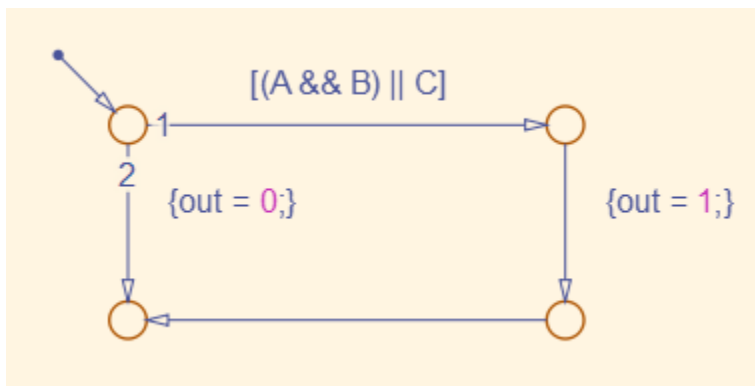
In this section...

“MCDC Definitions for Simulink Verification and Validation and Simulink Design Verifier” on page 20-4

“Cascaded Logic Blocks in Simulink Verification and Validation and Simulink Design Verifier” on page 20-6

MCDC Definitions for Simulink Verification and Validation and Simulink Design Verifier

There is a difference between the definition of modified condition and decision coverage for model coverage analysis in Simulink Verification and Validation and for test case generation analysis in Simulink Design Verifier. This difference can be seen in analysis results for logical expressions containing a mixture of AND and OR operators, as in this Stateflow transition.



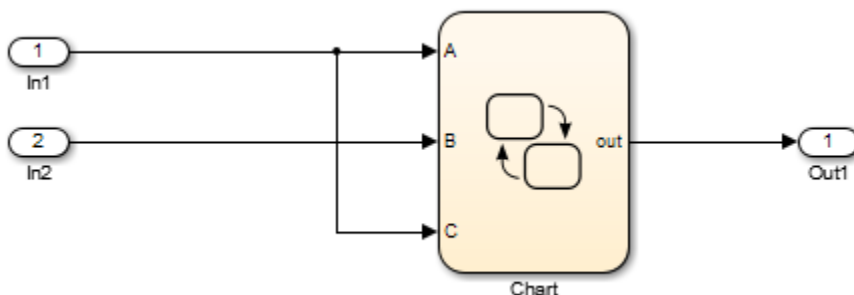
Given that A, B, and C are each separate inputs, there are five possible ways to evaluate the condition on the Stateflow transition, shown in the following table.

	A	B	C	(A && B) C
1	F	x	F	F
2	F	x	T	T
3	T	F	F	F
4	T	F	T	T
5	T	T	x	T

Satisfying MCDC for a Boolean variable requires a pair of condition evaluations, showing that a change in that variable alone changes the evaluation of the entire expression. In this example, MCDC can be satisfied for **C** with either the pair 1, 2 or the pair 3, 4. In both of those cases, the value of the expression changed because the value of **C** changed, while all other variable values stayed the same.

Each pair has a different set of values for **A** and **B** which are held constant, but each pair contains one evaluation where **C** and **out** are true and one evaluation where **C** and **out** are false. To satisfy MCDC for **C**, Simulink Design Verifier test generation analysis accepts any pair containing one evaluation of true values and one evaluation of false values for **C** and **out**. In this example, it accepts not only pair 1, 2 and pair 3, 4 but also pair 1, 4 and pair 2, 3. Simulink Verification and Validation model coverage analysis uses a strict form of MCDC that is satisfied only by pair 1, 2 or by pair 3, 4.

The preceding example assumes that **A**, **B**, and **C** are all separate inputs. When input **A** is constrained to be the same value as **C**, as in this model, only a subset of condition evaluations are possible.



This subset of condition evaluations for the Stateflow transition is shown in the following table.

	A	B	C	(A && B) C
1	F	x	F	F
4	T	F	T	T
5	T	T	x	T

Evaluations 2 and 3 are no longer possible, so neither pair 1, 2 nor pair 3, 4 is possible. As a result, MCDC for C can no longer be satisfied in Simulink Verification and Validation model coverage analysis. Since pair 1, 4 is still possible, however, Simulink Design Verifier test generation analysis reports that MCDC for C is satisfiable.

The complexity of MCDC analysis for logical expressions with a mixture of AND and OR operators causes this difference between results from Simulink Verification and Validation and Simulink Design Verifier. To minimize this effect, use the `IndividualObjectives` strategy for test generation analysis in Simulink Design Verifier.

Cascaded Logic Blocks in Simulink Verification and Validation and Simulink Design Verifier

When you generate tests in Simulink Design Verifier to satisfy MCDC objectives for a model containing Simulink logic block cascades, the “Treat Simulink logic blocks as short-circuited” on page 18-8 option for your model impacts the tests that Simulink Design Verifier generates. If you enable “Treat Simulink logic blocks as short-circuited” on page 18-8, Simulink Design Verifier generates tests for each cascade of logic blocks, but not for individual blocks in the cascades. If you do not enable “Treat Simulink logic blocks as short-circuited” on page 18-8, Simulink Design Verifier generates tests for each individual block in the cascades, but not for the cascades of logic blocks. For more information on cascaded logic blocks, see “Logical Operator Cascade Patterns” and “Analyzing MCDC for Cascaded Logic Blocks”.

More About

- “MCDC”

View Coverage Results in a Model

In this section...

“Overview of Model Coverage Highlighting” on page 20-7

“Enable Coverage Highlighting” on page 20-8

“View Results in Coverage Display Window” on page 20-11

Overview of Model Coverage Highlighting

When you simulate a Simulink model, you can configure your model to provide visual results that allow you to see at a glance which objects recorded 100% coverage. After the simulation:

- In the model window, model objects are highlighted in certain colors according to what coverage was recorded:
 - Light green indicates that an object received full coverage during testing.
 - Light red indicates that an object received incomplete coverage.
 - Gray indicates that an object was filtered from coverage.
 - Objects with no color highlighting received no coverage.
- When you click a colored object, the Coverage Display Window provides details about the coverage recorded for that block. For subsystems and Stateflow charts, the Coverage Display Window lists the summary coverage for all objects in that subsystem or chart. For other blocks, the Coverage Display Window lists specific details about the objects that did not receive 100% coverage.

The simulation highlights blocks that received the following types of model coverage:

- “Execution Coverage (EC)” on page 16-3
- “Decision Coverage (DC)” on page 16-3
- “Condition Coverage (CC)” on page 16-3
- “Modified Condition/Decision Coverage (MCDC)” on page 16-4
- “Relational Boundary Coverage” on page 16-8
- “Saturate on Integer Overflow Coverage” on page 16-8

- “Objectives and Constraints Coverage” on page 16-7

Enable Coverage Highlighting

To enable the model coverage colored diagram display:

- 1 In the Simulink Editor, select **Analysis > Coverage > Settings** to open the **Coverage** pane of the Configuration Parameters dialog box.
- 2 In the **Coverage** pane of the Configuration Parameters dialog box, select **Enable coverage analysis** and then select **Entire System**.
- 3 Select MCDC as the **Structural coverage level**.
- 4 Select **Objectives and Constraints** under **Other metrics**.
- 5 On the **Results** pane of the Configuration Parameters dialog box, select **Display coverage results using model coloring**. This is the default setting.

After you have enabled the coverage coloring, simulate your model. In the model, you can see at a glance which objects received full, partial, or no coverage.

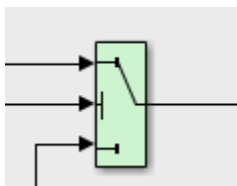
Highlighted Coverage Results

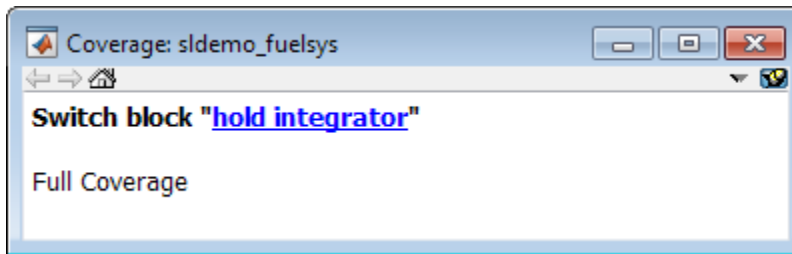
The following sections show examples of highlighted model objects in colors that correspond to the recorded coverage.

- “Green: Full Coverage” on page 20-8
- “Red: Partial Coverage” on page 20-9
- “Gray: Filtered Coverage” on page 20-11

Green: Full Coverage

In this example, the Switch block received 100% coverage, as indicated by the green highlighting and the information in the Coverage Display Window.

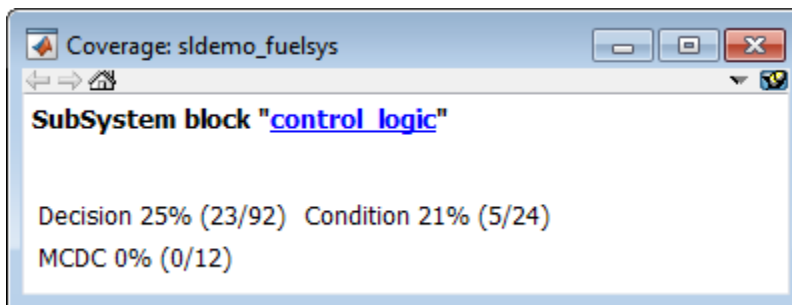
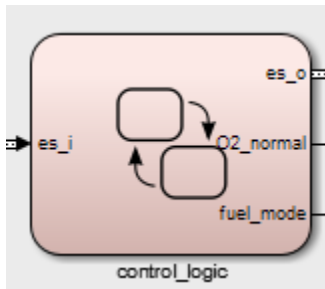




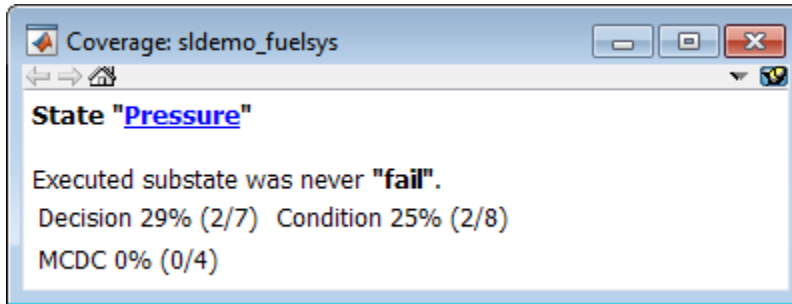
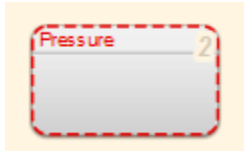
Red: Partial Coverage

In this example, the control_logic Stateflow chart received the following coverage:

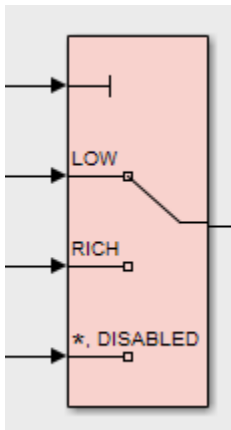
- Decision: 25%
- Condition: 21%
- MCDC: 0%

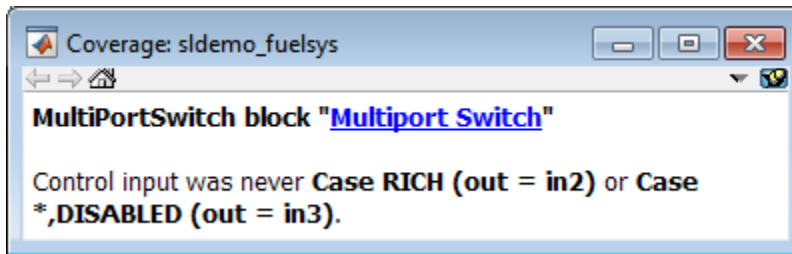


Inside the control_logic subsystem, the Pressure substate was never fail.



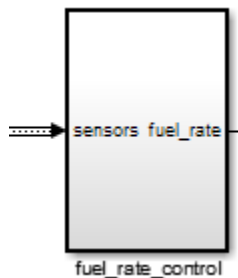
In the next example, in the Multiport Switch block, two of the data ports were never executed.





Gray: Filtered Coverage




In this example, the fuel_rate_control subsystem is highlighted in gray because it was filtered out of coverage recording.



View Results in Coverage Display Window

After simulating the model and recording coverage, by default, the Coverage Display Window is the top-most visible window. When you click an object that recorded coverage, the Coverage Display Window displays details of the coverage recorded during simulation.

In the Coverage Display Window, you can:

- Configure the window so it is not always the top-most visible window. Next to **Always on top**, click  and removing the check.
- Configure the window to display coverage information when you click an object that recorded coverage. Click  and select **Click**.
- Configure the window to display coverage information when you hover your cursor on an object that recorded coverage. Click  and select **Focus**.

- Close the window. Press **Alt+F4**.
- Close the window and remove highlighting on the model. Select **Display > Remove Highlighting**.

Abbreviations in Coverage Results

The Coverage Display Window shows results with abbreviations. You can view expanded results in the “Top-Level Model Coverage Report” on page 21-12.

Abbreviation	Meaning
CND	condition
ENBL	enable
FCALL	function call
IN	input
LGC	logic
LL	lower limit
MX ITER	maximum iterations exceeded
NA	not applicable
OffThresh	[switch] off threshold
OnThresh	switch on threshold
OUT	output
SO	saturate on integer overflow
TBL	lookup table
THRESH	threshold
TI	test interval
TO	test objective
TP	test point
TRIG	trigger
U	input
UL	upper limit
X	<ul style="list-style-type: none"> • integration result (Discrete-Time Integrator block) • slew rate (Rate Limiter block)

Model Coverage for Multiple Instances of a Referenced Model

In this section...

“About Coverage for Model Blocks” on page 20-13

“Record Coverage for Multiple Instances of a Referenced Model” on page 20-13

About Coverage for Model Blocks

Model blocks do not receive coverage directly; if you set the simulation mode of the Model block to `Normal`, `SIL`, or `PIL`, the Simulink Verification and Validation software records coverage for the model referenced from the Model block. If the simulation mode for the Model block is anything other than `Normal`, `SIL`, or `PIL`, the software does not record coverage for the referenced model.

Your Simulink model can contain multiple Model blocks with the same simulation mode that reference the same model. When the software records coverage, each instance of the referenced model can be exercised with different inputs or parameters, possibly resulting in additional coverage for the referenced model.

The Simulink Verification and Validation software records coverage for all instances of the referenced model with the same simulation mode and combines the coverage data for that referenced model in the final results.

Record Coverage for Multiple Instances of a Referenced Model

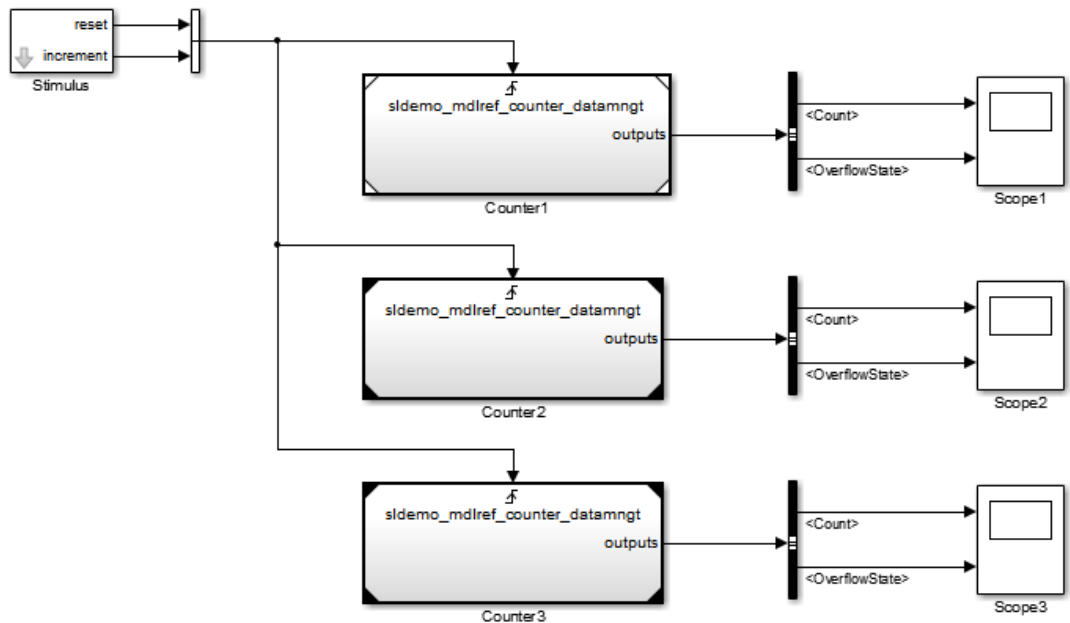
To see how this works, simulate a model twice. The first time, you record coverage for one Model block in `Normal` simulation mode. The second time, you record coverage for two Model blocks in `Normal` simulation mode. Both Model blocks reference the same model.

- “Record Coverage for the First Instance of the Referenced Model” on page 20-13
- “Record Coverage for the Second Instance of the Referenced Model” on page 20-19

Record Coverage for the First Instance of the Referenced Model

Record coverage for one Model block.

- 1 Open your top-level model. This example uses the following model:



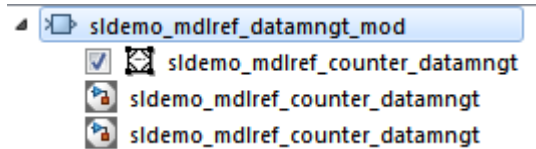
This model contains three Model blocks that reference the `sldemo_md1ref_counter_datamngt` example model. The corners of each Model block indicate the value of their **Simulation mode** parameter:

- Counter1 — Simulation mode: Normal
- Counter2 — Simulation mode: Accelerator
- Counter3 — Simulation mode: Accelerator

2 Configure your model to record coverage during simulation:

- a** In the model window, select **Analysis > Coverage > Settings**.
- b** On the **Coverage** pane of the Configuration Parameters dialog box, select:
 - **Enable coverage analysis**
 - **Referenced Models**
- c** Click **Select Models**. In the Select Models for Coverage Analysis dialog box, you can select only those referenced models whose simulation mode is

Normal, SIL, or PIL. In this example, only the first Model block that references `sldemo_mdhref_counter_datamngt` is available for recording coverage.

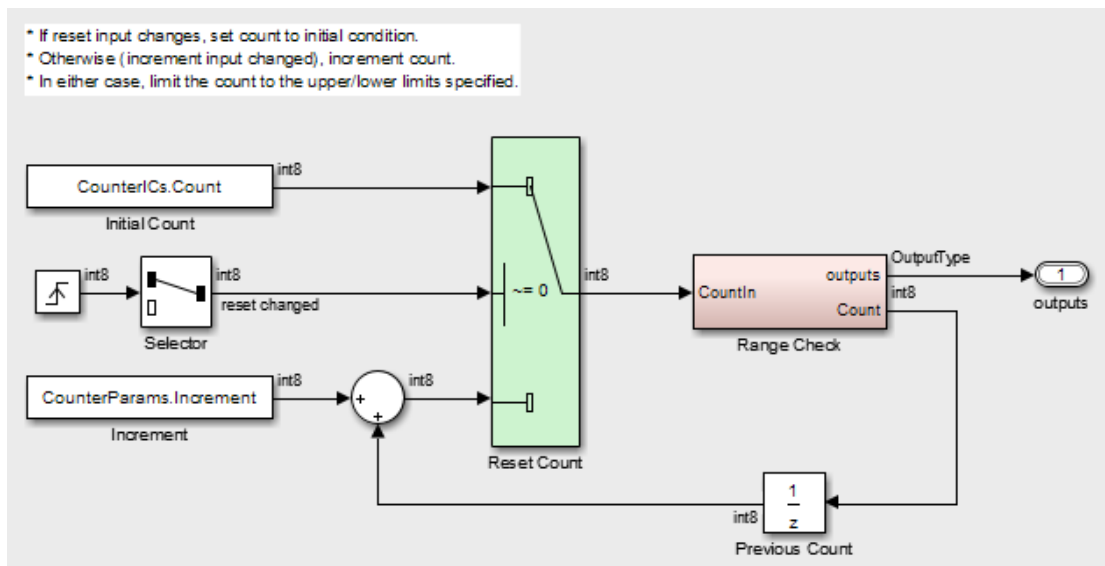


- d Click **OK** to exit the Select Models for Coverage Analysis dialog box.
- 3 Click **OK** to save your coverage settings and exit the Configuration Parameters dialog box.
- 4 Simulate your model.

When the simulation is complete, the HTML coverage report opens. In this example, the coverage data for the referenced model, `sldemo_mdhref_counter_datamngt`, shows that the model achieved 69% coverage.

- 5 Click the hyperlink in the report for the referenced model.

The detailed coverage report for the referenced model opens, and the referenced model appears with highlighting to show coverage results.



Note the following about the coverage for the Range Check subsystem in this example:

- The Saturate Count block executed 100 times. This block has four Boolean decisions. Decision coverage was 50%, because two of the four decisions were never recorded:
 - The decision `input > lower limit` was never false.
 - The decision `input >= upper limit` was never true.

Saturate block "[Saturate Count](#)"

Parent: [sldemo_mdhref_counter_datamngt/Range Check](#)

Uncovered Links: ➡

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


Decisions analyzed:

input > lower limit	50%
false	0/50
true	50/50
input >= upper limit	50%
false	50/50
true	0/50

- The `DetectOverflow` function executed 50 times. This script has five decisions. The `DetectOverflow` script achieved 60% coverage because two of the five decisions were never recorded:
 - The expression `count >= CounterParams.UpperLimit` was never true.
 - The expression `count > CounterParams.LowerLimit` was never false.

MATLAB Function "[DetectOverflow](#)"

Parent: [sldemo_mdref_counter_datamngt/Range Check/Detect Overflow](#)

Uncovered Links: 

Metric	Coverage
Cyclomatic Complexity	3
Decision	60% (3/5) decision outcomes

```

1 function result = DetectOverflow(count, CounterParams)
2 % DETECTOVERFLOW Check count
3 %#codegen
4
5 if (count >= CounterParams.UpperLimit)
6     result = s1DemoRangeCheck.UpperLimit;
7 elseif (count > CounterParams.LowerLimit)
8     result = s1DemoRangeCheck.InRange;
9 else
10    result = s1DemoRangeCheck.LowerLimit;
11 end
12

```

[#1: function result = DetectOverflow\(count, CounterParams\)](#)

Decisions analyzed:

function result = DetectOverflow(count, CounterParams)	100%
executed	50/50

[#5: if \(count >= CounterParams.UpperLimit\)](#)

Decisions analyzed:

if (count >= CounterParams.UpperLimit)	50%
false	50/50
true	0/50

[#7: elseif \(count > CounterParams.LowerLimit\)](#)

Decisions analyzed:

elseif (count > CounterParams.LowerLimit)	50%
false	0/50
true	50/50

Record Coverage for the Second Instance of the Referenced Model

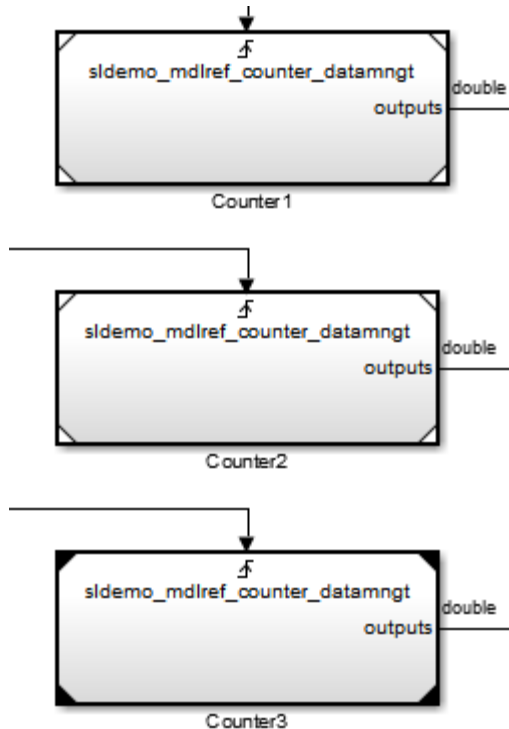
Record coverage for two Model blocks. Set the simulation mode of a second Model block to **Normal** and simulate the model. In this example, the Counter2 block adds to the coverage for the model referenced from both Model blocks.

- 1 In the Simulink Editor for your top-level model, right-click a second Model block and select **Block Parameters (ModelReference)**.

The Function Block Parameters dialog box opens.

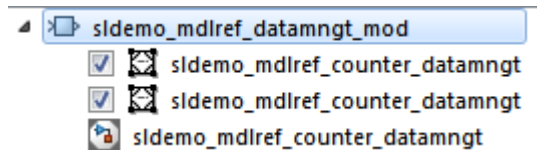
- 2 Set the **Simulation mode** parameter to **Normal**.
- 3 Click **OK** to save your change and exit the Function Block Parameters dialog box.

The corners of the Model block change to indicate that the simulation mode for this block is **Normal**, as in the example below.



- 4 To make sure that the software records coverage for both instances of this model:
 - a Select **Analysis > Coverage > Settings**.
 - b On the **Coverage** pane, select **Enable coverage analysis**.
 - c Select **Referenced Models** and click **Select Models**.

In the Select Models for Coverage Analysis dialog box, verify that both instances of the referenced model are selected. In this example, the list now looks like the following.



If you have multiple instances of a referenced model in **Normal** mode, you can choose to record coverage for all of them or none of them.

- d Click **OK** to close the Select Models for Coverage Analysis dialog box.
- 5 Simulate your model again.
- 6 When the simulation is complete, open the HTML coverage report.

In this example, the referenced model achieved 85% coverage. Note the following about the coverage data for the Range Check subsystem:

- The Saturate Count block executed 179 times. The simulation of the Counter2 block executed the Saturate Count block an additional 79 times, for a total of 179 executions.

The decision input `>= upper limit` was `true` 21 times during this simulation, compared to 0 during the first simulation. The fourth decision input `> lower limit` was still never `false`. Three out of four decisions were recorded during simulation, so this block achieved 75% coverage.

Saturate block "[Saturate Count](#)"

Parent: [sldemo_mdref_counter_datamngt/Range Check](#)

Uncovered Links: ➔

Metric	Coverage
Cyclomatic Complexity	2
Decision	75% (3/4) decision outcomes

Decisions analyzed:


input > lower limit	50%
false	0/79
true	79/79
input >= upper limit	100%
false	79/100
true	21/100

- The DetectOverflow function executed 100 times. The simulation of the Counter2 block executed the DetectOverflow function an additional 50 times.

The DetectOverflow function has five decisions. The expression `count >= CounterParams.UpperLimit` was `true` 21 times during this simulation, compared to 0 during the first simulation. The expression `count > CounterParams.LowerLimit` was never `false`. Four out of five decisions were recorded during simulation, so the DetectOverflow function achieved 80% coverage.

MATLAB Function "[DetectOverflow](#)"

Parent: [sldemo_mdref_counter_datamngt/Range Check/Detect Overflow](#)

Uncovered Links: 

Metric	Coverage
Cyclomatic Complexity	3
Decision	80% (4/5) decision outcomes

```

1 function result = DetectOverflow(count, CounterParams)
2 % DETECTOVERFLOW Check count
3 %#codegen
4
5 if (count >= CounterParams.UpperLimit)
6     result = SlDemoRangeCheck.UpperLimit;
7 elseif (count > CounterParams.LowerLimit)
8     result = SlDemoRangeCheck.InRange;
9 else
10    result = SlDemoRangeCheck.LowerLimit;
11 end
12

```

[#1: function result = DetectOverflow\(count, CounterParams\)](#)

Decisions analyzed:

function result = DetectOverflow(count, CounterParams)	100%
executed	100/100

[#5: if \(count >= CounterParams.UpperLimit\)](#)

Decisions analyzed:

if (count >= CounterParams.UpperLimit)	100%
false	79/100
true	21/100

[#7: elseif \(count > CounterParams.LowerLimit\)](#)

Decisions analyzed:

elseif (count > CounterParams.LowerLimit)	50%
false	0/79
true	79/79

Model Coverage for MATLAB Functions

In this section...

“About Model Coverage for MATLAB Functions” on page 20-23

“Types of Model Coverage for MATLAB Functions” on page 20-23

“How to Collect Coverage for MATLAB Functions” on page 20-25

“Examples: Model Coverage for MATLAB Functions” on page 20-26

About Model Coverage for MATLAB Functions

The Simulink Verification and Validation software simulates a Simulink model and reports model coverage data for the decisions and conditions of code in MATLAB Function blocks. Model coverage only supports coverage for MATLAB functions configured for code generation.

For example, consider the following `if` statement:

```
if (x > 0 || y > 0)
    reset = 1;
```

The `if` statement contains a decision with two conditions (`x > 0` and `y > 0`). The Simulink Verification and Validation software verifies that all decisions and conditions are taken during the simulation of the model.

Types of Model Coverage for MATLAB Functions

The types of model coverage that the Simulink Verification and Validation software records for MATLAB functions configured for code generation are:

- “Decision Coverage” on page 20-23
- “Condition and MCDC Coverage” on page 20-24
- “Simulink Design Verifier Coverage” on page 20-24
- “Relational Boundary Coverage” on page 20-25

Decision Coverage

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

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

Condition and MCDC Coverage

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

- **if** statement conditions
- **while** statement conditions

Simulink Design Verifier Coverage

The following MATLAB functions are active in code generation and in Simulink Design Verifier:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

When you specify the **Objectives and Constraints** coverage metric in the **Coverage** pane of the Configuration Parameters dialog box, the Simulink Verification and Validation software records coverage for these functions.

Each of these functions evaluates an expression *expr*, for example, `sldv.test(expr)`, where *expr* is a valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to **true**.

If *expr* is **true** for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Verification and Validation software reports coverage for that function as 0%.

For an example of coverage data for Simulink Design Verifier functions in a coverage report, see “Simulink Design Verifier Coverage” on page 21-45.

Relational Boundary Coverage

If the MATLAB function block contains a relational operation, the relational boundary coverage metric applies to this block.

If the MATLAB function block calls functions containing relational operations multiple times, the relational boundary coverage reports a cumulative result over all instances where the function is called. If a relational operation in the function uses operands of different types in the different calls, relational boundary coverage uses tolerance rules for the stricter operand type. For instance, if a relational operation uses `int32` operands in one call, and `double` operands in another call, relational boundary coverage uses tolerance rules for `double` operands.

For information on the tolerance rules and the order of strictness of types, see “Relational Boundary Coverage” on page 16-8.

How to Collect Coverage for MATLAB Functions

When you simulate your model, the Simulink Verification and Validation software can collect coverage data for MATLAB functions configured for code generation. To enable model coverage, select **Analysis > Coverage > Settings** and select **Enable coverage analysis**.

You collect model coverage for MATLAB functions as follows:

- Functions in a MATLAB Function block
- Functions in an external MATLAB file

To collect coverage for an external MATLAB file, **Coverage** pane of the Configuration Parameters dialog box, select **Coverage for MATLAB files**.

- Simulink Design Verifier functions:
 - `sldv.condition`
 - `sldv.test`
 - `sldv.assume`
 - `sldv.prove`

To collect coverage for these functions, on the **Coverage** pane of the Configuration Parameters dialog box, select the **Objectives and Constraints** coverage metric.

The following section provides model coverage examples for each of these situations.

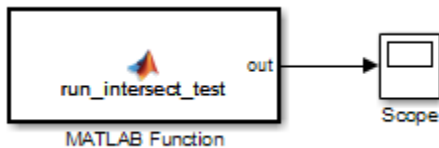
Examples: Model Coverage for MATLAB Functions

- “Model Coverage for MATLAB Function Blocks” on page 20-26
- “Model Coverage for MATLAB Functions in an External File” on page 20-36
- “Model Coverage for Simulink Design Verifier MATLAB Functions” on page 20-36

Model Coverage for MATLAB Function Blocks

Simulink Verification and Validation software measures model coverage for functions in a MATLAB Function block.

The following model contains two MATLAB functions in its MATLAB Function block:



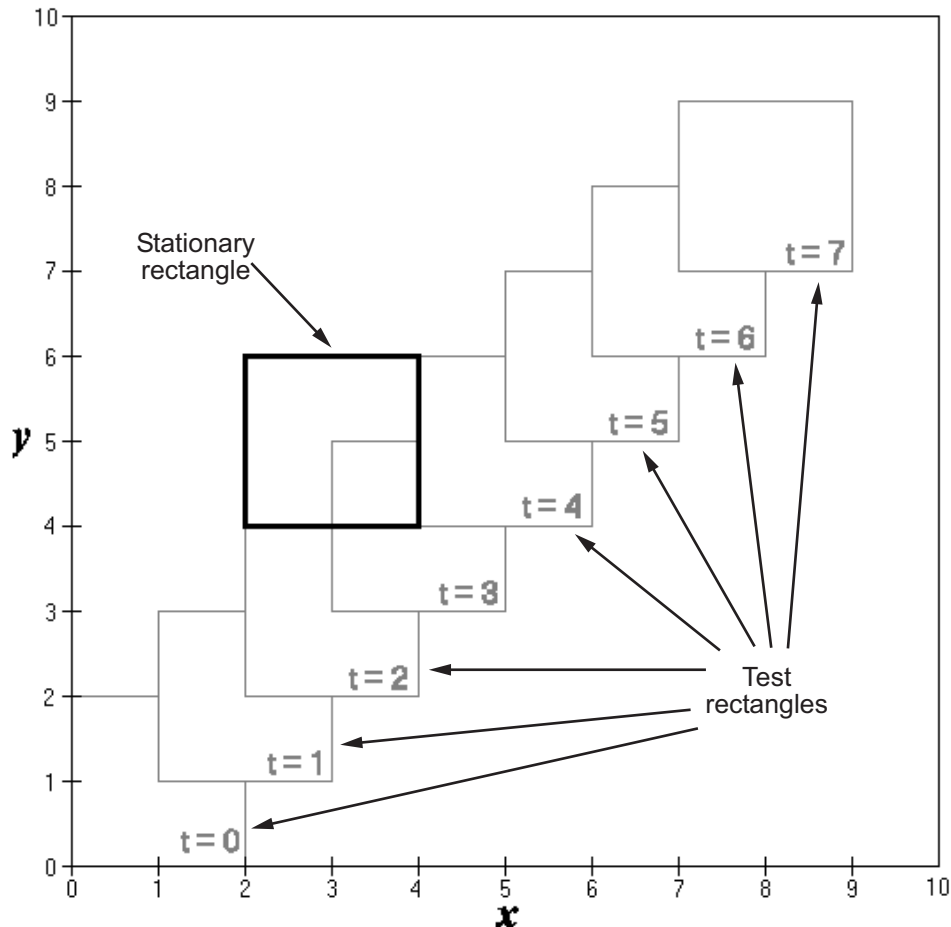
In the Configuration Parameters dialog box, on the **Solver** pane, under **Solver options**, the simulation parameters are set as follows:

- **Type** — Fixed-step
- **Solver** — discrete (no continuous states)
- **Fixed-step size (fundamental sample time)** — 1

The MATLAB Function block contains two functions:

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

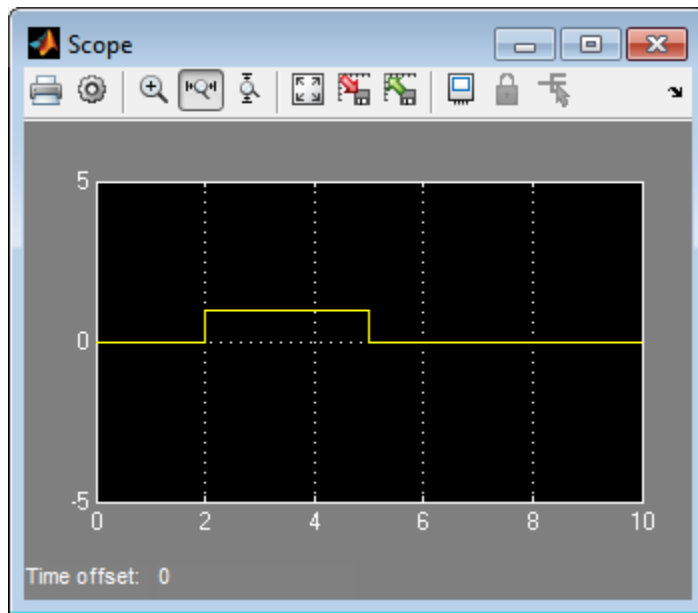
The coordinates for the origin of the moving test rectangle are represented by persistent data $x1$ and $y1$, which are both initialized to -1 . For the first sample, $x1$ and $y1$ both increase to 0 . From then on, the progression of rectangle arguments during simulation is as shown in the following graphic.



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

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

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



After the simulation, the model coverage report appears in a browser window. After the summary in the report, the Details section of the model coverage report reports on each part of the model.

The model coverage report for the MATLAB Function block shows that the block itself has no decisions of its own apart from its function.

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

Coverage for the MATLAB Function `run_intersect_test`

Model coverage for the MATLAB Function block function `run_intersect_test` appears under the linked name of the function. Clicking this link opens the function in the editor.

Below the linked function name is a link to the model coverage report for the parent MATLAB Function block that contains the code for `run_intersect_test`.

MATLAB Function "run_intersect_test"	
Parent:	ex_mc_eml_intersecting_rectangles/MATLAB Function
Uncovered Links:	
Metric	Coverage
Cyclomatic Complexity	7
Decision	100% (8/8) decision outcomes
Condition	88% (7/8) condition outcomes
MCDC	75% (3/4) conditions reversed the outcome

The top half of the report for the function summarizes its model coverage results. The coverage metrics for `run_intersect_test` include decision, condition, and MCDC coverage. You can best understand these metrics by examining the code for `run_intersect_test`.

```
1 function out = run_intersect_test
2 % Call rect_intersect to see if a moving test rectangle
3 % and a stationary rectangle intersect
4
5 persistent x1 y1;
6 if isempty(x1)
7     x1 = -1, y1 = -1;
8 end
9
10 x1 = x1 + 1;
11 y1 = y1 + 1;
12 out = rect_intersect([x1 y1 2 2]', [2 4 2 2]');
13
14 function out = rect_intersect(rect1, rect2);
15 % Return 1 if two 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 local function `rect_intersect` is executed.

- Lines 27 and 30 receive decision, condition, and MCDC coverage for their `if` statements and conditions.

Each of these lines is the subject of a report that follows the listing.

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

The following sections display the coverage for each `run_intersect_test` decision line. The coverage for each line is titled with the line itself, which if clicked, opens the editor to the designated line.

Coverage for Line 1

The coverage metrics for line 1 are part of the coverage data for the function `run_intersect_test`.

The first line of every MATLAB function configured for code generation receives coverage analysis indicative of the decision to run the function in response to a call. Coverage for `run_intersect_test` indicates that it executed at least once during simulation.

[#1: function out = run_intersect_test](#)

Decisions analyzed:

function out = run_intersect_test	100%
executed	11/11

Coverage for Line 6

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

#6: if isempty(x1)**Decisions analyzed:**

if isempty(x1)	100%
false	10/11
true	1/11

Coverage for Line 14

The *Decisions analyzed* table indicates that the local function `rect_intersect` executed during testing, thus receiving 100% coverage.

#14: function out = rect_intersect(rect1, rect2);**Decisions analyzed:**

function out = rect_intersect(rect1, rect2);	100%
executed	11/11

Coverage for Line 27

The *Decisions analyzed* table indicates that there are two possible outcomes for the decision in line 27: `true` and `false`. Five of the eight times it was executed, the decision evaluated to `false`. The remaining three times, it evaluated to `true`. Because both possible outcomes occurred, decision coverage is 100%.

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

the total true occurrences for the decision to three, as reported in the *Decisions analyzed* table.

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

#27: if (top1 < bottom2 || top2 < bottom1)

Decisions analyzed:

if (top1 < bottom2 top2 < bottom1)	100%
false	5/11
true	6/11

Conditions analyzed:

Description:	True	False
top1 < bottom2	2	9
top2 < bottom1	4	5

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

Decision/Condition:	True Out	False Out
top1 < bottom2 top2 < bottom1		
top1 < bottom2	Tx	FF
top2 < bottom1	FT	FF

Coverage for Line 30

The line 30 decision, if (right1 < left2 || right2 < left1), is nested in the if statement of the line 27 decision and is evaluated only if the line 27 decision is false.

Because the line 27 decision evaluated **false** five times, line 30 is evaluated five times, three of which are **false**. Because both the **true** and **false** outcomes are achieved, decision coverage for line 30 is 100%.

Because line 30, like line 27, has two conditions related by a logical OR operator (| |), condition 2 is tested only if condition 1 is **false**. Because condition 1 tests **false** five times, condition 2 is tested five times. Of these, condition 2 tests **true** two times and **false** three times, which accounts for the two occurrences of the **true** outcome for this decision.

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

#30: if (right1 < left2 || right2 < left1)

Decisions analyzed:

if (right1 < left2 right2 < left1)	100%
false	3/5
true	2/5

Conditions analyzed:

Description:	True	False
right1 < left2	0	5
right2 < left1	2	3

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

Decision/Condition:	True Out	False Out
right1 < left2 right2 < left1		
right1 < left2	(Tx)	FF
right2 < left1	FT	FF

Coverage for run_intersect_test

On the *Details* tab, the metrics that summarize coverage for the entire run_intersect_test function are reported and repeated as shown.

MATLAB Function " run_intersect_test "	
Parent:	ex_mc_eml_intersecting_rectangles/MATLAB Function
Uncovered Links:	
Metric	Coverage
Cyclomatic Complexity	7
Decision	100% (8/8) decision outcomes
Condition	88% (7/8) condition outcomes
MCDC	75% (3/4) conditions reversed the outcome

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

- There are eight decision outcomes reported for run_intersect_test in the line reports:
 - One for line 1 (executed)
 - Two for line 6 (true and false)
 - One for line 14 (executed)
 - Two for line 27 (true and false)
 - Two for line 30 (true and false).

The decision coverage for each line shows 100% decision coverage. This means that decision coverage for run_intersect_test is eight of eight possible outcomes, or 100%.

- There are four conditions reported for run_intersect_test in the line reports. Lines 27 and 30 each have two conditions, and each condition has two condition outcomes (true and false), for a total of eight condition outcomes in run_intersect_test. All conditions tested positive for both the true and false outcomes except the first condition of line 30 (`right1 < left2`). This means that condition coverage for run_intersect_test is seven of eight, or 88%.

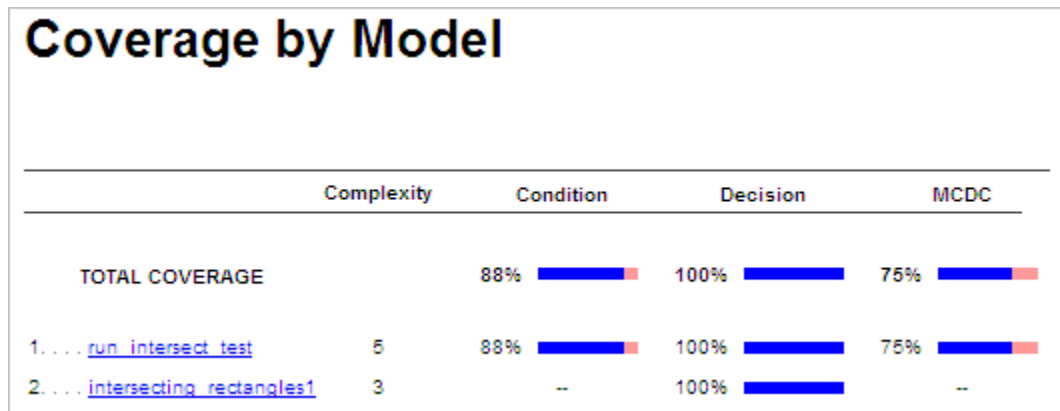
- The MCDC coverage tables for decision lines 27 and 30 each list two cases of decision reversal for each condition, for a total of four possible reversals. Only the decision reversal for a change in the evaluation of the condition `right1 < left2` of line 30 from `true` to `false` did not occur during simulation. This means that three of four, or 75% of the possible reversal cases were tested for during simulation, for a coverage of 75%.

Model Coverage for MATLAB Functions in an External File

Using the same model in “Model Coverage for MATLAB Function Blocks” on page 20-26, suppose the MATLAB functions `run_intersect_test` and `rect_intersect` are stored in an external MATLAB file named `run_intersect_test.m`.

To collect coverage for MATLAB functions in an external file, on the **Coverage** pane of the Configuration Parameters dialog box, select **Coverage for MATLAB files**.

After simulation, the model coverage report summary contains sections for the top-level model and for the external function.



The model coverage report for `run_intersect_test.m` reports the same coverage data as if the functions were stored in the MATLAB Function block.

For a detailed example of a model coverage report for a MATLAB function in an external file, see “External MATLAB File Coverage Report” on page 21-5.

Model Coverage for Simulink Design Verifier MATLAB Functions

If the MATLAB code includes any of the following Simulink Design Verifier functions configured for code generation, you can measure coverage:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

For this example, consider the following model that contains a MATLAB Function block.



The MATLAB Function block contains the following code:

```
function y = fcn(u)
% This block supports MATLAB for code generation.

sldv.condition(u > -30)
sldv.test(u == 30)
y = 1;
```

To collect coverage for Simulink Design Verifier MATLAB functions, on the **Coverage** pane of the Configuration Parameters dialog box, under **Other metrics**, select **Objectives and Constraints**.

After simulation, the model coverage report listed coverage for the `sldv.condition` and `sldv.test` functions. For `sldv.condition`, the expression `u > -30` evaluated to true 51 times. For `sldv.test`, the expression `u == 30` evaluated to true 51 times.

eM Function "fcn"

Parent: [ex_mc_eml_sldv_blocks/MATLAB Function](#)

Metric	Coverage
Cyclomatic Complexity	1
Decision	100% (1/1) decision outcomes
Test Objective	100% (1/1) objective outcomes
Test Condition	100% (1/1) objective outcomes

```

1 function y = fcn(u)
2 % This block supports MATLAB for code generation.
3
4 sldv.condition(u > -30)
5 sldv.test(u == 30)
6 y = 1;
    
```

#1: function y = fcn(u)

Decisions analyzed:

function y = fcn(u)	100%
executed	51/51

#4: sldv.condition(u > -30)

Test Condition analyzed:

sldv.condition(u > -30)	51/51
-------------------------	-------

#5: sldv.test(u == 30)

Test Objective analyzed:

sldv.test(u == 30)	51/51
--------------------	-------

For an example of model coverage data for Simulink Design Verifier blocks, see “Objectives and Constraints Coverage” on page 16-7.

Model Coverage for C and C++ S-Functions

When you record coverage for models containing supported C/C++ S-Functions, coverage is recorded for the code within the C/C++ S-Functions. The coverage results for S-Function blocks can be viewed in the same report as the rest of the model. For each S-Function block, the report links to a detailed coverage report for the C/C++ code in the block.

To generate coverage report for S-Functions:

- 1 When creating S-Functions, enable support for coverage.
- 2 When generating coverage report, enable support for S-Functions.

In this section...
“Make S-Function Compatible with Model Coverage” on page 20-40
“Generate Coverage Report for S-Function” on page 20-41

Make S-Function Compatible with Model Coverage

If you use the `legacy_code` function, S-Function Builder block or mex function to create your S-Functions, adapt your method appropriately to make the S-Function compatible with model coverage.

For more information on the three approaches, see “Creating C MEX S-Functions”.

- “S-Function Using `legacy_code` Function” on page 20-40
- “S-Function Using S-Function Builder” on page 20-41
- “S-Function Using mex Function” on page 20-41

S-Function Using `legacy_code` Function

- 1 Initialize a MATLAB structure with fields that represent Legacy Code Tool properties.

```
def = legacy_code('initialize')
```

- 2 To enable model coverage, turn on the option `def.Options.supportCoverage`.

```
def.Options.supportCoverage = true;
```

- 3 Use the structure `def` in the usual way to generate an S-function. For an example, see “Coverage for S-Functions”.

S-Function Using S-Function Builder

- 1 Copy an instance of the S-Function Builder block from the **User-Defined Functions** library in the Library Browser into the your model.
- 2 Double-click the block to open the S-Function Builder dialog box.
- 3 On the **Build Info** tab, select **Enable support for coverage**.

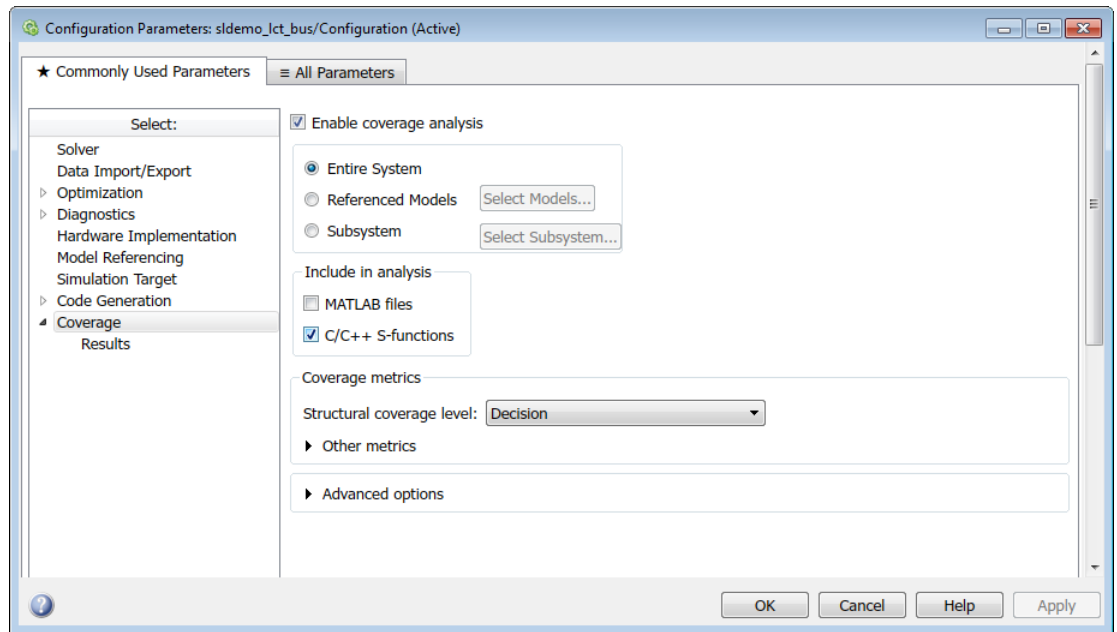
S-Function Using mex Function

If you use the `mex` function to compile and link your source files, use the `slcovmex` function instead. The `slcovmex` function compiles your source code and also makes it compatible with coverage.

This function has the same syntax and takes the same options as the `mex` function. In addition, you can provide some options relevant for model coverage. For more information, see `slcovmex`.

Generate Coverage Report for S-Function

- 1 Select **Analysis > Coverage > Settings**.
- 2 On the **Coverage** pane of the Configuration Parameters dialog box, select **C/C++ S-functions**.



When you run a simulation, the coverage report contains coverage metrics for C/C++ S-Function blocks in your model. For each S-Function block, the report links to a detailed coverage report for the C/C++ code in the block.

Related Examples

- “View Coverage Results for C/C++ Code in S-Function Blocks” on page 20-43

More About

- “C/C++ S-Function” on page 17-25

View Coverage Results for C/C++ Code in S-Function Blocks

This example shows how to view coverage results for the C/C++ code in **S-Function** blocks in your model. To view coverage results for the C/C++ code in the blocks:

- Enable support for S-Function coverage. For more information, see “Model Coverage for C and C++ S-Functions” on page 20-40.
- Run simulation and view the coverage report.


The coverage results for **S-Function** blocks can be viewed in the same report as the rest of the model. For each S-Function block, the report links to a detailed coverage report for the C/C++ code in the block.

To view the full code coverage report used in this example, follow the steps in “Coverage for S-Functions”.

- 1 In the coverage report, view the coverage metrics for the S-Function block.

S-Function block "[sldemo_sfun_counterbus](#)"

Parent: [sldemo_lct_bus/TestCounter](#)

Uncovered Links: 

Metric	Coverage
Cyclomatic Complexity	3
Condition	67% (4/6) condition outcomes
Decision	75% (3/4) decision outcomes
MCDC	50% (1/2) conditions reversed the outcome

Detailed Report: [sldemo_lct_bus_sldemo_sfun_counterbus_instance_1_cov.html](#)

For more information on the coverage report format, see “Top-Level Model Coverage Report” on page 21-12.

- 2 Select the **Detailed Report** link. The code coverage report for the S-Function block opens.

- 3 Select each of the links in **Table Of Contents** to navigate to various sections of the report.

Code Coverage Report for S-Function sldemo_sfun_counterbus









Table Of Contents

1. [Analysis Information](#)
2. [Tests](#)
3. [Summary](#)
4. [Details](#)
5. [Code](#)

Section Title	Purpose	
Analysis information	Contains information such as time when model was created and last modified, and file size.	
Tests	Contains information about the simulation such as start and end time.	
Summary	Contains coverage information about the files and functions in the S-Function block. For each file and function, the percentage coverage is displayed. The coverage types relevant for the code are the following:	
	Coverage Type	Label
	“Cyclomatic Complexity for Code Coverage” on page 19-4	Complexity
	“Condition Coverage for Code Coverage” on page 19-3	Condition.
	“Decision Coverage for Code Coverage” on page 19-3	Decision
“Modified Condition/ Decision Coverage (MCDC) for Code Coverage” on page 19-4	MCDC	

Section Title	Purpose	
	“Relational Boundary for Code Coverage” on page 19-5	Relational Boundary
	Percentage of statements covered	Stmt
Details	Contains coverage information about the statements that receive condition, decision or MCDC coverage. The information is grouped by file and function.	
Code	Contains the C/C++ code. Statements that are not covered are highlighted in pink.	

- 4 In the **Summary** section, select each file or function name to see details of coverage for statements in the file or function.

File Contents	Complexity	Decision	Condition	MCDC	Stmt
1. counterbus.c	3	75% 	67% 	50% 	90% 
2. ... counterbusFcn	3	75% 	67% 	50% 	90% 

- 5 The condition, decision or MCDC outcomes that were not tested during simulation are highlighted in pink. Within the details for a file or function, scroll down to note these cases and investigate them further.

2.1 Decision/Condition `(u1->limits.upper_saturation_limit >= limit) && inputGElower`

Function: `counterbusFcn` (line 6)

Metric	Coverage
Decision	100% (2/2) decision outcomes
Condition	75% (3/4) condition outcomes
MCDC	50% (1/2) conditions reversed the outcome

Decisions analyzed:

<code>(u1->limits.upper_saturation_limit >= limit) && inputGElower</code>	100%
false	61/201
true	140/201

Conditions analyzed:

Description:	True	False
<code>u1->limits.upper_saturation_limit >= limit</code>	140	61
<code>inputGElower</code>	140	0

- To obtain an overview of the statements that were not covered, navigate to the **Code** section. This section contains your code with the uncovered statements highlighted in pink.

Code

```
1  /* Copyright 2005-2006 The MathWorks, Inc. */
2
3
4  #include "counterbus.h"
5
6  void counterbusFcn(COUNTERBUS *u1, int32_T u2, COUNTERBUS *y1, int32_T *y2)
7  {
8      int32_T limit;
9      boolean_T inputGELower;
10
11     limit = u1->inputsignal.input + u2;
12
13     inputGELower = (limit >= u1->limits.lower_saturation_limit);
14
15     if((u1->limits.upper_saturation_limit >= limit) && inputGELower) {
16         *y2 = limit;
17     } else {
18
19         if(inputGELower) {
20             limit = u1->limits.upper_saturation_limit;
21         } else {
22             limit = u1->limits.lower_saturation_limit;
23         }
24         *y2 = limit;
25     }
26
27     y1->inputsignal.input = *y2;
28     y1->limits = u1->limits;
29
30 }
31
```

More About

- “C/C++ S-Function” on page 17-25

Model Coverage for Stateflow Charts

In this section...

“How Model Coverage Reports Work for Stateflow Charts” on page 20-48

“Specify Coverage Report Settings for Stateflow Charts” on page 20-49

“Cyclomatic Complexity for Stateflow Charts” on page 20-49

“Decision Coverage for Stateflow Charts” on page 20-50

“Condition Coverage for Stateflow Charts” on page 20-53

“MCDC Coverage for Stateflow Charts” on page 20-54

“Relational Boundary Coverage for Stateflow Charts” on page 20-54

“Simulink Design Verifier Coverage for Stateflow Charts” on page 20-54

“Model Coverage Reports for Stateflow Charts” on page 20-56

“Model Coverage for Stateflow State Transition Tables” on page 20-65

“Model Coverage for Stateflow Atomic Subcharts” on page 20-66

“Model Coverage for Stateflow Truth Tables” on page 20-69

“Colored Stateflow Chart Coverage Display” on page 20-74

How Model Coverage Reports Work for Stateflow Charts

To generate a Model Coverage report, select **Analysis > Coverage > Settings** and specify the desired options on the **Coverage > Results** pane of the **Coverage** pane of the Configuration Parameters dialog box. For Stateflow charts, the Simulink Verification and Validation software records the execution of the chart itself and the execution of states, transition decisions, and individual conditions that compose each decision. After simulation ends, the model coverage reports on how thoroughly a model was tested. The report shows:

- How many times each exclusive substate is executed or exited from its parent superstate and entered due to parent superstate history
- How many times each transition decision has been evaluated as true or false
- How many times each condition has been evaluated as true or false

Note: To measure model coverage data for a Stateflow chart, you must:

- Have a Stateflow license.
 - Have debugging/animation enabled for the chart.
-

Specify Coverage Report Settings for Stateflow Charts

To specify coverage recording settings, select **Analysis > Coverage > Settings** in the Simulink Editor. Then select **Enable coverage analysis**.

By selecting the **Generate report automatically after analysis** option in the **Coverage > Results** pane of the Configuration Parameters dialog box, you can create an HTML report containing the coverage data generated during simulation of the model. The report appears in the MATLAB Help browser at the end of simulation.

Enabling coverage analysis also enables the selection of different coverages that you can specify for your reports. The following sections address only coverage metrics that affect reports for Stateflow charts. These metrics include decision coverage, condition coverage, and MCDC coverage.

Cyclomatic Complexity for Stateflow Charts

Cyclomatic complexity is a measure of the complexity of a software module based on its edges, nodes, and components within a control-flow chart. It provides an indication of how many times you need to test the module.

The calculation of cyclomatic complexity is as follows:

$$CC = E - N + p$$

where **CC** is the cyclomatic complexity, **E** is the number of edges, **N** is the number of nodes, and **p** is the number of components.

Within the Model Coverage tool, each decision is exactly equivalent to a single control flow node, and each decision outcome is equivalent to a control flow edge. Any additional structure in the control-flow chart is ignored since it contributes the same number of nodes as edges and therefore has no effect on the complexity calculation. Therefore, you can express cyclomatic complexity as follows:

$$CC = \text{OUTCOMES} - \text{DECISIONS} + p$$

For analysis purposes, each chart counts as a single component.

Decision Coverage for Stateflow Charts

Decision coverage interprets a model execution in terms of underlying decisions where behavior or execution must take one outcome from a set of mutually exclusive outcomes.

Note: Full coverage for an object of decision means that every decision has had at least one occurrence of each of its possible outcomes.

Decisions belong to an object making the decision based on its contents or properties. The following table lists the decisions recorded for model coverage for the Stateflow objects owning them. The sections that follow the table describe these decisions and their possible outcomes.

Object	Possible Decisions
Chart	<p>If a chart is a triggered Simulink block, it must decide whether or not to execute its block.</p> <p>If a chart contains exclusive (OR) substates, it must decide which of its states to execute.</p>
State	<p>If a state is a superstate containing exclusive (OR) substates, it must decide which substate to execute.</p> <p>If a state has on <i>event name</i> actions (which might include temporal logic operators), the state must decide whether or not to execute the actions.</p>
Transition	<p>If a transition is a conditional transition, it must decide whether or not to exit its active source state or junction and enter another state or junction.</p>

Chart as a Triggered Simulink Block Decision

If the chart is a triggered block in a Simulink model, the decision to execute the block is tested. If the block is not triggered, there is no decision to execute the block, and the measurement of decision coverage is not applicable (NA).

Chart Containing Exclusive OR Substates Decision

If the chart contains exclusive (OR) substates, the decision on which substate to execute is tested. If the chart contains only parallel AND substates, this coverage measurement is not applicable (NA).

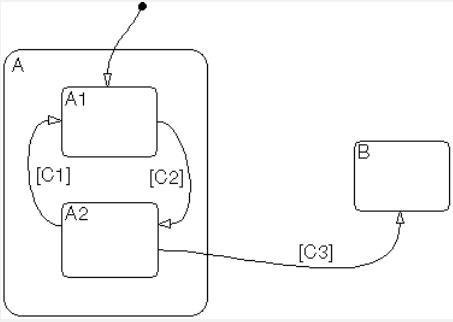
Superstate Containing Exclusive OR Substates Decision

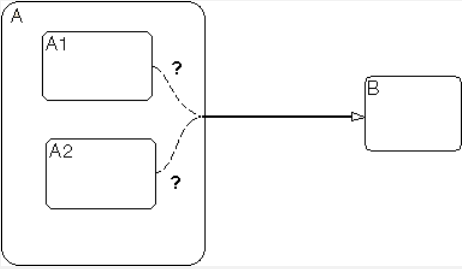
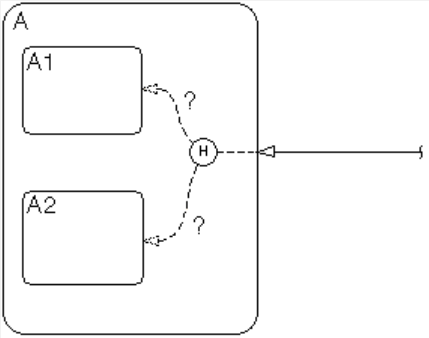
Since a chart is hierarchically processed from the top down, procedures such as exclusive (OR) substate entry, exit, and execution are sometimes decided by the parenting superstate.

Note: Decision coverage for superstates applies only to exclusive (OR) substates. A superstate makes no decisions for parallel (AND) substates.

Since a superstate must decide which exclusive (OR) substate to process, the number of decision outcomes for the superstate is the number of exclusive (OR) substates that it contains. In the examples that follow, the choice of which substate to process can occur in one of three possible contexts.

Note: Implicit transitions appear as dashed lines in the following examples.

Context	Example	Decisions That Occur
Active call	<p>States A and A1 are active.</p>  <p>The diagram shows a stateflow chart. A large rounded rectangle labeled 'A' contains two smaller rounded rectangles labeled 'A1' and 'A2'. A solid arrow points from a start node to 'A1'. A solid arrow points from 'A1' to 'A2' labeled '[C1]'. A solid arrow points from 'A2' to 'A1' labeled '[C2]'. A solid arrow points from 'A2' to a separate rounded rectangle labeled 'B' labeled '[C3]'. A dashed line connects the start node to 'A2'.</p>	<ul style="list-style-type: none"> • The parent of states A and B must decide which of these states to process. This decision belongs to the parent. Since A is active, it is processed. • State A, the parent of states A1 and A2, must decide which of these states to process. This decision belongs to state A. Since A1 is active, it is processed. <p>During processing of state A1, all outgoing transitions are tested. This decision belongs to the transition</p>

Context	Example	Decisions That Occur
		and not to the parent state A. In this case, the transition marked by condition C2 is tested and a decision is made whether to take the transition to A2 or not.
Implicit substate exit	A transition takes place whose source is superstate A and whose destination is state B. 	If the superstate has two exclusive (OR) substates, it is the decision of superstate A which substate performs the implicit transition from substate to superstate.
Substate entry with a history junction	A history junction records which substate was last active before the superstate was exited. 	If that superstate becomes the destination of one or more transitions, the history junction decides which previously active substate to enter.

For more information, see “State Details Report Section” on page 20-59.

State with On Event_Name Action Statement Decision

A state that has an on *event_name* action statement must decide whether to execute that statement based on the reception of a specified event or on an accumulation of the specified event when using temporal logic operators.

Conditional Transition Decision

A conditional transition is a transition with a triggering event and/or a guarding condition. In a conditional transition from one state to another, the decision to exit one state and enter another is credited to the transition itself.

Note: Only conditional transitions receive decision coverage. Transitions without decisions are not applicable to decision coverage.

Condition Coverage for Stateflow Charts

Condition coverage reports on the extent to which all possible outcomes are achieved for individual subconditions composing a transition decision.

Note: Full condition coverage means that all possible outcomes occurred for each subcondition in the test of a decision.

For example, for the decision [A & B & C] on a transition, condition coverage reports on the true and false occurrences of each of the subconditions A, B, and C. This results in eight possible outcomes: true and false for each of three subconditions.

Outcome	A	B	C
1	T	T	T
2	T	T	F
3	T	F	T
4	T	F	F
5	F	T	T
6	F	T	F
7	F	F	T

Outcome	A	B	C
8	F	F	F

For more information, see “Transition Details Report Section” on page 20-62.

MCDC Coverage for Stateflow Charts

The Modified Condition Decision Coverage (MCDC) option reports a test's coverage of occurrences in which changing an individual subcondition within a transition results in changing the entire transition trigger expression from true to false or false to true.

Note: If matching true and false outcomes occur for each subcondition, coverage is 100%.

For example, if a transition executes on the condition [C1 & C2 & C3 | C4 & C5], the MCDC report for that transition shows actual occurrences for each of the five subconditions (C1, C2, C3, C4, C5) in which changing its result from true to false is able to change the result of the entire condition from true to false.

Relational Boundary Coverage for Stateflow Charts

If a transition in a Stateflow chart involves a relational operation, it receives relational boundary coverage. For more information, see “Relational Boundary Coverage” on page 16-8.

Simulink Design Verifier Coverage for Stateflow Charts

You can use the following Simulink Design Verifier functions inside Stateflow charts:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

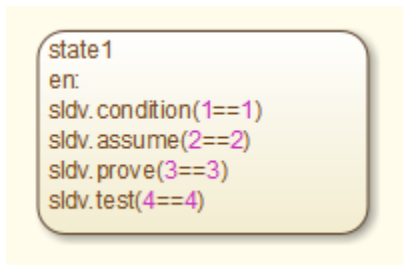
If you do not have a Simulink Design Verifier license, you can collect model coverage for a Stateflow chart containing these functions, but you cannot analyze the model using the Simulink Design Verifier software.

When you specify the **Objectives and Constraints** coverage metric in the **Coverage** pane of the Configuration Parameters dialog box, the Simulink Verification and Validation software records coverage for these functions.

Each of these functions evaluates an expression *expr*, for example, `sldv.test(expr)`, where *expr* is any valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to `true`.

If *expr* is `true` for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Verification and Validation software reports coverage for that function as 0%.

Consider a model that contains this Stateflow chart:



To collect coverage for Simulink Design Verifier functions, on the **Coverage** pane of the Configuration Parameters dialog box, select **Objectives and Constraints**.

After simulation, the model coverage report lists coverage for the `sldv.condition`, `sldv.assume`, `sldv.prove`, and `sldv.test` functions.

Metric	Coverage
Cyclomatic Complexity	0
Proof Assumption	100% (1/1) objective outcomes
Test Condition	100% (1/1) objective outcomes
Proof Objective	100% (1/1) objective outcomes
Test Objective	100% (1/1) objective outcomes

Proof Assumption analyzed:

sldv.assume(2==2)	1/1
-------------------	-----

Test Condition analyzed:

sldv.condition(1==1)	1/1
----------------------	-----

Proof Objective analyzed:

sldv.prove(3==3)	1/1
------------------	-----

Test Objective analyzed:

sldv.test(4==4)	1/1
-----------------	-----

Model Coverage Reports for Stateflow Charts

- “Summary Report Section” on page 20-56
- “Subsystem and Chart Details Report Sections” on page 20-57
- “State Details Report Section” on page 20-59
- “Transition Details Report Section” on page 20-62

The following sections of a Model Coverage report were generated by simulating the `sf_boiler` model, which includes the Bang-Bang Controller chart. The coverage metrics for **MCDC** are enabled for this report.

Summary Report Section

The Summary section shows coverage results for the entire test and appears at the beginning of the Model Coverage report.

Summary

Model Hierarchy/Complexity:		Test 1					
		DI		CI		MCDC	
1. sf_boiler	20	89%		71%		43%	
2. . . . Bang-Bang Controller	16	95%		71%		43%	
3. SF: Bang-Bang Controller	15	95%		71%		43%	
4. SF: Heater	12	94%		71%		43%	
5. SF: Off	2	100%		75%		50%	
6. SF: On	4	88%		NA		NA	
7. SF: flash_LED	1	100%		NA		NA	
8. SF: turn_boiler	1	100%		NA		NA	
9. . . . Boiler Plant model	3	67%		NA		NA	
10. digital thermometer	2	50%		NA		NA	
11. ADC	2	50%		NA		NA	

Each line in the hierarchy summarizes the coverage results at that level and the levels below it. You can click a hyperlink to a later section in the report with the same assigned hierarchical order number that details that coverage and the coverage of its children.

The top level, `sf_boiler`, is the Simulink model itself. The second level, Bang-Bang Controller, is the Stateflow chart. The next levels are superstates within the chart, in order of hierarchical containment. Each superstate uses an SF: prefix. The bottom level, Boiler Plant model, is an additional subsystem in the model.

Subsystem and Chart Details Report Sections

When recording coverage for a Stateflow chart, the Simulink Verification and Validation software reports two types of coverage for the chart—Subsystem and Chart.

- *Subsystem* — This section reports coverage for the chart:

- *Coverage (this object)*: Coverage data for the chart as a container object
- *Coverage (inc.) descendants*: Coverage data for the chart and the states and transitions in the chart.

If you click the hyperlink of the subsystem name in the section title, the Bang-Bang Controller block is highlighted in the block diagram.

Decision coverage is not applicable (NA) because this chart does not have an explicit trigger. Condition coverage and MCDC are not applicable (NA) for a chart, but apply to its descendants.

2. SubSystem block "[Bang-Bang Controller](#)"

Parent: [/sf_boiler](#)
Child Systems: [Bang-Bang Controller](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	16
Condition (C1)	NA	71% (10/14) condition outcomes
Decision (D1)	NA	95% (21/22) decision outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

- *Chart* — This section reports coverage for the chart:
 - *Coverage (this object)*: Coverage data for the chart and its inputs
 - *Coverage (inc.) descendants*: Coverage data for the chart and the states and transitions in the chart.

If you click the hyperlink of the chart name in the section title, the chart opens in the Stateflow Editor.

Decision coverage is listed appears for the chart and its descendants. Condition coverage and MCDC are not applicable (NA) for a chart, but apply to its descendants.

3. Chart "[Bang-Bang Controller](#)"

Parent: [sf_boiler/Bang-Bang Controller](#)
Child Systems: [Heater](#), [flash LED](#), [turn boiler](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	15
Condition (C1)	NA	71% (10/14) condition outcomes
Decision (D1)	100% (2/2) decision outcomes	95% (21/22) decision outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

Decisions analyzed:

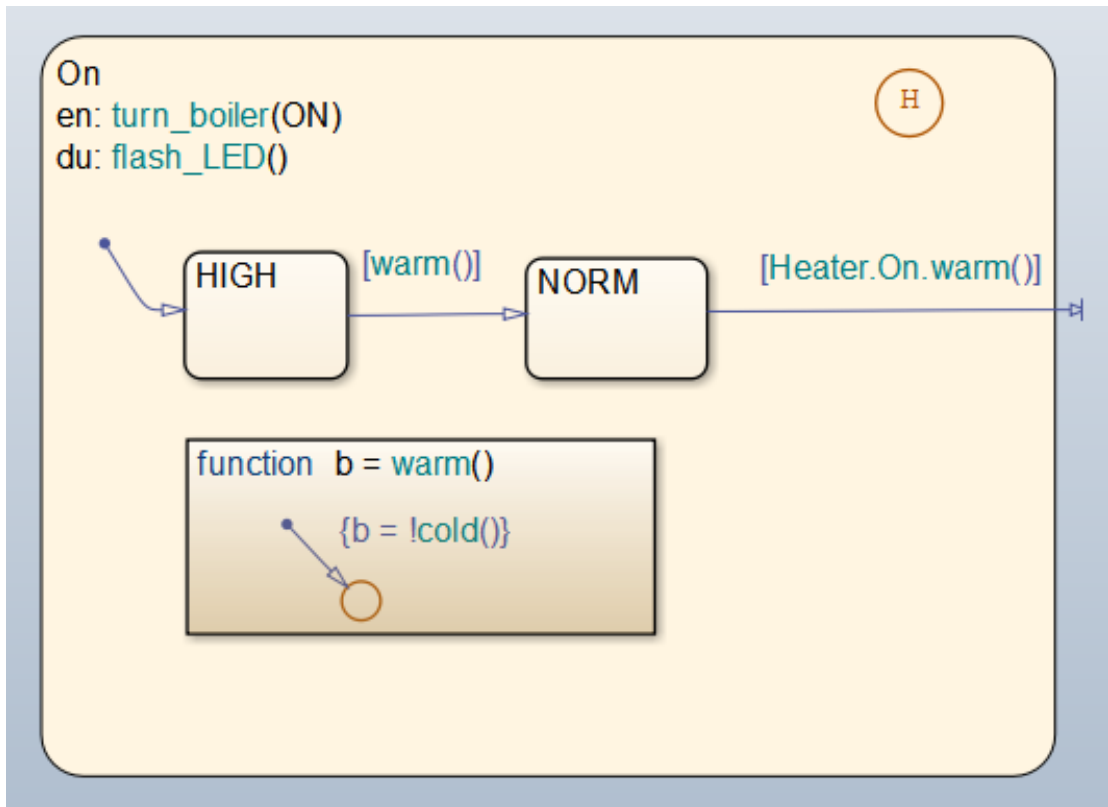
Substate executed	100%
State "Off"	1160/1400
State "On"	240/1400

State Details Report Section

For each state in a chart, the coverage report includes a *State* section with details about the coverage recorded for that state.

In the `sf_boiler` model, the state `On` resides in the box `Heater`. `On` is a superstate that contains:

- Two substates `HIGH` and `NORM`
- A history junction
- The function `warm`



The coverage report includes a *State* section on the state `On`.

6. State "On"

Parent: [sf_boiler/Bang-Bang Controller.Heater](#)

Uncovered Links: ◀ ▶

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	3	4
Decision (D1)	83% (5/6) decision outcomes	88% (7/8) decision outcomes

Decisions analyzed:

Substate executed	100%
State "HIGH"	150/233
State "NORM"	83/233
Substate exited when parent exits	50%
State "HIGH"	7/7
State "NORM"	0/7
Previously active substate entered due to history	100%
State "HIGH"	7/28
State "NORM"	21/28

The decision coverage for the On state tests the decision of which substate to execute.

The three decisions are listed in the report:

- Under *Substate executed*, which substate to execute when On executes.
- Under *Substate exited when parent exited*, which substate is active when On exits. NORM is listed as never being active when On exits because the coverage tool sees the supertransition from NORM to Off as a transition from On to Off.
- Under *Previously active substate entered due to history*, which substate to reenter when On re-executes. The history junction records the previously active substate.

Because each decision can result in either HIGH or NORM, the total possible outcomes are $3 \times 2 = 6$. The results indicate that five of six possible outcomes were tested during simulation.

Cyclomatic complexity and decision coverage also apply to descendants of the On state. The decision required by the condition [warm()] for the transition from HIGH to NORM brings the total possible decision outcomes to 8. Condition coverage and MCDC are not applicable (NA) for a state.



Note: Nodes and edges that make up the cyclomatic complexity calculation have no direct relationship with model objects (states, transitions, and so on). Instead, this calculation requires a graph representation of the equivalent control flow.

Transition Details Report Section

Reports for transitions appear under the report sections of their owning objects. Transitions do not appear in the model hierarchy of the Summary section, since the hierarchy is based on superstates that own other Stateflow objects.

Transition "[after\(40,sec\) \[cold\(\)\]](#)" from "[Off](#)" to "[On](#)"

Parent: [sf_boiler/Bang-Bang Controller.Heater](#)

Uncovered Links:  

Metric	Coverage
Cyclomatic Complexity	3
Condition (C1)	67% (4/6) condition outcomes
Decision (D1)	100% (2/2) decision outcomes
MCDC (C1)	33% (1/3) conditions reversed the outcome

Decisions analyzed:

Transition trigger expression	100%
false	1131/1160
true	29/1160

Conditions analyzed:

Description:	True	False
Condition 1, "sec"	1160	0
Condition 2, "after(40,sec)"	29	1131
Condition 3, "cold()"	29	0

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

Decision/Condition:	True Out	False Out
Transition trigger expression		
Condition 1, "sec"	TTT	(Fxx)
Condition 2, "after(40,sec)"	TTT	TFx
Condition 3, "cold()"	TTT	(TTF)

The decision for this transition depends on the time delay of 40 seconds and the condition [cold()]. If, after a 40 second delay, the environment is cold (cold() = 1), the

decision to execute this transition and turn the Heater on is made. For other time intervals or environment conditions, the decision is made not to execute.

For decision coverage, both true and false outcomes occurred. Because two of two decision outcomes occurred, coverage was full or 100%.

Condition coverage shows that only 4 of 6 condition outcomes were tested. The temporal logic statement `after(40, sec)` represents two conditions: the occurrence of `sec` and the time delay `after(40, sec)`. Therefore, three conditions on the transition exist: `sec`, `after(40, sec)`, and `cold()`. Since each of these decisions can be true or false, six possible condition outcomes exist.

The **Conditions analyzed** table shows each condition as a row with the recorded number of occurrences for each outcome (true or false). Decision rows in which a possible outcome did not occur are shaded. For example, the first and the third rows did not record an occurrence of a false outcome.

In the MC/DC report, all sets of occurrences of the transition conditions are scanned for a particular pair of decisions for each condition in which the following are true:

- The condition varies from true to false.
- All other conditions contributing to the decision outcome remain constant.
- The outcome of the decision varies from true to false, or the reverse.

For three conditions related by an implied AND operator, these criteria can be satisfied by the occurrence of these conditions.

Condition Tested	True Outcome	False Outcome
1	TTT	Fxx
2	TTT	TFx
3	TTT	TTF

Notice that in each line, the condition tested changes from true to false while the other condition remains constant. Irrelevant contributors are coded with an "x" (discussed below). If both outcomes occur during testing, coverage is complete (100%) for the condition tested.

The preceding report example shows coverage only for condition 2. The false outcomes required for conditions 1 and 3 did not occur, and are indicated by parentheses for both

conditions. Therefore, condition rows 1 and 3 are shaded. While condition 2 has been tested, conditions 1 and 3 have not and MCDC is 33%.

For some decisions, the values of some conditions are irrelevant under certain circumstances. For example, in the decision [C1 & C2 & C3 | C4 & C5] the left side of the | is false if any one of the conditions C1, C2, or C3 is false. The same applies to the right side result if either C4 or C5 is false. When searching for matching pairs that change the outcome of the decision by changing one condition, holding some of the remaining conditions constant is irrelevant. In these cases, the MC/DC report marks these conditions with an "x" to indicate their irrelevance as a contributor to the result. These conditions appear as shown.

Transition "[c1&c2&c3 | c4&c5]" . . .

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

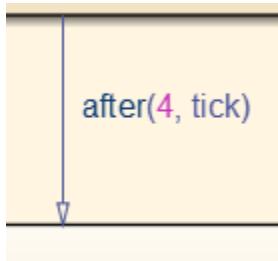
Decision/Condition:	#1 True Out	#1 False Out
Transition trigger expression		
Condition 1, "c1"	TTxx	FxxFx
Condition 2, "c2"	TTxx	TFxFx
Condition 3, "c3"	TTxx	TTFFx
Condition 4, "c4"	FxxTT	FxxFx
Condition 5, "c5"	FxxTT	FxxTF

Consider the first matched pair. Since condition 1 is true in the **True** outcome column, it must be false in the matching **False** outcome column. This makes the conditions C2 and C3 irrelevant for the false outcome since C1 & C2 & C3 is always false if C1 is false. Also, since the false outcome is required to evaluate to false, the evaluation of C4 & C5 must also be false. In this case, a match was found with C4 = F, making condition C5 irrelevant.

Model Coverage for Stateflow State Transition Tables

State transition tables are an alternative way of expressing modal logic in Stateflow. Stateflow charts represent modal logic graphically, and state transition tables can represent equivalent modal logic in tabular form. For more information, see “Tabular Expression of Modal Logic” in the Stateflow documentation.

Coverage results for state transition tables are the same as coverage results for equivalent Stateflow charts, except for a slight difference that arises in coverage of temporal logic. For example, consider the temporal logic expression `after(4, tick)` in the Mode Logic chart of the `slvndemo_covfilt` example model.



In chart coverage, the `after(4, tick)` transition represents two conditions: the occurrence of `tick` and the time delay `after(4, tick)`. Since the temporal event `tick` is never false, the first condition is not satisfiable, and you cannot record 100% condition and MC/DC coverage for the transition `after(4, tick)`.

In state transition table coverage, the `after(4, tick)` transition represents a single decision, with no subcondition for the occurrence of `tick`. Therefore, only decision coverage is recorded.

For state transition tables containing temporal logic decisions, as in the above example, condition coverage and MC/DC is not recorded.

Model Coverage for Stateflow Atomic Subcharts

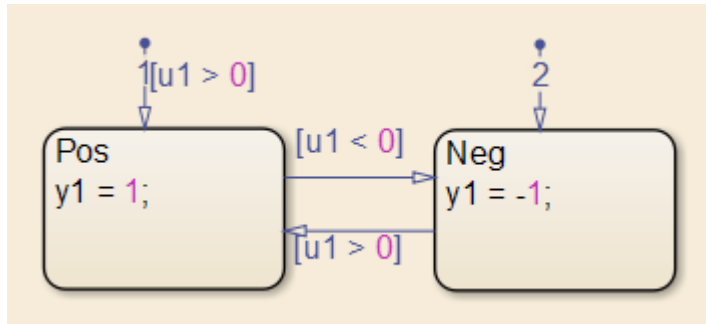
In a Stateflow chart, an atomic subchart is a graphical object that allows you to reuse the same state or subchart across multiple charts and models.

When you specify to record coverage data for a model during simulation, the Simulink Verification and Validation software records coverage for any atomic subcharts in your model. The coverage data records the execution of the chart itself, and the execution of states, transition decisions, and individual conditions that compose each decision in the atomic subchart.

Simulate the `doc_atomic_subcharts_map_io` example model and record decision coverage:

- 1 Open the `doc_atomic_subcharts_map_iodata` model.

This model contains two Sine Wave blocks that supply input signals to the Stateflow chart. Chart contains two atomic subcharts—A and B—that are linked from the same library chart, also named A. The library chart contains the following objects:



- 2 In the Simulink Editor, select **Analysis > Coverage > Settings**

The **Coverage** pane of the Configuration Parameters dialog box appears.

- 3 Select **Enable coverage analysis** and then select **Entire System**.
- 4 On the **Coverage > Results** pane, select **Generate report automatically after analysis**.
- 5 Click **OK** to close the Configuration Parameters dialog box.
- 6 Simulate the `doc_atomic_subcharts_map_iodata` model.

When the simulation completes, the coverage report opens.

The report provides coverage data for atomic subcharts A and B in the following forms:

- For the atomic subchart instance and its contents. Decision coverage is not applicable (NA) because this chart does not have an explicit trigger.

4. Atomic Subchart "A"

Parent: [doc_atomic_subcharts_map iodata/Chart](#)
 Child Systems: [A](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	4
Decision (D1)	NA	88% (7/8) decision outcomes

- For the library chart A and its contents. The chart itself achieves 100% coverage on the input u1, and 88% coverage on the states and transitions inside the library chart.

5. Chart "A"

Parent: [doc_atomic_subcharts_map iodata/Chart.A](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	4
Decision (D1)	100% (2/2) decision outcomes	88% (7/8) decision outcomes

Decisions analyzed:

Substate executed	100%
State "Neg"	4/10
State "Pos"	6/10

Atomic subchart B is a copy of the same library chart A. The coverage of the contents of subchart B is identical to the coverage of the contents of subchart A.

Model Coverage for Stateflow Truth Tables

- “Types of Coverage in Stateflow Truth Tables” on page 20-69
- “Analyze Coverage in Stateflow Truth Tables” on page 20-69

Types of Coverage in Stateflow Truth Tables

Simulink Verification and Validation software reports model coverage for the decisions the objects make in a Stateflow chart during model simulation. The report includes coverage for the decisions the truth table functions make.

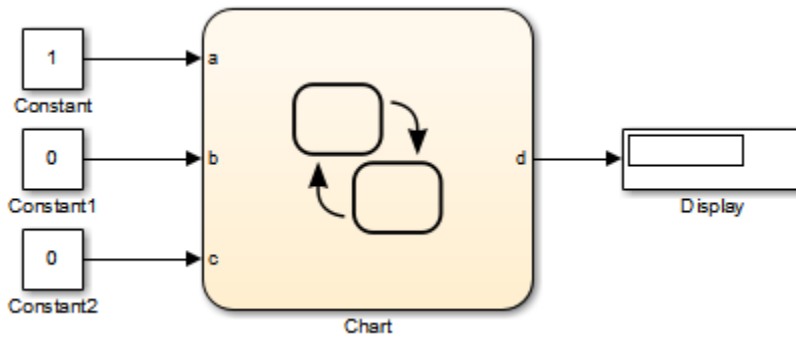
For this type of truth table...	The report includes coverage data for...
Stateflow Classic	Conditions only.
MATLAB	Conditions and only those actions that have decision points. Note: With the MATLAB for code generation action language, you can specify decision points in actions using control flow constructs, such as loops and switch statements.

Note: To measure model coverage data for a Stateflow truth table, you must have a Stateflow license. For more information about Stateflow truth tables, see “Decision Logic” in the Stateflow documentation.

Analyze Coverage in Stateflow Truth Tables

If you have a Stateflow license, you can generate a model coverage report for a truth table.

Consider the following model.



The Stateflow chart contains the following truth table:

The screenshot shows a window titled "Stateflow (truth table) mTruthTable_classic/Chart.ttable". The window contains two tables: a "Condition Table" and an "Action Table".

Condition Table

	Description	Condition	D1	D2	D3	D4	D5
1	x is equal to 1	XEQ1: x == 1	T	F	F	-	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4	A5

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	set t to 5	A5: t=5;
7	Final action: Display message	FINAL: ml disp('truth table ttable exited');

When you simulate the model and collect coverage, the model coverage report includes the following data:

4. Truth Table "ttable"

Parent: [mTruthTable_classic/Chart](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	10
Condition (C1)	NA	17% (3/18) condition outcomes
Decision (D1)	NA	13% (1/8) decision outcomes
MCDC (C1)	NA	0% (0/9) conditions reversed the outcome

Condition table analysis (missing values are in parentheses)

x is equal to 1	XEQ1: x == 1	T (F)	F (TF)	F (TF)	-	-
y is equal to 1	YEQ1: y == 1	F (T)	T (TF)	F (TF)	-	-
z is equal to 1	ZEQ1: z == 1	F (T)	F (TF)	T (TF)	T (TF)	-
	Actions	A1 (F)	A2 (TF)	A3 (TF)	A4 (TF)	A5

The **Coverage (this object)** column shows no coverage. The reason is that the container object for the truth table function—the Stateflow chart—does not decide whether to execute the `ttable` truth table.

The **Coverage (inc. descendants)** column shows coverage for the graphical function. The graphical function has the decision logic that makes the transitions for the truth table. The transitions in the graphical function contain the decisions and conditions of the truth table. Coverage for the descendants in the **Coverage (inc. descendants)** column includes coverage for these conditions and decisions. Function calls to the truth table test the model coverage of these conditions and decisions.

Note: See “How Stateflow Generates Content for Truth Tables” for a description of the graphical function for a truth table.

Coverage for the decisions and their individual conditions in the `ttable` truth table function are as follows.

Coverage	Explanation
No model coverage for the default decision, D5	All logic that leads to taking a default decision is based on a false outcome for all preceding decisions. This means that the default decision requires no logic, so there is no model coverage.
13% (1/8) decision coverage	<p>The three constants that are inputs to the truth table (1, 0, 0) cause only decision <i>D1</i> to be true. These inputs satisfy only one of the eight decisions (<i>D1</i> through <i>D4</i>, T or F).</p> <p>Because each condition can have an outcome value of T or F, three conditions can have six possible values. However, decision <i>D4</i> has only decision coverage, not condition coverage or MCDC coverage, because it represents a decision with a single predicate.</p>
3 of the 18 (17%) condition coverage	Three decisions <i>D1</i> , <i>D2</i> , and <i>D3</i> have condition coverage, because the set of inputs (1, 0, 0) make only decision <i>D1</i> true.
No (0/9) MCDC coverage	MCDC coverage looks for decision reversals that occur because one condition outcome changes from T to F or F to T. The simulation tests only one set of inputs, so the model reverses no decisions.
Missing coverage	The red letters T and F indicate that model coverage is missing for those conditions. For decision <i>D1</i> , only the T decision is satisfied. For decisions <i>D2</i> , <i>D3</i> , and <i>D4</i> , none of the conditions are satisfied.

Colored Stateflow Chart Coverage Display

The Model Coverage tool displays model coverage results for individual blocks directly in Simulink diagrams. If you enable this feature, the Model Coverage tool:

- Highlights Stateflow objects that receive model coverage during simulation
- Provides a context-sensitive display of summary model coverage information for each object

Note: The coverage tool changes colors only for open charts at the time coverage information is reported. When you interact with the chart, such as selecting a transition or a state, colors revert to default values.

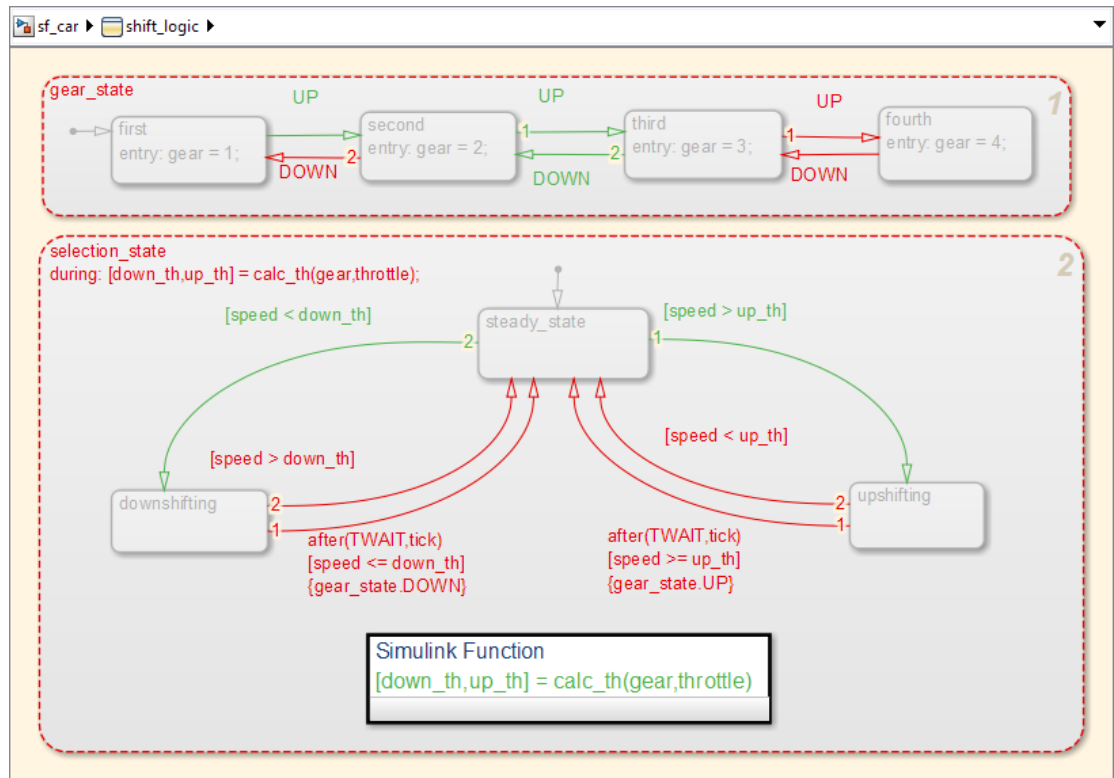
For details on enabling and selecting this feature in the Simulink window, see “Enable Coverage Highlighting” on page 20-8 in the Simulink Verification and Validation documentation.

Display Model Coverage with Model Coloring

Once you enable display coverage with model coloring, anytime that the model generates a model coverage report, individual chart objects receiving coverage appear highlighted with light green or light red.

- 1 Open the `sf_car` model.
- 2 Select **Analysis > Coverage > Settings**.
- 3 In the **Coverage** pane of the Configuration Parameters dialog box, select **Enable coverage analysis**.
- 4 In the **Coverage > Results** pane, select **Display coverage results using model coloring**.
- 5 Click **OK**.
- 6 Simulate the model.

After simulation ends, chart objects with coverage appear highlighted.



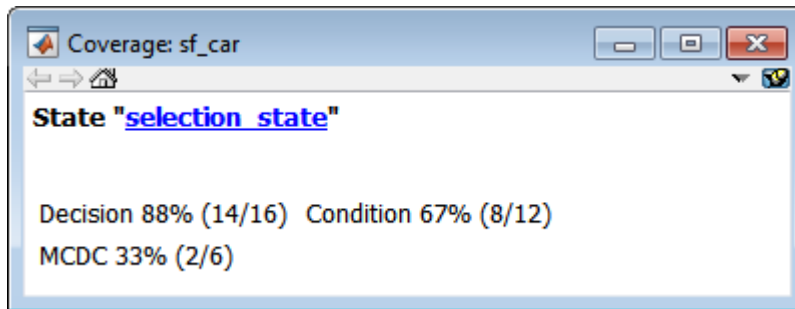
Object highlighting indicates coverage as follows:

- Light green for full coverage
- Light red for partial coverage
- No color for zero coverage

Note: To revert the chart to show original colors, select and deselect any objects.

7 Click `selection_state` in the chart.

The following summary report appears.



When you click a highlighted Stateflow object, the summarized coverage for that object appears in the Coverage Display Window. Clicking the hyperlink opens the section of the coverage report for this object.

Tip You can set the Coverage Display Window to appear for a block in response to a hovering mouse cursor instead of a mouse click in one of two ways:

- Select the downward arrow on the right side of the Coverage Display Window and select **Focus**.
 - Right-click a colored block and select **Coverage > Display details on mouse-over**.
-

Results Review

- “Types of Coverage Reports” on page 21-2
- “Top-Level Model Coverage Report” on page 21-12
- “Export Model Coverage Web View” on page 21-47

Types of Coverage Reports

If you choose to generate a coverage report automatically after analysis from the **Coverage > Results** pane of the Configuration Parameters dialog box or you generate a report from the Results Explorer, the Simulink Verification and Validation software creates one or more model coverage reports after a simulation.

Report Type	Description	HTML Report File Name
“Top-Level Model Coverage Report” on page 21-12	Provides coverage information for all model elements, including the model itself.	<i>model_name_cov.html</i>
“Model Summary Report” on page 21-3	Provides links to coverage results for referenced models and external MATLAB files in the model hierarchy. Created when the top-level model includes Model blocks or calls one or more external files.	<i>model_name_summary_cov.html</i>
“Model Reference Coverage Report” on page 21-4	Created for each referenced model in the model hierarchy; has the same format as the model coverage report.	<i>reference_model_name_cov.html</i>
“External MATLAB File Coverage Report” on page 21-5	Provides detailed coverage information about any external MATLAB file that the model calls. There is one report for each external file called from the model.	<i>MATLAB_file_name_cov.html</i>
“Subsystem Coverage Report” on page 21-9	Model coverage report includes only coverage results for the subsystem, if you select one.	<i>model_name_cov.html</i> ; <i>model_name</i> is the name of the top-level model
“Code Coverage Report” on page 21-11	Provides coverage information for C/C++ code	<i>model_name_block_name_instance_n_</i> or <i>model_name_cov.html</i>

Report Type	Description	HTML Report File Name
	in S-Function blocks, or for models in SIL mode.	

Model Summary Report













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

The summary report lists and provides links to coverage reports for Model block referenced models and external files called by MATLAB code in the model. For more information, see “External MATLAB File Coverage Report” on page 21-5.

The following graphic shows an example of a model summary report. It contains links to the model coverage report (mExternalMfile), a report for the Model block (mExternalMfileRef), and three external files called from the model (externalmfile,I externalmfile1, andexternalmfile2).

Coverage Report by Model

Top Model: mExternalMfile

	Complexity	Decision	Condition	MCDC
TOTAL COVERAGE		90% 	75% 	25% 
1. . . . mExternalMfile	5	50% 	--	--
2. . . . externalmfile1	5	88% 	75% 	0% 
3. . . . mExternalMfileRef	3	100% 	--	--
4. . . . externalmfile	5	100% 	75% 	50% 
5. . . . externalmfile2	2	100% 	--	--

The following models have signal range coverage:

[mExternalMfile](#)

[mExternalMfileRef](#)

Model Reference Coverage Report

If your top-level model references a model in a Model block, the software creates a separate report, named *reference_model_name_cov.html*, that includes coverage for the referenced model. This report has the same format as the “Top-Level Model Coverage Report” on page 21-12. Coverage results are recorded as if the referenced model was a standalone model; the report gives no indication that the model is referenced in a Model block.

External MATLAB File Coverage Report

If your top-level model calls any external MATLAB files, select **MATLAB files** on the **Coverage** pane of the Configuration Parameters dialog box. The software creates a report, named *MATLAB_file_name_cov.html*, for each distinct file called from the model. When there are several calls to a given file from the model, the software creates only one report for that file, but it accumulates coverage from all the calls to the file. The external MATLAB file coverage report does not include information about what parts of the model call the external file.

The first section of the external MATLAB file coverage report contains summary information about the external file, similar to the model coverage report.

Coverage Report for externalmfile1

Table of Contents

1. [Analysis Information](#)
2. [Tests](#)
3. [Summary](#)
4. [Details](#)

Analysis Information

MATLAB Function File Information

Last saved 13-Nov-2008 12:39:55

Simulation Optimization Options

Inline parameters off
 Block reduction forced off
 Conditional branch optimization on

Coverage Options

Analyzed model externalmfile1
 Logic block short circuiting off

Tests

Test 1

Started execution 20-Dec-2013 15:45:08
 Ended execution 20-Dec-2013 15:45:09

Summary

Model Hierarchy/Complexity	Test 1		
	D1	C1	MCDC
1. externalmfile1	5 88%	75%	0%

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

Details

1. MATLAB Function file "[externalmfile1](#)"

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	5
Condition (C1)	NA	75% (3/4) condition outcomes
Decision (D1)	NA	88% (7/8) decision outcomes
MCDC (C1)	NA	0% (0/2) conditions reversed the outcome

MATLAB Function "[externalmfile1](#)"

Parent: [externalmfile1](#)

Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	4
Condition (C1)	75% (3/4) condition outcomes
Decision (D1)	88% (7/8) decision outcomes
MCDC (C1)	0% (0/2) conditions reversed the outcome

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

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


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

#2: function y = externalmfile1(u)

Decisions analyzed

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

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

Decisions analyzed

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

Subsystem Coverage Report

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

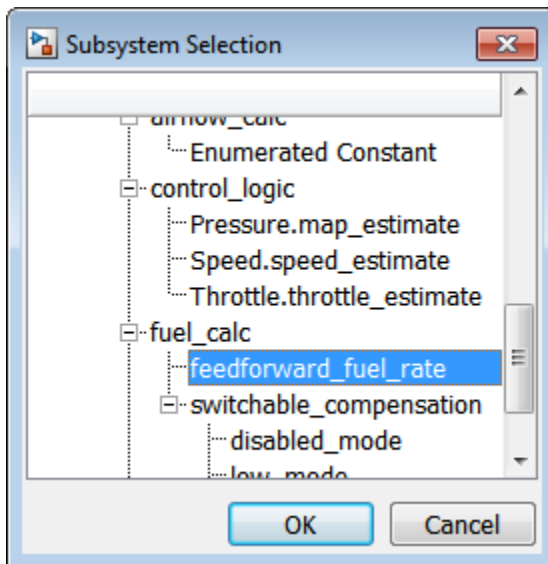
However, if the top-level model calls any external files and you select **MATLAB files** in the **Coverage** pane of the Configuration Parameters dialog box, the results include coverage for all external files called from:

- The subsystem for which you are recording coverage

- The top-level model that includes the subsystem

If the subsystem parameter **Read/Write Permissions** is set to `NoReadOrWrite`, the software does not record coverage for that subsystem.

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



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

Summary

Model Hierarchy/Complexity: **Test 1**

D1

1. [feedforward_fuel_rate](#) 3 33% 

Details:

1. SubSystem block "[feedforward_fuel_rate](#)"

Parent: [sldemo_fuelsys/fuel_rate_control/fuel_calc](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	3
Decision (D1)	NA	33% (1/3) decision outcomes

Code Coverage Report

For each S-Function block, the model coverage report links to a detailed code coverage report for the C/C++ code in the block. For more information on how to navigate the report, see “View Coverage Results for C/C++ Code in S-Function Blocks” on page 20-43.

If you have Embedded Coder installed, you can also generate code coverage reports from models in SIL or PIL mode. For more information on how to generate code coverage reports for models in SIL or PIL mode, see “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode” on page 19-6.

Top-Level Model Coverage Report

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

In this section...
“Coverage Summary” on page 21-12
“Details” on page 21-14
“Cyclomatic Complexity” on page 21-22
“Decisions Analyzed” on page 21-24
“Conditions Analyzed” on page 21-25
“MCDC Analysis” on page 21-26
“Cumulative Coverage” on page 21-27
“N-Dimensional Lookup Table” on page 21-30
“Block Reduction” on page 21-36
“Relational Boundary” on page 21-37
“Saturate on Integer Overflow Analysis” on page 21-41
“Signal Range Analysis” on page 21-42
“Signal Size Coverage for Variable-Dimension Signals” on page 21-44
“Simulink Design Verifier Coverage” on page 21-45

Coverage Summary

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

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

Coverage Report for sldemo_fuelsys

Table of Contents

1. [Analysis Information](#)
2. [Tests](#)
3. [Summary](#)
4. [Details](#)

Analysis Information

Model Information

Model version	1.533
Author	The MathWorks Inc.
Last saved	Wed Nov 20 00:04:05 2013

Simulation Optimization Options

Inline parameters	on
Block reduction	forced off
Conditional branch optimization	on

Coverage Options

Analyzed model	sldemo_fuelsys/fuel_rate_control/control_logic
Logic block short circuiting	off

The coverage summary has two subsections:

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

Tests

Test 1

Started execution 20-Dec-2013 16:30:10

Ended execution 20-Dec-2013 16:30:59

Summary

Model Hierarchy/Complexity	Test 1	D1
1. control_logic	56 25%	
2. SF: control_logic	55 25%	
3. SF: Fail	12 9%	
4. SF: Multi	6 0%	
5. SF: Fueling_Mode	19 28%	
6. SF: Fuel_Disabled	4 0%	
7. SF: Running	10 35%	
8. SF: Low_Emissions	4 50%	
9. SF: O2	5 56%	
10. SF: A	5 56%	
11. SF: Pressure	6 29%	
12. SF: map_estimate	1 0%	
13. Pressure.map_estimate	1 0%	
14. SF: Speed	7 18%	
15. SF: speed_estimate	3 0%	
16. Speed.speed_estimate	3 0%	
17. SF: Throttle	6 29%	
18. SF: throttle_estimate	1 0%	
19. Throttle.throttle_estimate	1 0%	

Details

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

- “Filtered Objects” on page 21-15
- “Model Details” on page 21-15
- “Subsystem Details” on page 21-16
- “Block Details” on page 21-17

- “Chart Details” on page 21-18
- “Coverage Details for MATLAB Functions and Simulink Design Verifier Functions” on page 21-18

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

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

Filtered Objects

The Filtered Objects section lists all the objects in the model that were filtered from coverage recording, and the rationale you specified for filtering those objects. If the filter rule specifies that all blocks of a certain type be filtered, all those blocks are listed here.

In the following graphic, several blocks, subsystems, and transitions were filtered. Two library-linked blocks, protected division and protected division1, were filtered because their block library was filtered.

Blocks Eliminated from Coverage Analysis

Model Object	Rationale
slvndemo covfilt/Saturation	It might not be executed because of Conditional input branch optimization
slvndemo covfilt/protected division/Compare To Zero/Compare	It might not be executed because of Conditional input branch optimization
slvndemo covfilt/protected division/Switch	It might not be executed because of Conditional input branch optimization
slvndemo covfilt/protected division/Switch1	It might not be executed because of Conditional input branch optimization
slvndemo covfilt/protected division1/Switch	It might not be executed because of Conditional input branch optimization

Model Details

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

The following graphic shows the Details section for the `sldemo_fuelsys` example model.

1. Model "sldemo_fuelsys"

Child Systems: [Engine Gas Dynamics](#), [fuel_rate_control](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	76
Condition (C1)	NA	34% (11/32) condition outcomes
Decision (D1)	NA	32% (37/114) decision outcomes
MCDC (C1)	NA	7% (1/14) conditions reversed the outcome
Lookup Table	NA	1% (13/1511)interpolation/extrapolation intervals
Relational Boundary	NA	13% (6/45) objective outcomes
Saturation on integer overflow	NA	50% (10/20) objective outcomes

Subsystem Details

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

The following graphic shows the coverage results for the Engine Gas Dynamics subsystem in the sldemo_fuelsys example model.

2. SubSystem block "[Engine Gas Dynamics](#)"

Parent: [/sldemo_fuelsys](#)

Child Systems: [Mixing & Combustion](#), [Throttle & Manifold](#)


Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	13
Decision (D1)	NA	71% (10/14) decision outcomes

Block Details

The following graphic shows decision coverage results for the MinMax block in the Mixing & Combustion subsystem of the Engine Gas Dynamics subsystem in the sldemo_fuelsys example model.

MinMax block "[MinMax](#)"

Parent: [sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion](#)

Uncovered Links: 

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

Decisions analyzed:

Logic to determine output	50%
input 1 is the maximum	204508/204508
input 2 is the maximum	0/204508

The *Uncovered Links* element first appears in the Block Details section of the first block in the model hierarchy that does not achieve 100% coverage. The first *Uncovered Links* element has an arrow that links to the Block Details section in the report of the *next* block that does not achieve 100% coverage.

Subsequent blocks that do not achieve 100% coverage have links to the Block Details sections in the report of the previous and next blocks that do not achieve 100% coverage.

Saturate block "[Limit to Positive](#)"

Parent: [sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold](#)



Uncovered Links:  

Chart Details

The following graphic shows the coverage results for the Stateflow chart `control_logic` in the `sldemo_fuelsys` example model.

13. SubSystem block "[control_logic](#)"

Parent: [sldemo_fuelsys/fuel_rate_control](#)

Child Systems: [control_logic](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	56
Condition (C1)	NA	21% (5/24) condition outcomes
Decision (D1)	NA	25% (23/92) decision outcomes
MCDC (C1)	NA	0% (0/12) conditions reversed the outcome
Look-up Table	NA	0% (0/1082)interpolation/extrapolation intervals

14. Chart "[control_logic](#)"

Parent: [sldemo_fuelsys/fuel_rate_control/control_logic](#)

Child Systems: [Fail](#), [Fueling Mode](#), [O2](#), [Pressure](#), [Speed](#), [Throttle](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	55
Condition (C1)	NA	21% (5/24) condition outcomes
Decision (D1)	NA	25% (23/92) decision outcomes
MCDC (C1)	NA	0% (0/12) conditions reversed the outcome
Look-up Table	NA	0% (0/1082)interpolation/extrapolation intervals

For more information about model coverage reports for Stateflow charts and their objects, see "Model Coverage for Stateflow Charts" on page 20-48.

Coverage Details for MATLAB Functions and Simulink Design Verifier Functions

By default, Simulink Verification and Validation records coverage for all MATLAB functions in a model. MATLAB functions are in MATLAB Function blocks, Stateflow charts, or external MATLAB files.

Note: For a detailed example of coverage reports for external MATLAB files, see “External MATLAB File Coverage Report” on page 21-5.

To record Simulink Design Verifier coverage for `sldv.*` functions called by MATLAB functions, and any Simulink Design Verifier blocks, select **Objectives and Constraints** on the **Coverage** pane of the Configuration Parameters dialog box.

The following example shows coverage details for a MATLAB function, `hFcnsInExternalEML`, that calls four Simulink Design Verifier functions. In this example, the code for `hFcnsInExternalEML` resides in an external file.

This example also shows Simulink Design Verifier coverage details for the following functions:

- `sldv.assume`
- `sldv.condition`
- `sldv.prove`
- `sldv.test`

In the coverage results, code that achieves 100% coverage is green. Code that achieves less than 100% coverage is red.

Embedded MATLAB function "[hfcnsinexternalem1](#)"

Parent: [hfcnsinexternalem1](#)

Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	4
Decision (D1)	40% (2/5) decision outcomes
Test Objective	50% (1/2) objective outcomes
Proof Objective	0% (0/1) objective outcomes
Test Condition	100% (1/1) objective outcomes
Proof Assumption	0% (0/1) objective outcomes

```

1 function y = hFcnsInExternaleM1(u1, u2)
2 % use all four functions.
3 %#eml
4 sldv.assume(u1 > u2);
5 sldv.condition(u1 == 0);
6 switch u1
7     case 0
8         y = u2;
9     case 1
10        y = 3;
11    case 2
12        y = 0;
13    otherwise
14        y = 0;
15        sldv.prove(u2 < u1);
16 end
17 sldv.test(y > u1); sldv.test(y == 4);
18

```

Coverage for the hFcnsInExternaleM1 function and the sldv.* calls is:

- Line 1, the function declaration for hFcnsInExternaleM1 is green because the simulation executes that function at least once. fcn calls hFcnsInExternaleM1 11 times during simulation.

#1: [function y = hFcnslnExternalEML\(u1, u2\)](#)**Decisions analyzed:**

function y = hFcnslnExternalEML(u1, u2)	100%
executed	11/11

Line 4, `sldv.assume(u1 > u2)`, achieves 0% coverage because `u1 > u2` never evaluates to true.

#4: [sldv.assume\(u1 > u2\);](#)**Proof Assumption analyzed:**

<code>sldv.assume(u1 > u2)</code>	0/11
--------------------------------------	------

- Line 5, `sldv.condition(u1 == 0)`, achieves 100% coverage because `u1 == 0` evaluates to true for at least one time step.

#5: [sldv.condition\(u1 == 0\);](#)**Test Condition analyzed:**

<code>sldv.condition(u1 == 0)</code>	11/11
--------------------------------------	-------

- Line 6, `switch u1`, achieves 25% coverage because only one of the four outcomes in the `switch` statement (`case 0`) occurs during simulation.

[#6: switch u1](#)

Decisions analyzed:

switch u1	25%
otherwise	0/11
case 0	11/11
case 1	0/11
case 2	0/11

- Line 17, `sldv.test(y > u1); sldv.test (y == 4)` achieves 50% coverage. The first `sldv.test` call achieves 100% coverage, but the second `sldv.test` call achieves 0% coverage.

[#17: sldv.test\(y > u1\); sldv.test\(y == 4\);](#)

Test Objective analyzed:

<code>sldv.test(y > u1)</code>	11/11
<code>sldv.test(y == 4)</code>	0/11

For more information about coverage for MATLAB functions, see “Model Coverage for MATLAB Functions” on page 20-23.

For more information about coverage for Simulink Design Verifier functions, see “Objectives and Constraints Coverage” on page 16-7.

Cyclomatic Complexity

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

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

Summary

Model Hierarchy/Complexity:

1. fuelsys	78
2. . . . engine gas dynamics	5
3. Mixing & Combustion	1
4. Throttle & Manifold	4
5. Throttle	2
6. . . . fuel rate controller	72
7. Airflow calculation	1
8. Fuel Calculation	11
9. Switchable Compensation	7
10. LOW Mode	2
11. RICH Mode	2
12. Sensor correction and Fault Redundancy	9
13. MAP Estimate	2
14. Speed Estimate	2
15. Throttle Estimate	2
16. control logic	51
17. SF: control logic	50
18. SF: Fail	12
19. SF: Multi	6
20. SF: Fueling Mode	19
21. SF: Fuel Disabled	4
22. SF: Running	10
23. SF: Low Emissions	4
24. SF: O2	5
25. SF: Pressure	5
26. SF: Speed	4
27. SF: Throttle	5

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

4. SubSystem block "[Throttle & Manifold](#)"

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

Child Systems: [Throttle](#)

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


Decisions Analyzed

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

The following graphic shows the Decisions analyzed table for the Saturate block in the Throttle & Manifold subsystem of the Engine Gas Dynamics subsystem in the sldemo_fuelsys example model.

Saturate block "[Limit to Positive](#)"

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

Uncovered Links: 

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

Decisions analyzed:

input > lower limit	50%
false	0/204508
true	204508/204508
input >= upper limit	50%
false	204508/204508
true	0/204508

To display and highlight the block in question, click the block name at the top of the section containing the block's Decisions analyzed table.



Conditions Analyzed

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

Conditions analyzed:

Description:	True	False
input port 1	199521	480
input port 2	200001	0

MCDC Analysis

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

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

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

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

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

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

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

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

Parentheses around the expression indicate that the specified combination of inputs did not occur during the first (or only) test case included in this report. In other words, the test case did not cover the corresponding condition case. The *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** pane of the Configuration Parameters dialog box, MC/DC coverage analysis does not verify whether short-circuited inputs actually occur. The MC/DC analysis table uses an x in a condition expression (for example, TFxxx) to indicate short-circuited inputs that were not analyzed by the tool.

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

Select the **Treat Simulink Logic blocks as short-circuited** option for where you want the MCDC coverage analysis to approximate the degree of coverage that your test cases achieve for the generated code (most high-level languages short-circuit logic expressions).

Cumulative Coverage

After you record successive coverage results, you can “Access, Manage, and Accumulate Coverage Results” on page 18-12 from within the Coverage Results Explorer. By default, the results of each simulation are saved and recorded cumulatively in the report.

In a cumulative coverage report, the results located in the right-most area in all tables reflect the running total value. The report is organized so that you can easily compare

the additional coverage from the most recent run with the coverage from all prior runs in the session.

A cumulative coverage report contains information about:

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

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

Summary

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

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

Decisions analyzed:

Transition trigger expression	100%	50%	100%
false	1097/1098	1097/1097	1097/1100
true	1/1098	0/1097	3/1100

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

Conditions analyzed:

Description:	#1 T	#1 F	#2 T	#2 F	Tot T	Tot F
Condition 1, "alt_ctrl"	1	1097	0	1097	3	1097
Condition 2, "wow"	0	1	0	0	0	3
Condition 3, "in(GS.Active.Coupled)"	0	1	0	0	0	3

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

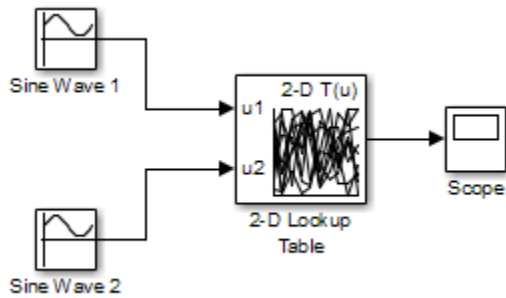
MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	#1 True Out	#1 False Out	#2 True Out	#2 False Out	Total Out T	Total Out F
Transition trigger expression						
Condition 1, "alt_ctrl"	TFF	Fxx	(TFF)	Fxx	TFF	Fxx
Condition 2, "wow"	TFF	(TTx)	(TFF)	(TTx)	TFF	(TTx)
Condition 3, "in(GS.Active.Coupled)"	TFF	(TFT)	(TFF)	(TFT)	TFF	(TFT)

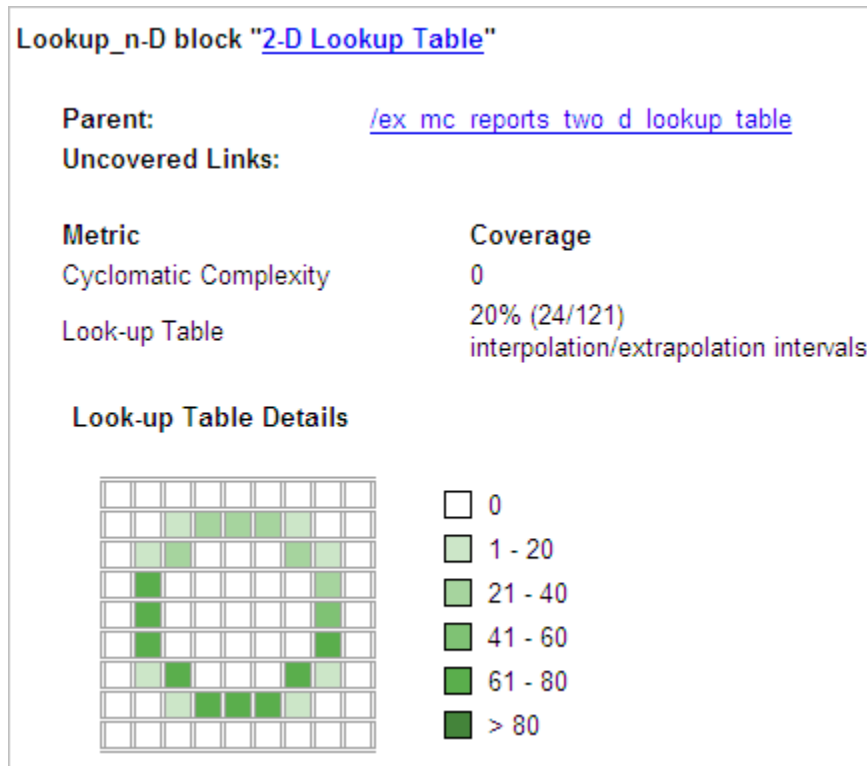
Note: You can calculate cumulative coverage for reusable subsystems and Stateflow constructs at the command line. For more information, see “Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs” on page 23-8.

N-Dimensional Lookup Table

The following interactive chart summarizes the extent to which elements of a lookup table are accessed. In this example, two Sine Wave blocks generate x and y indices that access a 2-D Lookup Table block of 10-by-10 elements filled with random values.



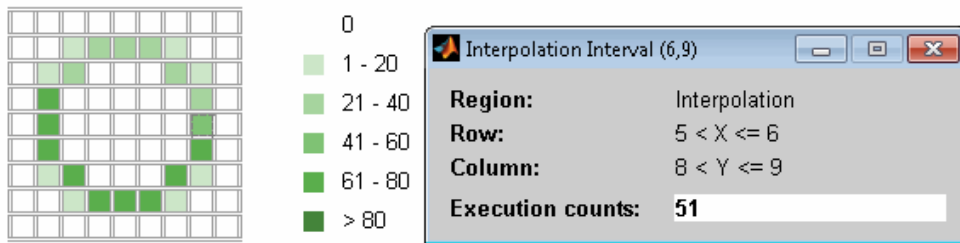
In this model, the lookup table indices are 1, 2,..., 10 in each direction. The Sine Wave 2 block is out of phase with the Sine Wave 1 block by $\pi/2$ radians. This generates x and y numbers for the edge of a circle, which you see when you examine the resulting Lookup Table coverage.



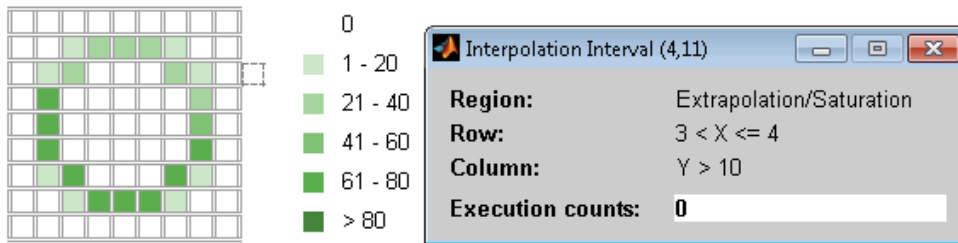
The report contains a two-dimensional table representing the elements of the lookup table. The element indices are represented by the cell border grid lines, which number 10 in each dimension. Areas where the lookup table interpolates between table values are represented by the cell areas. Areas of extrapolation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

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

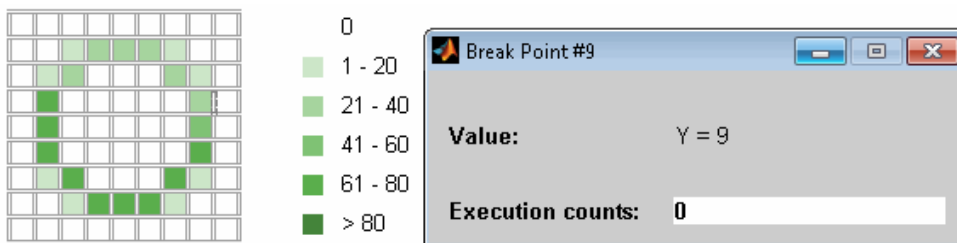
If you click an individual table cell, you see a dialog box that displays the index location of the cell and the exact number of execution counts generated for it during testing. The following example shows the contents of a color-shaded cell on the right edge of the circle.



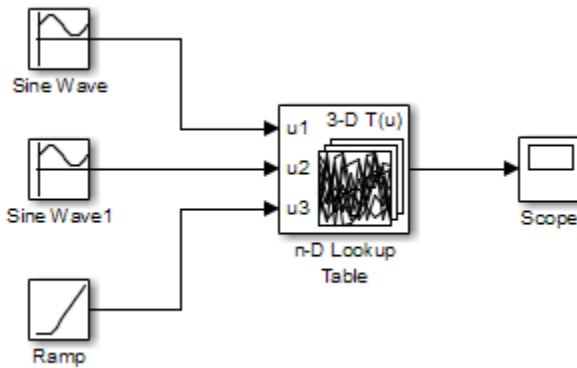
The selected cell is outlined in red. You can also click the extrapolation cells on the edge of the table.



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



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



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

After simulation, you see the following lookup table report.

Lookup_n-D block "[n-D Lookup Table](#)"

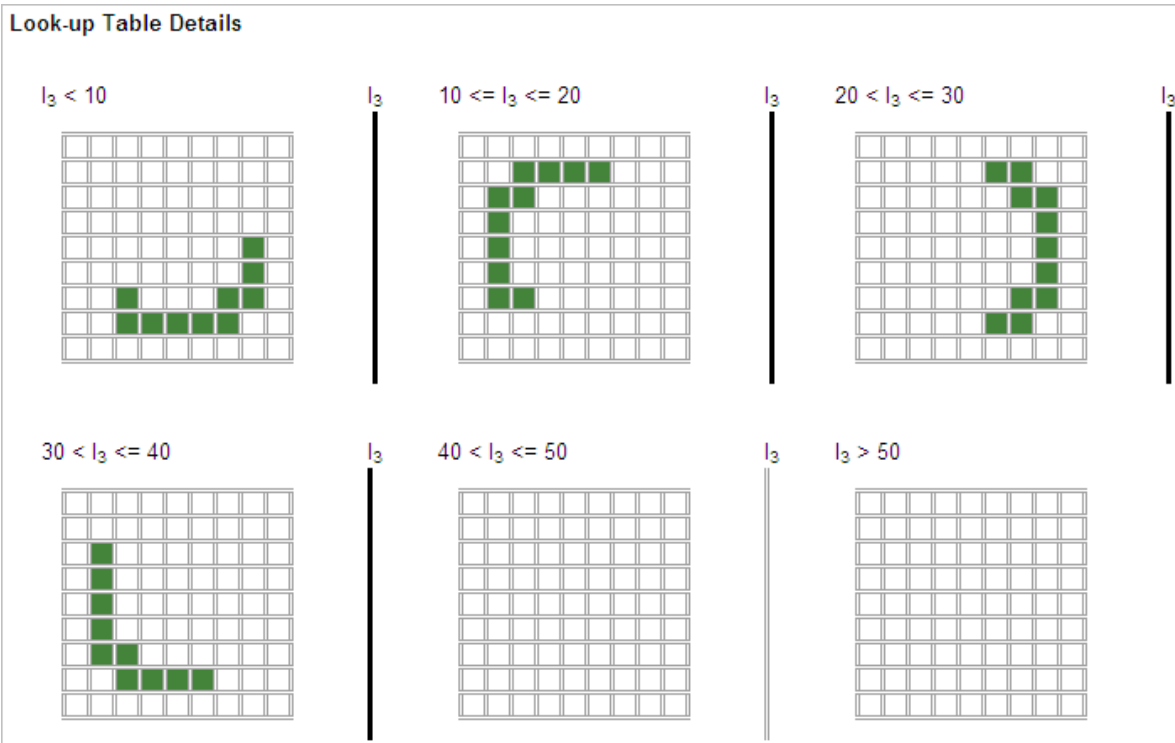
Parent: [/ex mc reports three d lookup table](#)

Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	0
Look-up Table	6% (42/726) interpolation/extrapolation intervals

Table map was not generated due to the table size.
[Force Map Generation.](#)

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



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

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

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

Block Reduction

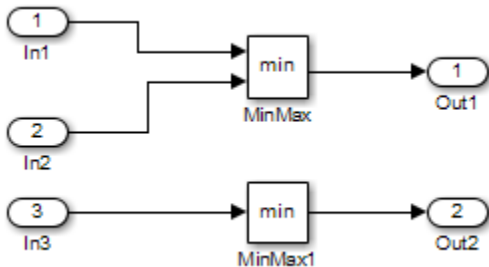
All model coverage reports indicate the status of the Simulink **Block reduction** parameter at the beginning of the report. In the following example, you set **Force block reduction off**.

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

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

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

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



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

MinMax block " MinMax1 "	
Parent:	/ex_minmax_coverage
Uncovered Links:	◀
Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	0% (0/1) decision outcomes
Decisions analyzed:	
Logic to determine output	0%
input 1 is the minimum	0/0

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

Reduced Blocks
Blocks eliminated from coverage analysis by block reduction model simulation setting:
... ex_minmax_coverage/MinMax1

Relational Boundary

On the "Coverage Pane" on page 18-2 of the Configuration Parameters dialog box, if you select the **Relational Boundary** coverage metric, the software creates a Relational Boundary table in the model coverage report for each model object that is supported for

this coverage. The table applies to the explicit or implicit relational operation involved in the model object. For more information, see:

- “Relational Boundary Coverage” on page 16-8.
- The **Relational Boundary** column in “Model Objects That Receive Coverage” on page 17-2.

The tables below show the relational boundary coverage report for the relation `input1 <= input2`. The appearance of the tables depend on the operand data type.

- “Integers” on page 21-38
- “Fixed point” on page 21-39
- “Floating point” on page 21-39

Integers

If both operands are integers (or if one operand is an integer and the other a Boolean), the table appears as follows.

<code>input1 - input2</code>	33%
-1	0/51
0	51/51
+1	0/51

For a relational operation such as `operand_1 <= operand_2`:

- The first row states the two operands in the form `operand_1 - operand_2`.
- The second row states the number of times during the simulation that `operand_1 - operand_2` is equal to -1.
- The third row states the number of times during the simulation that `operand_1` is equal to `operand_2`.

- The fourth row states the number of times during the simulation that *operand_1* - *operand_2* is equal to 1.

Fixed point

If one of the operands has fixed-point type and the other operand is either a fixed point or an integer, the table appears as follows. **LSB** represents the value of the least significant bit. For more information, see “Precision”. If the two operands have different precision, the smaller value of precision is used.

Relational Boundary

input1 - input2	33%
-LSB	51/51
0	0/51
+LSB	0/51

For a relational operation such as *operand_1* <= *operand_2*:

- The first row states the two operands in the form *operand_1* - *operand_2*.
- The second row states the number of times during the simulation that *operand_1* - *operand_2* is equal to -LSB.
- The third row states the number of times during the simulation that *operand_1* is equal to *operand_2*.
- The fourth row states the number of times during the simulation that *operand_1* - *operand_2* is equal to LSB.

Floating point

If one of the operands has floating-point type, the table appears as follows. **tol** represents a value computed using the input values and a tolerance that you specify. If you do not specify a tolerance, the default values are used. For more information, see “Relational Boundary Coverage” on page 16-8.

Relational Boundary

input1 - input2	50%
[-tol..0]	51/51
(0..tol]	0/51

For a relational operation such as $operand_1 \leq operand_2$:

- The first row states the two operands in the form $operand_1 - operand_2$.
- The second row states the number of times during the simulation that $operand_1 - operand_2$ has values in the range $[-tol..0]$.
- The third row states the number of times during the simulation that $operand_1 - operand_2$ has values in the range $(0..tol]$ during the simulation.

The appearance of this table changes according to the relational operator in the block. Depending on the relational operator, the value of $operand_1 - operand_2$ equal to 0 is either:

- Excluded from relational boundary coverage.
- Included in the region above the relational boundary.
- Included in the region below the relational boundary.

Relational Operator	Report Format	Explanation
==	[-tol..0]	0 is excluded.
	(0..tol]	
!=	[-tol..0]	0 is excluded.
	(0..tol]	
<=	[-tol..0]	0 is included in the region below the relational boundary.
	(0..tol]	

Relational Operator	Report Format	Explanation
<	[-tol..0)	0 is included in the region above the relational boundary.
	[0..tol]	
>=	[-tol..0)	0 is included in the region above the relational boundary.
	[0..tol]	
>	[-tol..0)	0 is included in the region below the relational boundary.
	(0..tol]	

0 is included below the relational boundary for <= but above the relational boundary for <. This rule is consistent with decision coverage. For instance:

- For the relation `input1 <= input2`, the decision is true if `input1` is less than or equal to `input2`. < and = are grouped together. Therefore, 0 lies in the region below the relational boundary.
- For the relation `input1 < input2`, the decision is true only if `input1` is less than `input2`. > and = are grouped together. Therefore, 0 lies in the region above the relational boundary.

Saturate on Integer Overflow Analysis

On the “Coverage Pane” on page 18-2 of the Configuration Parameters dialog box, if you select the **Saturate on integer overflow** coverage metric, the software creates a Saturation on Overflow analyzed table in the model coverage report. The software creates the table for each block with the **Saturate on integer overflow** parameter selected.

The Saturation on Overflow analyzed table lists the number of times a block saturates on integer overflow, indicating a true decision. If the block does not saturate on integer overflow, the table indicates a false decision. Outcomes that do not occur are in red highlighted table rows.

The following graphic shows the Saturation on Overflow analyzed table for the MinMax block in the Mixing & Combustion subsystem of the Engine Gas Dynamics subsystem in the `sldemo_fuelsys` example model.

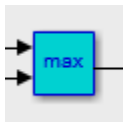
MinMax block "[MinMax](#)"**Parent:** [sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion](#)**Uncovered Links:** ➔

Metric	Coverage
Cyclomatic Complexity	0
Saturation on Overflow	50% (1/2) objective outcomes

Saturation on Overflow analyzed:

Saturate on integer overflow	50%
false	204508/204508
true	0/204508

To display and highlight the block in question, click the block name at the top of the section containing the block's Saturation on Overflow analyzed table.

**Signal Range Analysis**

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

Access the Signal Range Analysis report quickly with the *Signal Ranges* link in the nonscrolling region at the top of the model coverage report, as shown below in the sldemo_fuelsys example model report.

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

Signal Ranges:

Hierarchy	Min	Max
sldemo_fuelsys		
... Constant2	0	0
... Constant3	12	12
... Constant4	0	0
... Constant5	0	0
... EGO Sensor	0.456832	1
... Engine Speed Selector	300	300
... High Speed	700	700
... MAP Sensor	0.405559	0.889674
... Nominal Speed	300	300
... Speed Sensor	300	300
... Throttle Sensor	10	20

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



Signal Size Coverage for Variable-Dimension Signals

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

The following example shows the Variable Signal Widths section in a coverage report. In this example, the Abs block signal size varied from 2 to 5, with an allocation of 5.

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

Each block is reported in hierarchical fashion; child blocks appear directly under parent blocks. Each block name in the Variable Signal Widths list is a link. Clicking on the link highlights the corresponding block in the Simulink Editor. After the analysis, the variable-size signals have a wider line design.

Simulink Design Verifier Coverage

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

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

This model contains two Test Objective blocks:

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

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

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

Design Verifier Test Objective block "True"

Parent: [sldvdemo_debounce_testobjblks_harness/Test Unit \(copied from sldvdemo_debounce_testobjblks\)](#)

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

Points/Intervals analyzed:

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

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

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

Parent: [sldvdemo_debounce_testobjblks_harness/Test Unit \(copied from sldvdemo_debounce_testobjblks\)/Masked Objective](#)

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

Points/Intervals analyzed:

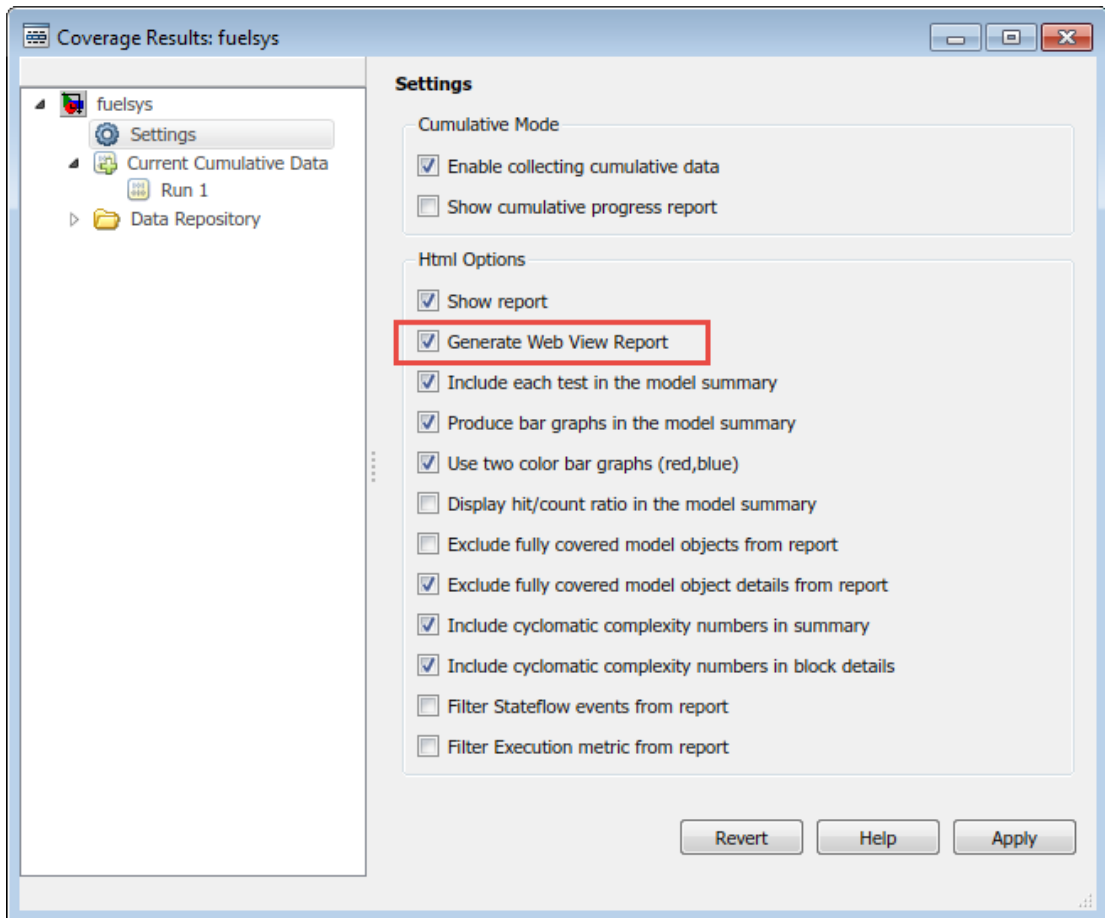
Point : T

1/14

Export Model Coverage Web View

You can export a Model Coverage Web View for your model. A Web View is an interactive rendition of a model that you can view in a Web browser. A Model Coverage Web View includes model coverage highlighting and analysis information from the Coverage Display Window, as described in “View Coverage Results in a Model” on page 20-7.

Use the Results Explorer to generate a Model Coverage Web View. After you record coverage, select **Analysis > Coverage > Open Results Explorer**. In the Results Explorer, open the **Settings**, select **Generate Web View Report**, and click **Apply**.



Next, select the Current Cumulative Data click **Generate report**.

When you generate a coverage report for your model with these settings enabled, the software generates a Model Coverage Web View that you can open in a browser. To see model coverage information for a block in a Model Coverage Web View, click that block. The model coverage data appears in the **Informer** pane, below the model.

For more information, see “Web Views” in the Simulink Report Generator documentation.

Excluding Model Objects from Coverage

- “Coverage Filtering” on page 22-2
- “Coverage Filter Rules and Files” on page 22-4
- “Model Objects to Filter from Coverage” on page 22-6
- “Create, Edit, and View Coverage Filter Rules” on page 22-7
- “Coverage Filter Viewer” on page 22-12

Coverage Filtering

In this section...
“When to Use Coverage Filtering” on page 22-2
“What Is Coverage Filtering?” on page 22-2

When to Use Coverage Filtering

Use coverage filtering to facilitate a bottom-up approach to recording model coverage. If you have a large model, there can be design elements that intentionally do not record 100% coverage. You can also have several design elements that you require to record 100% coverage but that do not achieve 100% coverage. You can temporarily or permanently eliminate these elements from coverage recording to focus on a subset of objects for testing and modification.

You can then iterate more efficiently—focus on a small issue, fix it, and then move on to resolve the next small issue. Before recording coverage for the entire model, you can resolve missing coverage issues within individual parts of the model.

What Is Coverage Filtering?

Coverage filtering enables you to exclude certain model objects from model coverage reporting after you simulate your Simulink model. You specify which objects you want to filter from coverage recording. There are two modes of filtering, Excluded and Justified.

Excluded objects do not contribute to coverage reports. After you specify the objects to exclude when you simulate your model, the coverage report does not record coverage for those objects.

Justified objects do contribute to coverage reports. After you specify the objects to justify when you simulate your model, the coverage report considers these blocks as achieving 100% coverage, and they appear light blue in the “Coverage Summary” on page 21-12.

Summary

Model Hierarchy/Complexity	Test 1			
	D1	CI	MCDC	Execution
1. slvnydemo_covfilt	29 52%	40%	50%	13%
2. ... Mode Logic	13 86%	75%	50%	NA
3. SF: Mode Logic	12 86%	75%	50%	NA
4. SF: Clipped	6 100%	NA	NA	NA
5. SF: Full	2 25%	NA	NA	NA


In the “Details” on page 21-14 section of the coverage report, justified objects show their coverage outcomes as ((covered outcomes + justified outcomes)/possible decisions).

4. State "[Clipped](#)"

Justified ([Remove this rule](#))

Justification rationale: Justification rationale

Parent: [slvnydemo_covfilt/Mode Logic](#)

Uncovered Links: 

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	2	6
Decision (D1)	100% ((2+2)/4) decision outcomes	100% ((5+7)/12) decision outcomes

To filter objects, see “Create, Edit, and View Coverage Filter Rules” on page 22-7 and “Creating and Using Coverage Filters”.

Coverage Filter Rules and Files

In this section...
“What Is a Coverage Filter Rule?” on page 22-4
“What Is a Coverage Filter File?” on page 22-4

What Is a Coverage Filter Rule?

A coverage filter rule specifies a model object, a set of objects, or lines of code that you want to exclude from coverage recording or that you want to justify for coverage.

Each coverage filter rule includes the following fields:

- **Name**—Name or path of the object to filter from coverage
- **Type**—Whether a specific object is filtered or all objects of a given type are filtered
- **Mode**—Whether the object to be filtered is **Excluded** or **Justified**

Coverage reports do not include **Excluded** blocks. The coverage reports assume that **Justified** blocks receive full coverage, but show that they are distinct from other covered blocks in the coverage report.

- **Rationale**—An optional description that describes why this object is filtered from coverage

What Is a Coverage Filter File?

A coverage filter file is a set of coverage filter rules. Each rule specifies one or more objects or lines of code to exclude from coverage recording.

To apply the coverage filter rules after coverage recording, you create coverage filter rules or load an existing coverage filter file. After you create the coverage filter rules, the specified objects or lines of code are excluded from coverage when you generate a report. You can reuse a coverage filter file for several Simulink models. However, a model can have only one attached coverage filter file.

When you make changes to the coverage filter rules after you record coverage, you can update the coverage report without needing to resimulate your model. After you make changes, click **Apply** and then **Generate Report** in the Coverage Filter Viewer to update the report.

If you use the default file name for the active model, and the coverage filter file exists on the MATLAB path, you see the coverage filter rules each time that you open the model. To save your current coverage filter rules to a file, click **Save filter**. To load an existing coverage filter file, click **Load filter**

Model Objects to Filter from Coverage

In your model, the objects that you can filter from coverage recording are:

- Simulink blocks that receive coverage, including MATLAB Function blocks
- Subsystems and their contents. When you exclude a subsystem from coverage recording, none of the objects inside the subsystem record coverage.
- Individual library-linked blocks or charts
- All reference blocks linked to a library
- Stateflow charts, subcharts, states, transitions, and events

For a complete list of model objects that receive coverage, see “Model Objects That Receive Coverage” on page 17-2.

Create, Edit, and View Coverage Filter Rules

In this section...

- “Create and Edit Coverage Filter Rules” on page 22-7
- “Save Coverage Filter to File” on page 22-9
- “Load Coverage Filter File” on page 22-10
- “Update the Report with the Current Filter Settings” on page 22-10
- “View Coverage Filter Rules in Your Model” on page 22-10
- “View Coverage Filter Rules in Your Model” on page 22-11

Create and Edit Coverage Filter Rules

- “Create a Coverage Filter Rule” on page 22-7
- “Select the Filtering Mode” on page 22-8
- “Add Rationale to a Coverage Filter Rule” on page 22-8

Create a Coverage Filter Rule

To create a coverage filter rule:

- 1 In the **Coverage** pane of the Configuration Parameters dialog box, enable model coverage.
- 2 To record coverage results, simulate the model.
- 3 Create a new filter rule in one of two ways:
 - In the model window, right-click a model object and select **Coverage > Exclude**.
 - In the Details section of the Coverage Report, click **Justify or Exclude** for a model object.

The following table lists the **Exclude** menu options. Depending on which option you select, the **Type** field is set for the coverage filter rule you selected. You cannot override the value in the **Type** field.

If you select Coverage >	The rule type is
Exclude this block	by block path

If you select Coverage >	The rule type is
Exclude all blocks with type <block_type>	by block type
Exclude all blocks with type MATLAB Function	by block type
Exclude all blocks with type Truth Table	by block type
Exclude subsystem with all dependents	by subsystem
Exclude referenced library: <library_name>	by library reference
Exclude subsystem with all descendants	by subsystem
Exclude chart with all descendants	by chart
Exclude mask type <mask name>	by mask type
Exclude state with all descendants	by state
Exclude this transition	by transition
Exclude temporal event <event_name>	by temporal event

Select the Filtering Mode

When you create a filtering rule, the default filtering mode is **Excluded**. Excluded objects do not appear in the coverage reports. You can also set the filtering mode to **Justified**. Justified blocks appear as achieving 100% coverage.

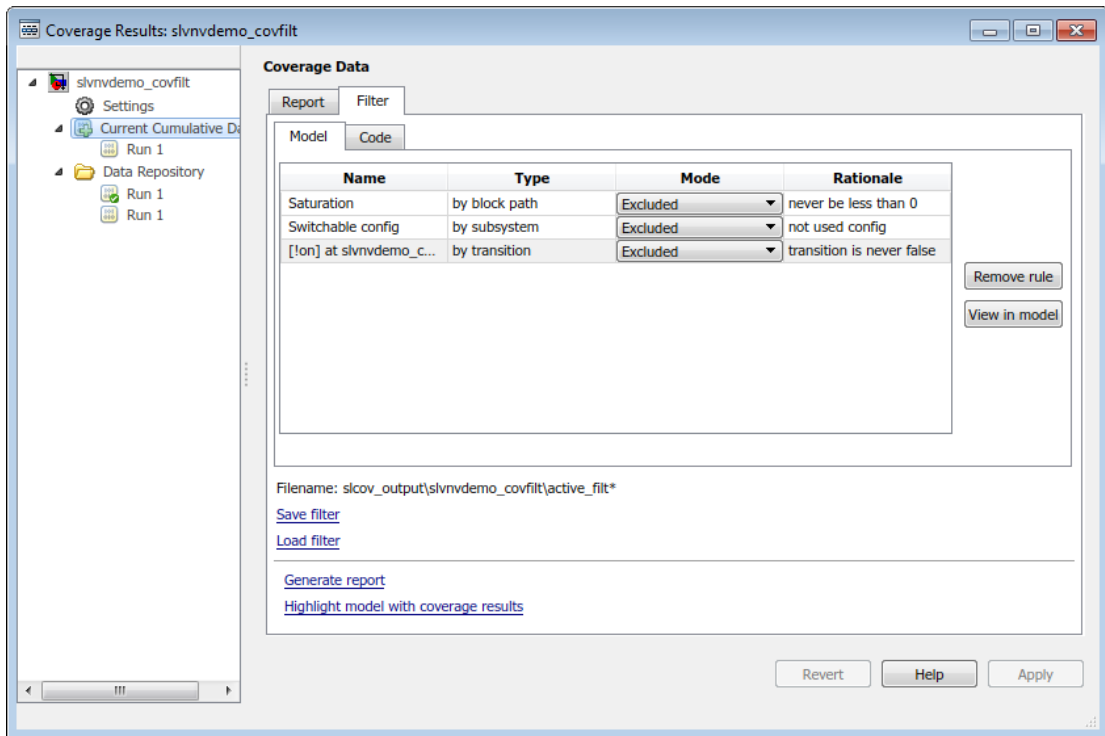
For more information, see “Coverage Filtering” on page 22-2.

Add Rationale to a Coverage Filter Rule

Optionally, you can add text that describes why you exclude that object or objects from coverage recording. This information can be useful to others who review the coverage for your model. When you add a coverage filter rule, the Coverage Filter Viewer opens. To add the rationale:

- 1 Double-click the Rationale field for the rule.
- 2 Delete the existing text.
- 3 Add the rationale for excluding this object.

The following graphic shows examples of text in the **Rationale** field.



Note: The **Rationale** field and **Mode** field are the only coverage filter rule fields that you can edit in the Coverage Filter Viewer.

After you add a new coverage filter rule or edit an existing coverage filter rule, click **Apply** to enable the **Generate report** and **Highlight model with coverage results** links.

Save Coverage Filter to File

After you define the coverage filter rules, save the rules to a file so that you can reuse them with this model or with other models. By default, coverage filter files are named `<model_name>_covfilter.cvf`.

In the Current Cumulative Data section of the Coverage Filter Viewer:

- 1 Click **Save filter**.
- 2 Specify a file name and folder for the filter file and click **Save**.

If you make multiple changes to the coverage filter rules, apply the changes to the coverage filter file each time.

Load Coverage Filter File

After you save a coverage filter file, you can load the coverage filter file for other models.

In the Current Cumulative Data section of the Coverage Filter Viewer:

- 1 Click **Load filter**.
- 2 Navigate to the filter file and click **Open**.

You can have only one coverage filter file attached to a model at a time. If you attach a different coverage filter file, the newly attached file replaces the previously attached file.

Two or more models can have the same coverage filter file attached. If a model has an attached filter file that contains coverage filter rules for specific objects in a different model, those rules are ignored during coverage recording.

Update the Report with the Current Filter Settings

If you change the filtering settings or add filters after you simulate the model, you can update the coverage report and model highlighting without resimulating the model. After you have simulated the model, in the Current Cumulative Data section of the Coverage Filter Viewer:

- 1 **Apply** or **Revert** any changes you have made.
- 2 Click **Generate Report**.

View Coverage Filter Rules in Your Model

Whenever you define a coverage filter rule or remove an existing coverage filter rule, the Coverage Filter Viewer opens. This dialog box lists the coverage filter rules for your model. For more information, see “Coverage Filter Viewer” on page 22-12.

The Coverage Filter Viewer is available in the Current Cumulative Data section of the **Coverage Results** viewer. Alternatively, you can right-click anywhere in the model window and select **Coverage > Open Filter Viewer**

If you are inside a subsystem, you can view any coverage filter rule attached to the subsystem. To open the Coverage Filter Viewer, right-click any object inside the subsystem and select **Coverage > Show filter parent**.

View Coverage Filter Rules in Your Model

Whenever you define a coverage filter rule or remove an existing coverage filter rule, the Coverage Filter Viewer opens. This dialog box lists all the coverage filter rules for your model. For more information, see “Coverage Filter Viewer” on page 22-12.

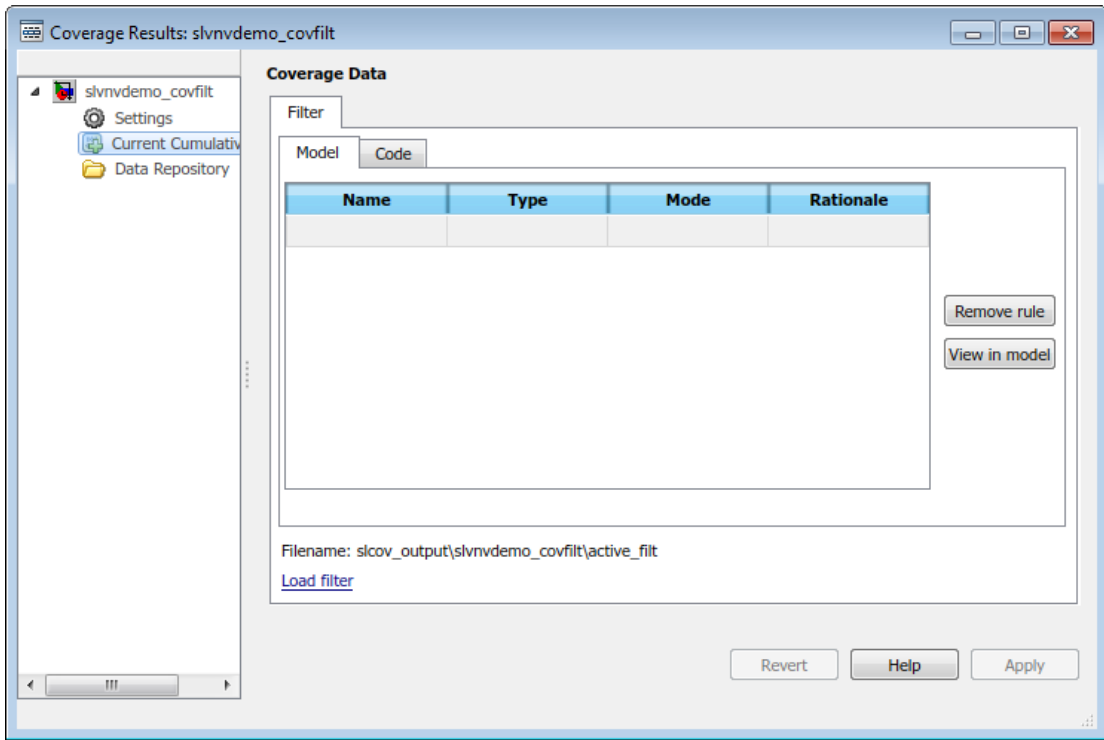
To open the Coverage Filter Viewer, right-click anywhere in the model window and select **Coverage > Open Filter Viewer**.

If you are inside a subsystem, you can view any coverage filter rule attached to the subsystem. To open the Coverage Filter Viewer, right-click any object inside the subsystem and select **Coverage > Show filter parent**.

Coverage Filter Viewer

In the Coverage Filter Viewer, you can:

- Review and manage the coverage filter rules for your Simulink model.
- Load or save coverage filter files for your model.
- Navigate to the model to create additional coverage filter rules.



To	Action
Navigate to a model object associated with a rule.	<ol style="list-style-type: none"> 1 Select the rule. 2 Click View in model.
Delete a rule.	<ol style="list-style-type: none"> 1 Select the rule. 2 Click Remove rule.

To	Action
Save the current rules to a file.	<ol style="list-style-type: none"><li data-bbox="793 298 1337 333">1 Click Save filter.<li data-bbox="793 333 1337 406">2 Specify a file name and folder for the filter file and click Save.
Load an existing coverage filter file.	<ol style="list-style-type: none"><li data-bbox="793 420 1337 454">1 Click Load filter.<li data-bbox="793 454 1337 527">2 Navigate to the filter file and click Open.
Update the current coverage report with the current filtering rules.	<ol style="list-style-type: none"><li data-bbox="793 541 1337 614">1 Apply or Revert any changes you have made.<li data-bbox="793 614 1337 649">2 Click Generate Report.

Automating Model Coverage Tasks

- “Commands for Automating Model Coverage Tasks” on page 23-2
- “Create Tests with cvtest” on page 23-3
- “Run Tests with cvsim” on page 23-5
- “Retrieve Coverage Details from Results” on page 23-7
- “Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs” on page 23-8
- “Create HTML Reports with cvhtml” on page 23-11
- “Save Test Runs to File with cvsave” on page 23-12
- “Load Stored Coverage Test Results with cvload” on page 23-13
- “Use Coverage Commands in a Script” on page 23-14

Commands for Automating Model Coverage Tasks

Using model coverage commands lets you automate the entire model coverage process with MATLAB scripts. You can use model coverage commands to set up model coverage tests, execute them in simulation, and store and report the results.

Create Tests with `cvtest`

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

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

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

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

To create a test object with a specified label (used for reporting results):

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

To create a test with a setup command:

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

You execute the setup command in the base MATLAB workspace, just prior to running the instrumented simulation. Use this command for loading data prior to a test.

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

Field	Description
<code>id</code>	Read-only internal data-dictionary ID
<code>modelcov</code>	Read-only internal data-dictionary ID
<code>rootPath</code>	Name of the system or subsystem for analysis
<code>label</code>	String for reporting results
<code>setupCmd</code>	Command executed prior to simulation
<code>settings.condition</code>	Set to 1 for condition coverage
<code>settings.decision</code>	Set to 1 for decision coverage
<code>settings.designverifier</code>	Set to 1 for coverage for Simulink Design Verifier blocks.
<code>settings.mcdc</code>	Set to 1 for MCDC coverage

Field	Description
<code>settings.overflowsaturation</code>	Set to 1 for saturate on integer overflow coverage
<code>settings.sigrange</code>	Set to 1 for signal range coverage
<code>settings.sigsize</code>	Set to 1 for signal size coverage.
<code>settings.tableExec</code>	Set to 1 for lookup table coverage
<code>modelRefSettings.enable</code>	String specifying one of the following values: <ul style="list-style-type: none"> • Off — Disables coverage for all referenced models • all — Enables coverage for all referenced models • filtered — Enables coverage for only referenced models not listed in the <code>excludedModels</code> subfield
<code>modelRefSettings.excludeTopModel</code>	Set to 1 for excluding coverage for the top model
<code>modelRefSettings.excludedModels</code>	String specifying a comma-separated list of referenced models for which coverage is disabled when <code>modelRefSettings.enable</code> specifies filtered
<code>emlSettings.enableExternal</code>	Set to 1 to enable coverage for external program files called by MATLAB functions in your model
<code>sfcnSettings.enableSfcn</code>	Set to 1 to enable coverage for C/C++ S-Function blocks in your model.
<code>options.forceBlockReduction</code>	Set to 1 to override the Simulink Block reduction parameter if it is enabled.

Run Tests with `cvsim`

Use the `cvsim` command to simulate a test specification object.

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

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

This command executes the `cvtest` object `cvto` by simulating the corresponding model. `cvsim` returns the coverage results in the `cvdata` object `cvdo`. When recording coverage for multiple models in a hierarchy, `cvsim` returns its results in a `cv.cvdatagroup` object.

You can also control the simulation in a `cvsim` command by setting model parameters for the Simulink `sim` command to apply during simulation:

- The following command executes the test object `cvto` and simulates the model using the default model parameters. The `cvsim` function returns the coverage results in the `cvdata` object `cvdo` and returns the simulation outputs in a Simulink `SimulationOutput` class object `simOut`:

```
[cvdo,simOut] = cvsim(cvto)
```

- The following commands create a structure, `paramStruct`, that specifies the model parameters to use during the simulation. The first command specifies that the simulation collect decision, condition, and MCDC coverage for this model.

```
paramStruct.CovMetricSettings = 'dcm';  
paramStruct.SimulationMode    = 'rapid';  
paramStruct.AbsTol            = '1e-5';  
paramStruct.SaveState         = 'on';  
paramStruct.StateSaveName     = 'xoutNew';  
paramStruct.SaveOutput        = 'on';  
paramStruct.OutputSaveName    = 'youtNew';
```

Note: For a complete list of model parameters, see “Model Parameters” in the Simulink documentation.

The following `cvsim` command executes the test object `cvto` and simulates the model using the model parameter values specified in `paramStruct`:

```
[cvdo,simOut] = cvsim(cvto,paramStruct);
```

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

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

Retrieve Coverage Details from Results

Simulink Verification and Validation provides commands that allow you to retrieve specific coverage information from the `cvtest` object after you have simulated your model and recorded coverage. Use these commands to retrieve the specified coverage information for a block, subsystem, or Stateflow chart in your model or for the model itself:

- `complexityinfo` — Cyclomatic complexity coverage
- `conditioninfo` — Condition coverage
- `decisioninfo` — Decision coverage
- `mcdcinfo` — Modified condition/decision (MCD) coverage
- `overflowsaturationinfo` — Saturate on integer overflow coverage
- `relationalboundaryinfo` — Relational boundary coverage
- `sigrangeinfo` — Signal range coverage
- `sigsizeinfo` — Signal size coverage
- `tableinfo` — Lookup Table block coverage
- `getCoverageinfo` — Coverage for Simulink Design Verifier blocks

The basic syntax of these functions is:

```
coverage = <coverage_type_prefix>info(cvdata_object, ...  
    object, ignore_descendants)
```

- `coverage` — Multipart vector containing the retrieved coverage results for `object`
- `cvdata_object` — `cvdata` object that you create when you call `cvsim`
- `object` — Handle to a model or object in the model
- `ignore_descendants` — Flag to ignore coverage results in subsystems, referenced models, and Stateflow charts

Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs

This example shows how to create and view cumulative coverage results for a model with a reusable subsystem.

Simulink® Verification and Validation™ provides cumulative coverage for multiple instances of identically configured:

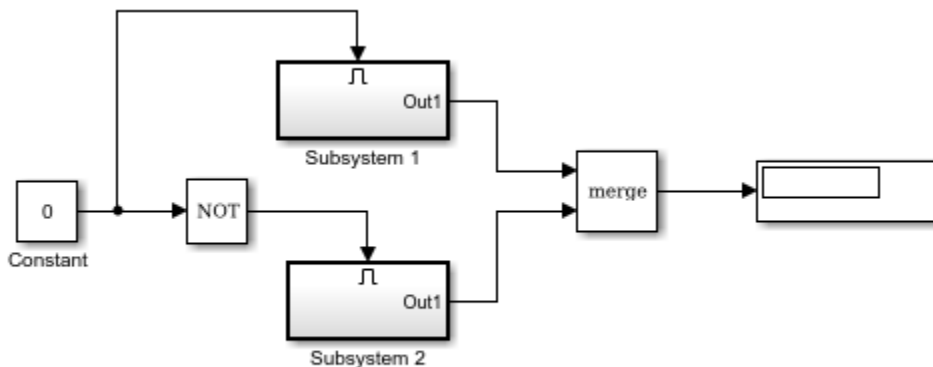
- Reusable subsystems
- Stateflow™ constructs

To obtain cumulative coverage, you add the individual coverage results at the command line. You can get cumulative coverage results for multiple instances across models and test harnesses by adding the individual coverage results.

Open example model

At the MATLAB® command line, type:

```
model = 'slvndemo_cv_mutual_exclusion';  
open_system(model);
```



Copyright 1990-2006 The MathWorks Inc.

This model has two instances of a reusable subsystem. The instances are named Subsystem 1 and Subsystem 2.

Get decision coverage for Subsystem 1

Execute the commands for Subsystem 1 decision coverage:

```
testobj1 = cvtest([model '/Subsystem 1']);  
testobj1.settings.decision = 1;  
covobj1 = cvsim(testobj1);
```

Get decision coverage for Subsystem 2

Execute the commands for Subsystem 2 decision coverage:

```
testobj2 = cvtest([model '/Subsystem 2']);  
testobj2.settings.decision = 1;  
covobj2 = cvsim(testobj2);
```

Add coverage results for Subsystem 1 and Subsystem 2

Execute the command to create cumulative decision coverage for Subsystem 1 and Subsystem 2:

```
covobj3 = covobj1 + covobj2;
```

Generate coverage report for Subsystem 1

Create an HTML report for Subsystem 1 decision coverage:

```
cvhtml('subsystem1',covobj1)
```

The report indicates that decision coverage is 50% for Subsystem 1. The true condition for enable logical value is not analyzed.

Generate coverage report for Subsystem 2

Create an HTML report for Subsystem 2 decision coverage:

```
cvhtml('subsystem2',covobj2)
```

The report indicates that decision coverage is 50% for Subsystem 2. The false condition for enable logical value is not analyzed.

Generate coverage report for cumulative coverage of Subsystem 1 and Subsystem 2

Create an HTML report for cumulative decision coverage for Subsystem 1 and Subsystem 2:

```
cvhtml('cum_subsystem',covobj3)
```

Cumulative decision coverage for reusable subsystems Subsystem 1 and Subsystem 2 is 100%. Both the `true` and `false` conditions for `enable_logical_value` are analyzed.

Create HTML Reports with `cvhtml`

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

The following command creates an HTML report of the coverage results in the `cvdata` object `cvdo`. The results are written to the file `file` in the current MATLAB folder.

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

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

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

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

Save Test Runs to File with `cvsave`

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

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

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

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

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

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

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

Load Stored Coverage Test Results with `cvload`

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

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

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

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

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

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

`cvload` Special Considerations

When using the `cvload` command, be aware of the following considerations:

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

Use Coverage Commands in a Script

The following script demonstrates some common model coverage commands.

This script:

- Creates two data files to load before simulation.
- Creates two `cvtest` objects, `testObj1` and `testObj2`, and simulates them using the default model parameters. Each `cvtest` object uses the `setupCmd` property to load a data file before simulation.
- Enables decision, condition, and MCDC coverage.
- Retrieves the decision coverage results for the Adjustable Rate Limited subsystem.
- Uses `cvhtml` to display the coverage results for the two tests and the cumulative coverage.
- Compute cumulative coverage with the `+` operator and save the results

```
mdl = 'slvndemo_ratelim_harness';
mdl_subsys = 'slvndemo_ratelim_harness/Adjustable Rate Limiter';

open_system(mdl);
open_system(mdl_subsys);

t_gain = (0:0.02:2.0)'; u_gain = sin(2*pi*t_gain);
t_pos = [0;2]; u_pos = [1;1]; t_neg = [0;2]; u_neg = [-1;-1];
save('within_lim.mat', 't_gain', 'u_gain', 't_pos', 'u_pos', ...
    't_neg', 'u_neg');
t_gain = [0;2]; u_gain = [0;4]; t_pos = [0;1;1;2];
u_pos = [1;1;5;5]*0.02; t_neg = [0;2]; u_neg = [0;0];
save('rising_gain.mat', 't_gain', 'u_gain', 't_pos', 'u_pos', ...
    't_neg', 'u_neg');

testObj1 = cvtest(mdl_subsys);
testObj1.label = 'Gain within slew limits';
testObj1.setupCmd = 'load(''within_lim.mat'')';
testObj1.settings.mcdc = 1;
testObj1.settings.condition = 1;
testObj1.settings.decision = 1;

testObj2 = cvtest(mdl_subsys);
testObj2.label = 'Rising gain that temporarily exceeds slew limit';
testObj2.setupCmd = 'load(''rising_gain.mat'')';
testObj2.settings.mcdc = 1;
testObj2.settings.condition = 1;
testObj2.settings.decision = 1;

[dataObj1, simOut1] = cvsim(testObj1);
decision_cov1 = decisioninfo(dataObj1, mdl_subsys);
percent_cov1 = 100 * decision_cov1(1) / decision_cov1(2)
cc_cov2 = complexityinfo(dataObj1, mdl_subsys);

[dataObj2, simOut2] = cvsim(testObj2, [0 2]);
```

```
decision_cov2 = decisioninfo(dataObj2,mdl_subsys);  
percent_cov2 = 100 * decision_cov2(1) / decision_cov2(2)  
cc_cov2 = complexityinfo(dataObj1, mdl_subsys);
```

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

```
cvsave('ratelim_testdata',cumulative);
```

```
close_system('slvndemo_ratelim_harness',0);
```


Checking Systems

Checking Systems Interactively

Check for Compliance in Model and Subsystems

You can use the Model Advisor to check a model or subsystem for adherence to modeling guidelines or standards. The Model Advisor includes checks that help you define and implement consistent design guidelines. Using model checks, you can apply guidelines across projects and development teams.

You can use the Model Advisor to check your model in these ways:

- Run checks interactively after you complete your model design.
- Configure the Model Advisor to check for violations while you edit.

The Model Advisor reviews your model for conditions and configuration settings that cause inaccurate or inefficient simulation and code generation of the system that the model represents.

Check Your Model Interactively

You can use the Model Advisor to check your model interactively against modeling standards and guidelines. In the model window, select **Analysis > Model Advisor > Model Advisor**. Select the model or system that you want to review. Select the checks that you want to run on your model from the **By Product** or **By Task** folders. Then run your selected checks. The Model Advisor reviews your model and, if selected, displays an HTML report of your results.

Depending on which products you have installed, the Model Advisor includes different checks.

For more information	See
Checking model compliance with the DO-178C safety standard	“Model Checks for DO-178C/DO-331 Standard Compliance” on page 24-31
Checking model compliance with the IEC 61508, IEC 62304, ISO 26262, or EN 50182 safety standards	“Model Checks for IEC 61508, IEC 62304, ISO 26262, and EN 50128 Standard Compliance” on page 24-36
Checking model compliance with MathWorks Automotive Advisory Board (MAAB) guidelines	“Model Checks for MathWorks Automotive Advisory Board (MAAB) Guideline Compliance” on page 24-39

For more information	See
Checking model compliance with the MISRA C:2012 standard	“Model Checks for MISRA C:2012 Compliance” on page 24-45
Checking requirements links	“Model Checks for Requirements Links” on page 24-46
Checking model metrics	“Model Metrics” on page 26-2

Check Your Model While You Edit

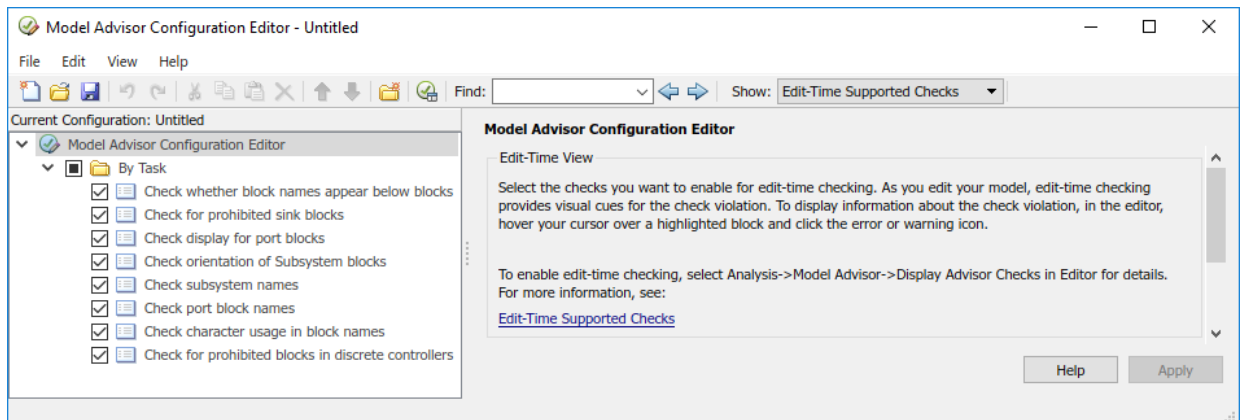
You can identify standards compliance issues earlier in the model design process using edit-time checking. Edit-time checking provides visual cues for some Model Advisor check violations. In the model editor, highlighted blocks alert you to issues as you design your model. Hover your cursor over a highlighted block for information about the violation.

Configure Your Model for Edit-Time Checking

To enable edit-time checking for Model Advisor checks, in the model window, select **Analysis > Model Advisor > Display Advisor Checks in Editor**.

Once you select this option, the Model Advisor provides visual cues for several standards compliance issues. To configure the selection and behavior of these checks:

- 1 Open a filtered view of the edit-time checks in the Model Advisor Configuration Editor. In the model window, select **Analysis > Model Advisor > Configure Advisor Edit-Time Checks**.



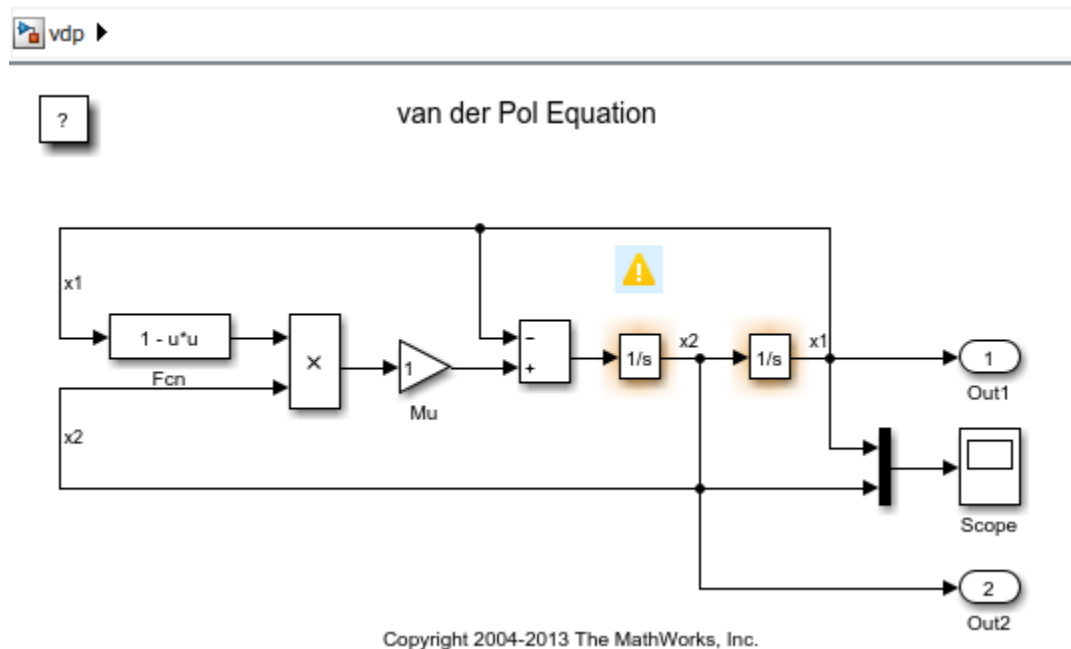
- 2 Use the configuration editor to enable, disable, or customize checks.
- 3 If you have made updates to check selection or behavior, save the current configuration. Then select **File > Set Current Configuration as Default**.

Note: Only the default configuration can change the behavior of edit-time checks.

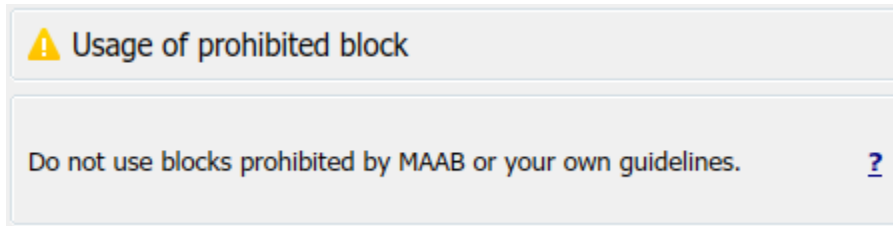
To customize the behavior of edit-time checks, configure updates in the filtered view of edit-time checks in the Model Advisor Configuration Editor. If a check appears in multiple folders of your Model Advisor tree, for edit-time checking, Model Advisor gives priority to the check in your custom folder. If the check is not in your custom folder, priority goes to the check in the **By Task** folder, and finally to the check in your **By Product** folder.

Look for Visual Cues in the Model Editor

Once you have configured edit-time checking, as you edit your model, highlighted blocks alert you to compliance issues. Hover your cursor over a highlighted block and click the error or warning icon.



A dialog box provides a description of the warning. Click the question mark for detailed documentation on the check that detected the issue.



Note: Edit-time checking does not support Model Advisor exclusions.

Software is inherently complex and may not be completely free of errors. Model Advisor checks might contain bugs. MathWorks reports known bugs brought to its attention on its Bug Report system at <http://www.mathworks.com/support/bugreports/>. The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

While applying Model Advisor checks to your model will increase the likelihood that your model does not violate certain modeling standards or guidelines, it is ultimately your responsibility to verify, using multiple methods, that the system being developed provides its intended functionality and does not include any unintended functionality.

Related Examples

- “Select Checks and Run Model Advisor”

More About

- “Consulting the Model Advisor”

Transform Model to Variant System

You can use the Model Transformer tool to improve model componentization by replacing qualifying modeling patterns with Variant Source and Variant Subsystem blocks. The Model Transformer reports the qualifying modeling patterns. You choose which modeling patterns the tool replaces with a Variant Source block or Variant Subsystem block.

The Model Transformer can perform these transformations:

- If an If block connects to one or more If Action Subsystems and each one has one output, replace this modeling pattern with a subsystem and a Variant Source block.
- If an If block connects to an If Action Subsystem that does not have an output or has two or more outputs, replace this modeling pattern with a Variant Subsystem block.
- If a Switch Case block connects to one or more Switch Case Action Subsystems and each one has one output, replace this modeling pattern with a subsystem and a Variant Source block.
- If a Switch Case block connects to a Switch Case Action Subsystem that does not have an output or has two or more outputs, replace this modeling pattern with a Variant Subsystem block.
- Replace a Switch block with a Variant Source block.
- Replace a Multiport Switch block that has two or more data ports with a Variant Source block.

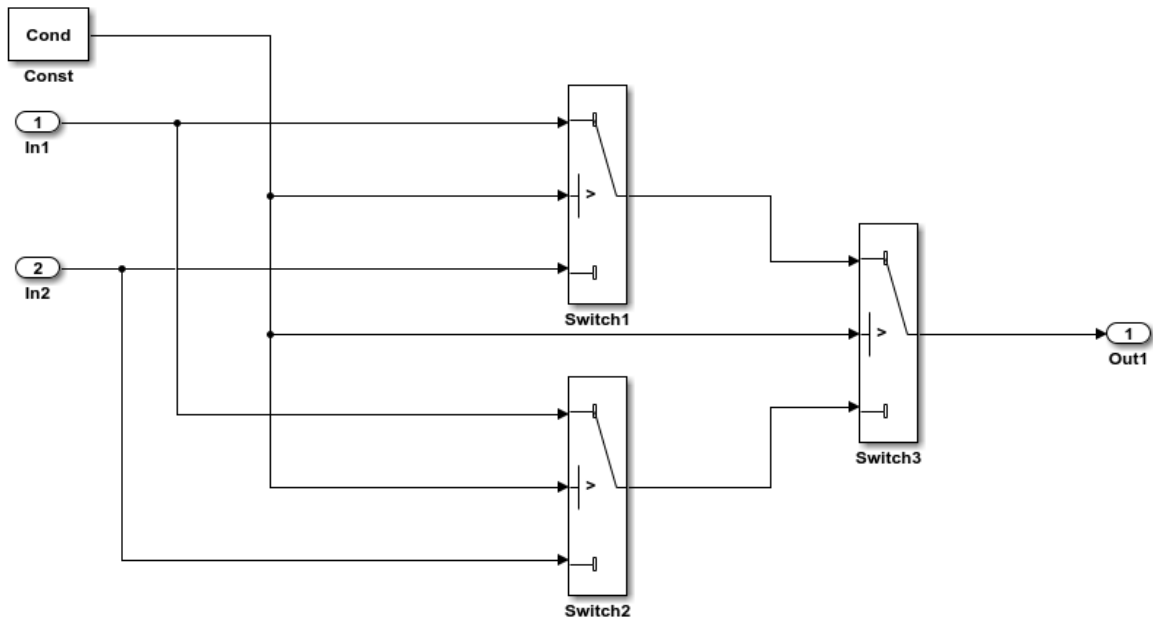
For the Model Transformer tool to perform the transformation, the control input to Multiport Switch or Switch blocks and the inputs to If or Switch Case blocks must be either of the following:

- A Constant block in which the **Constant value** parameter is a scalar and a `Simulink.Parameter` object with one of these storage classes:
 - `Define` with header file specified
 - `ImportedDefine` with header file specified
 - `CompilerFlag`
 - `SystemConstant` (AUTOSAR)
 - User-defined custom storage class
- Constant blocks in which the **Constant value** parameters are scalars and `Simulink.Parameter` objects with one of the preceding storage classes and some

other combination of blocks that form a supported MATLAB expression. For a list of supported MATLAB expressions, see “Operators and Operands in Variant Condition Expressions”.

Example Model

This example shows how to use the Model Transformer to transform a model into a variant system. The example uses the model `rtwdemo_controlflow_opt`. This model has three Switch blocks. The control input to these Switch blocks is the `Simulink.Parameter` `cond`. The Model Transformer dialog box and this example refer to `cond` as a system constant.

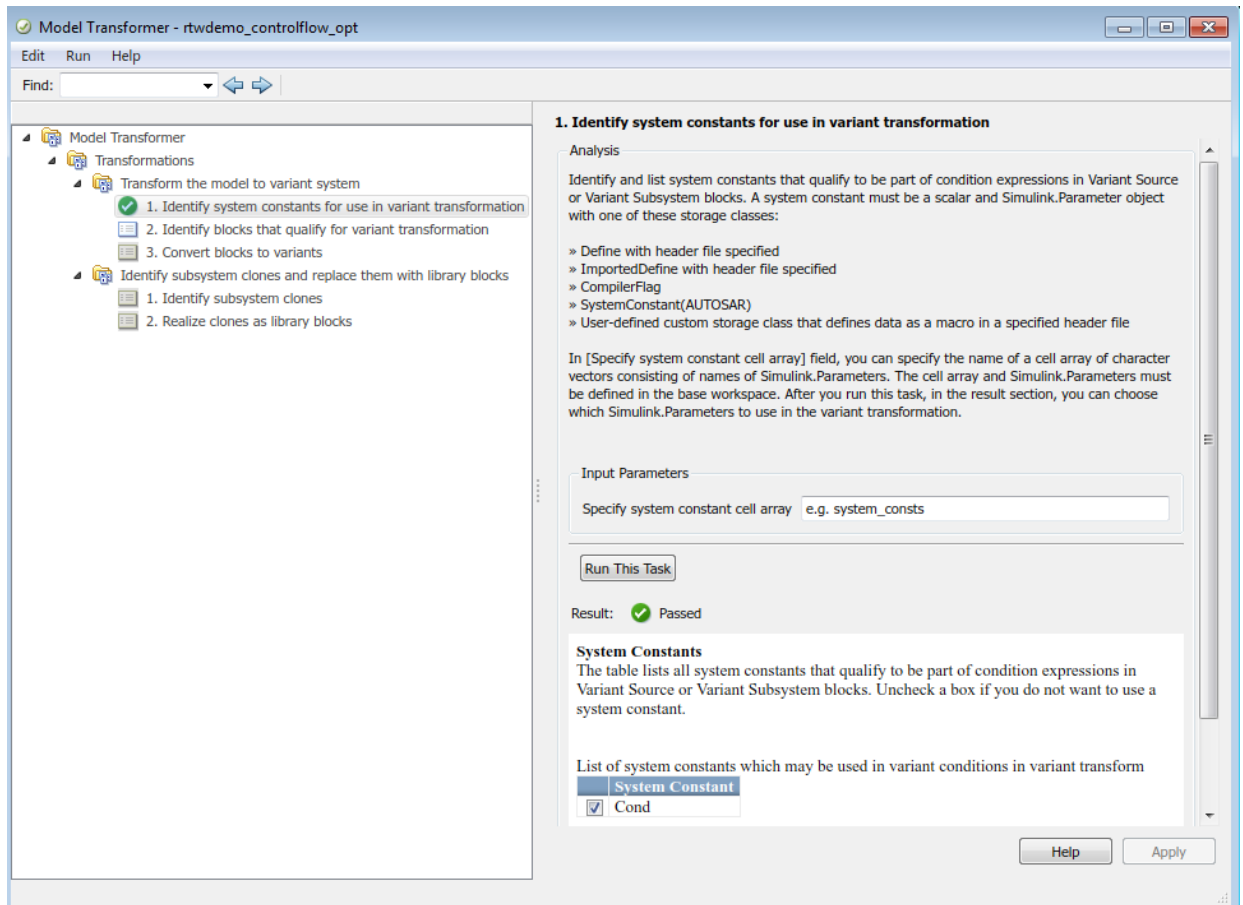


- 1 Open the model. In the Command Window, type `rtwdemo_controlflow_opt`.
- 2 Save the model to your working folder.
- 3 Open the Switch1 Block Parameters dialog box. Change the **Threshold** parameter to 0. The **Threshold** parameter must be an integer because after the variant transformation it is part of the condition expression in the Variant Source block.

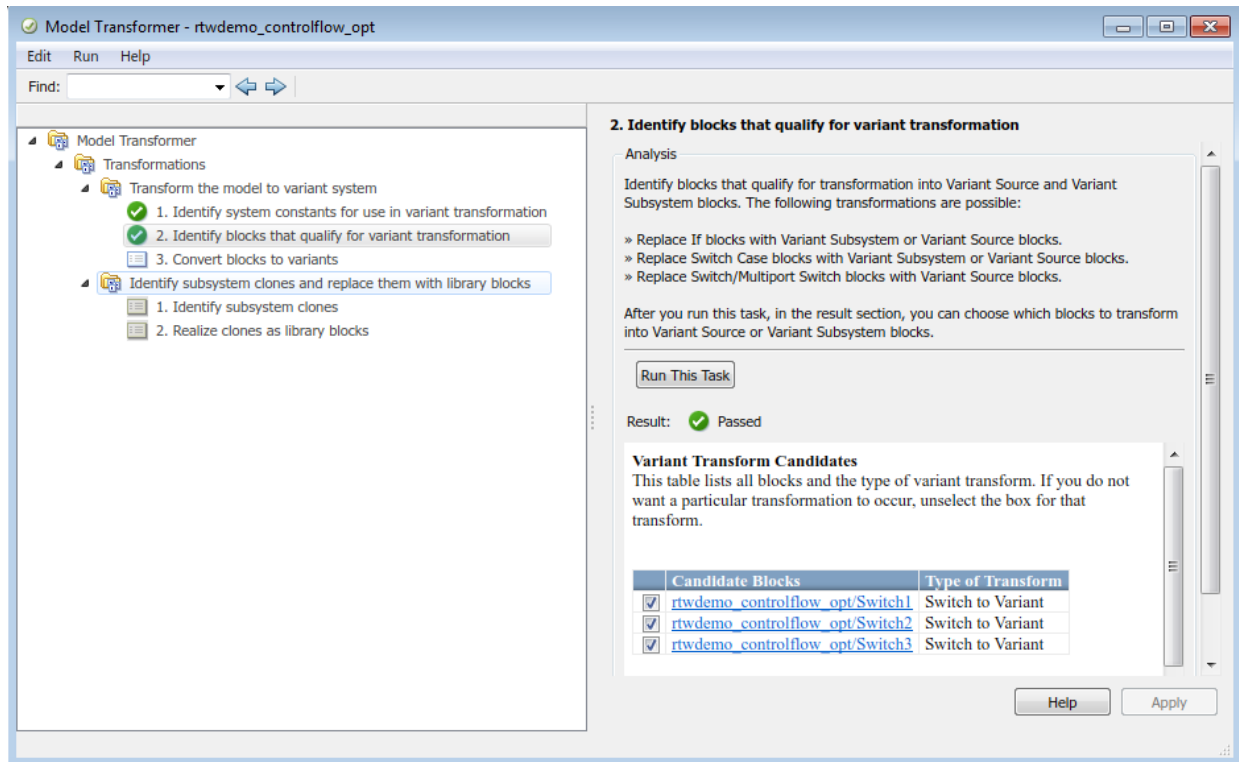
- 4 Repeat step 3 for the Switch blocks `Switch1`, `Switch2`, and `Switch3`.
- 5 Open the Model Explorer. Select the base workspace.
- 6 Select the `Simulink.Parameter`, `Cond`.
- 7 Change the **Storage class** parameter to `CompilerFlag`. Close the base workspace and save the model.

Identify Blocks That Qualify for Variant Transformation

- 1 From the Model Editor, open the Model Transformer by selecting **Analysis > Model Transformer**. Or, in the Command Window, type:
`mdltransformer('rtwdemo_controlflow_opt')`
- 2 Select the folder “Transform the model to variant system”. If you want to perform the three steps of the transformation at once, you can select **Run all**, or proceed one step at a time.
- 3 Select the first step “1. Identify system constants for use in variant transformation”. For the Model Transformer to list a system constant, it must be a `Simulink.Parameter` object with a custom storage class from the list at the beginning of this example.
- 4 Select **Run this Task**. The Model Transformer lists system constants that qualify to be part of condition expressions in Variant Source or Variant Subsystem blocks. For this example, `Cond` qualifies to part of a condition expression.



- 5 Select the second step “2. Identify blocks that qualify for variant transformation”.
- 6 Select **Run this Task**. The Model Transformer lists blocks that qualify for a variant transformation. If you do not want one of the transformations to occur, you can clear the check box next to it.



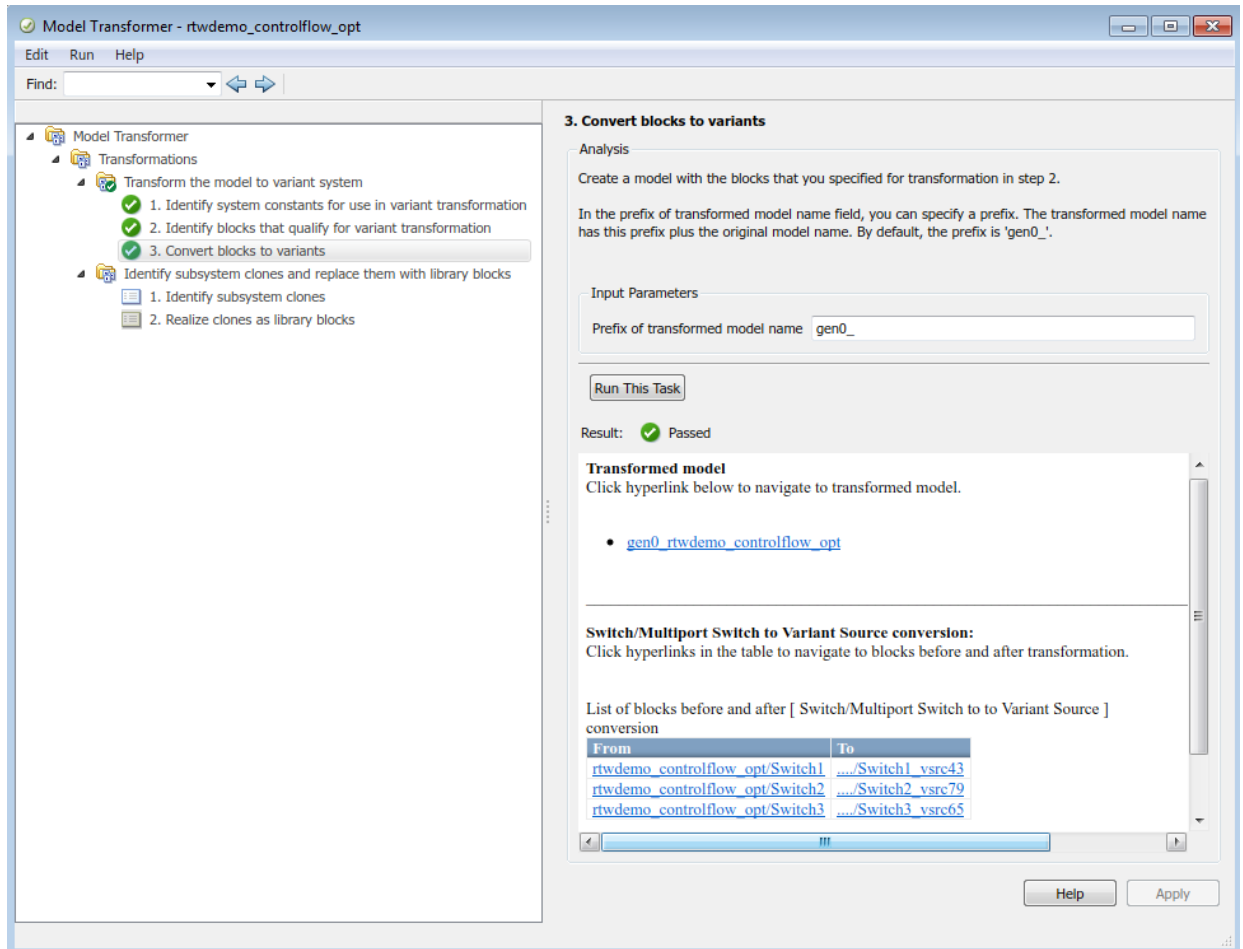
Perform Variant Transformation

- 1 Select the third step “3. Convert blocks to variants”. This step creates a model with the blocks that you specified for transformation in step 2.

In the **Prefix of transformed model name** field, specify a prefix for the model name. If you do not specify a prefix, the default is `gen0`. The transformed model is in the folder `m2m_rtwdemo_controlflow_opt`.

The transformed model or models are in the folder that has the prefix `m2m` plus the original model name. For this example, the folder name is `m2m_rtwdemo_controlflow_opt`.

- 2 Select **Run this Task**. The Model Transformer provides a hyperlink to the transformed model and hyperlinks to the corresponding blocks in the original model and the transformed model.



- 3 In the original model `rtwdemo_controlflow_opt`, right-click one of the Switch blocks. In the menu, select **Model Transformer > Traceability to Transformed Block**. In the transformed model `gen0_rtwdemo_controlflow_opt`, the corresponding Variant Source block is highlighted.

- 4 In the transformed model `gen0_rtwdemo_controlflow_opt`, right-click one of the Switch blocks. In the menu, select **Model Transformer > Traceability to Original Block**. In the original model `rtwdemo_controlflow_opt`, the corresponding Switch block is highlighted.

Variant Transformation Limitations

- If an If Action Subsystem block drives a Merge block, and the Merge block has another inport that is either unconnected or driven by another conditional subsystem, the Model Transformer does not add a Variant Source block. When you perform the step **Convert blocks to variants**, this modeling pattern produces a warning and an excluded candidate message.
- The Model Transformer cannot perform a variant transformation for every model patterns and settings. This list contains some exceptions:
 - The model contains a protected model reference block.
 - A model contains a Variant Source block with the **Analyze all choices during update diagram and generate preprocessor conditionals** parameter set to off.

Model Transformer Limitations

The Model Transformer tool has these limitations:

- If you want to run both the transformations, you must run **Transform the model to variant system** before **Identify subsystem clones and replace them with library blocks**.
- After you run one or more tasks, you cannot rerun tasks because the **Run this Task** and **Run All** buttons are deactivated. If you want to rerun a task, reset the Model Transformer by right-clicking **Model Transformer** and selecting **Reset**.
- Do not change a model in the middle of a transformation. If you want to change the model, close the **Model Transformer**, modify the model, and then reopen the **Model Transformer**.
- For the hyperlinks in the Model Transformer to work, you must have the model to which the links point to open.
- If you run both transformations, the Model Transformer creates one model. The name of this model begins with the prefixes from both transformations plus the original model name.

Related Examples

- “Variant Systems”

Enable Component Reuse with Clone Detection

You can use the Model Transformer tool to improve model componentization by creating library blocks from subsystem clones, and then replacing the clones with links to those library blocks. Creating library blocks enables component reuse.

The Model Transformer tool reports two or more subsystems as clones if they have identical structure and parameter settings. Two clones do not have to be completely identical. Clones can have the following differences:

- For a parameter setting, one clone can have a symbol while the other clone has a numeric value, as long as the symbol has the same numeric value.
- Two clones can have a different sorted order.
- The length of signal lines and the location and size of blocks can be different as long as the block connections are the same.
- Blocks can have different names.

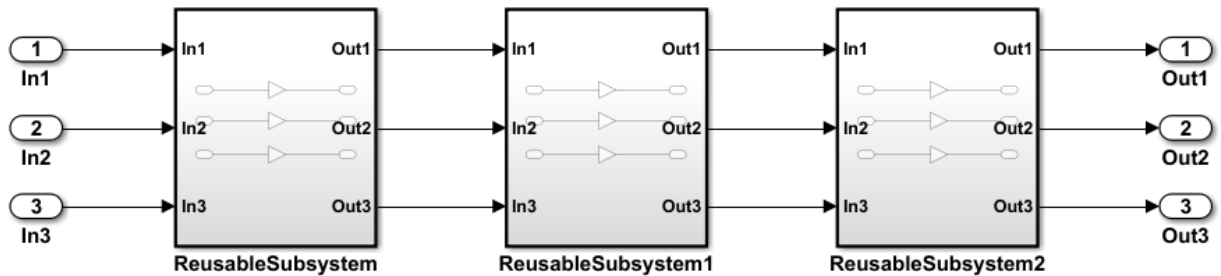
If one clone has a link to a library block, the Model Transformer reports a missing link for the other subsystem or subsystems. The Model Transformer also reports clones that do not have links to library blocks. You choose whether to create a library block and replace a clone with a link to that block.

The Model Transformer detects clones across referenced model boundaries.

Identify Subsystem Clones

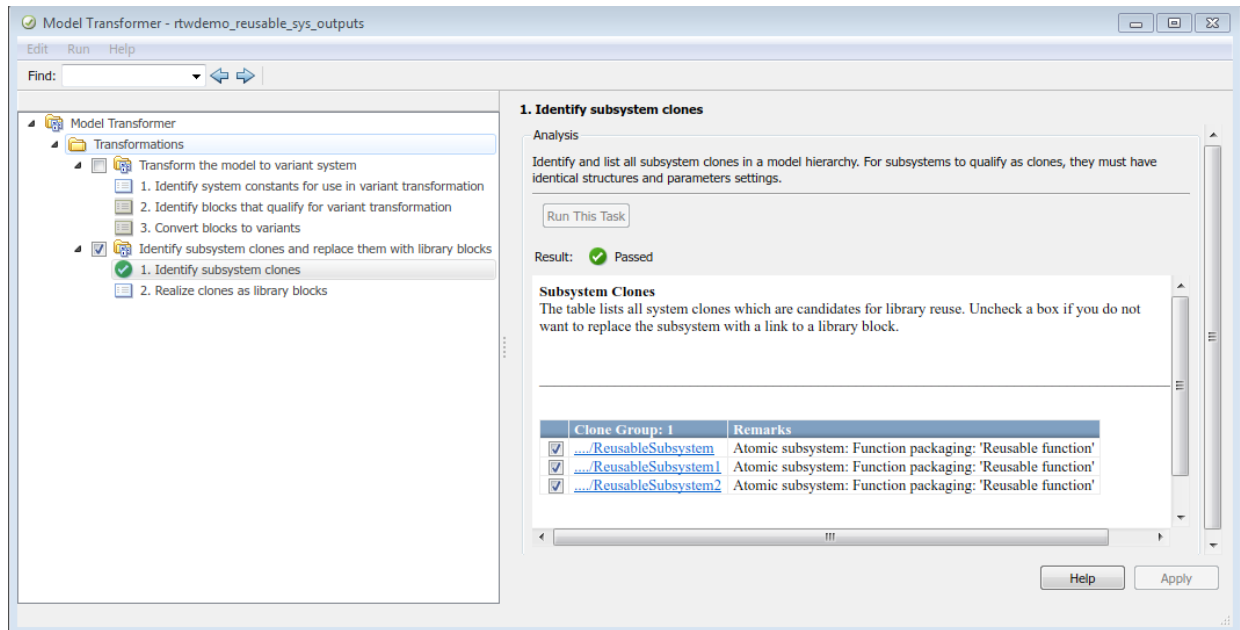
This example shows how to use the Model Transformer tool to identify clones and replace them with links to library blocks. This example uses the model `rtwdemo_reusable_sys_outputs`.

- 1 Open the model. In the Command Window, type `rtwdemo_reusable_sys_outputs`.
- 2 Save the model to your current working folder.
- 3 For `rtwdemo_reusable_sys_outputs`, create clones by copying and pasting the subsystem `ReusableSubsystem` two times between the original `ReusableSubsystem` and the three output ports. Save the model.



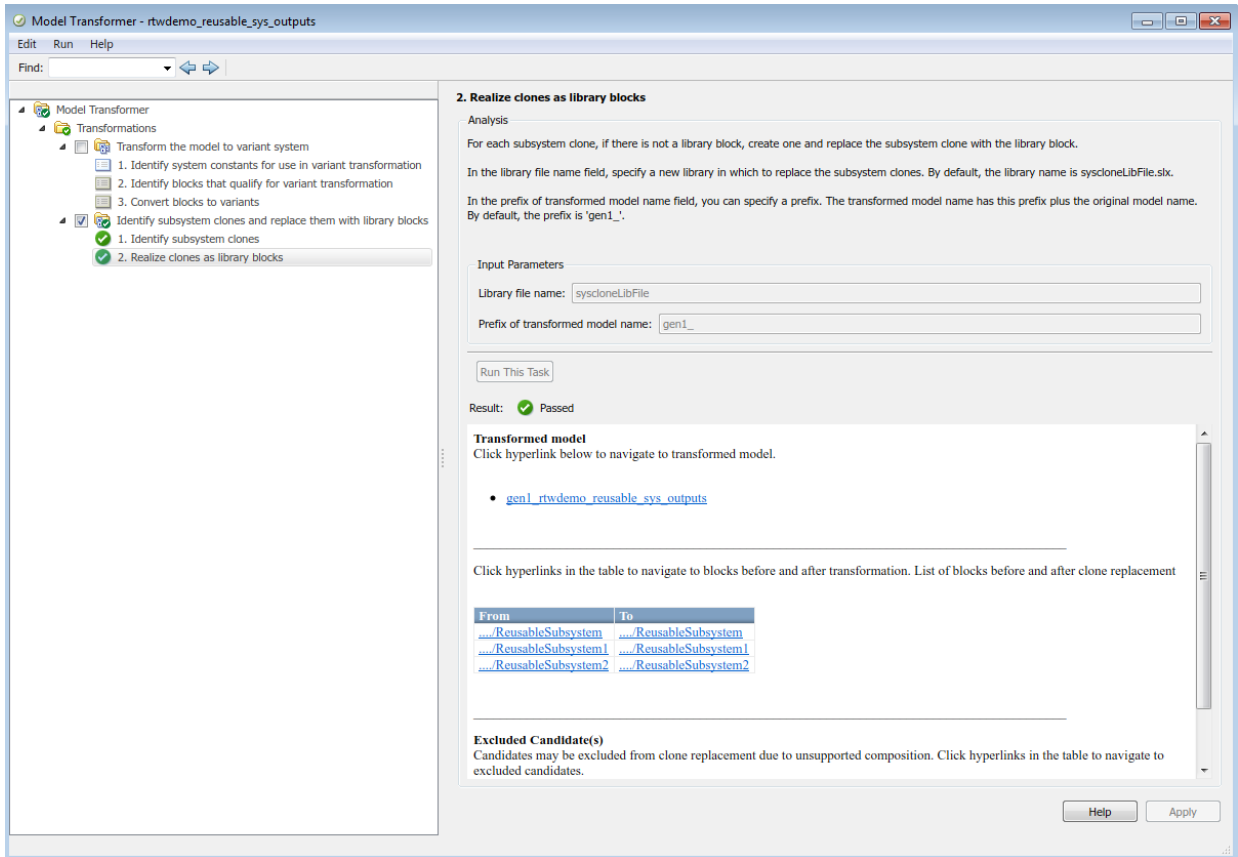
- 4 From the Model Editor, open the Model Transformer by selecting **Analysis > Model Transformer**. Or, in the Command Window, type: `mdltransformer('rtwdemo_controlflow_opt')`.
- 5 Clear the check box next to **Transform the model to variant system**. You must run the variant transform steps first.
- 6 Select the folder “Identify subsystem clones and replace them with library blocks”. If you want to perform both steps of the transformation at once, select **Run all**. Or, you can perform each step individually.
- 7 Select the first step “1. Identify subsystem clones”.
- 8 Select **Run This Task**. The Model Transformer lists subsystem clones. This model does not contain missing links from clones to library blocks. If a model is missing these links, the Model Transformer lists these broken links.

If you do not want to link from a clone to a library block, clear the check box next to the clone.



Replace Subsystem Clones with Links to Library Blocks

- 1 Select the second step “2. Realize clones as library blocks”.
- 2 For the **Library file name** parameter, specify a name for the library in which you want to place the subsystem clones. If you do not specify a name, the default value is `syscloneLibFile`.
- 3 For the **Prefix of transformed model name** parameter, specify a prefix for the name of the model that contains the links from the subsystem clones to the library blocks. If you do not specify a prefix, the default is `gen1_`.
- 4 Select **Run This Task**. The Model Transformer lists links to the transformed model and links to the corresponding blocks in the original model and the transformed model. The library containing the subsystem `ReusableSubsystem` is in the current working folder. The transformed model or models are in the folder that has the prefix `m2m` plus the original model name. For this example, the folder name is `m2m_rtwdemo_reusable_sys_outputs`.



- 5 In rtwdemo_reusable_sys_outputs, right-click one of the three subsystems. From the menu, select **Model Transformer > Traceability to transformed block**. In the transformed model gen1_rtwdemo_reusable_sys_outputs, the corresponding linked library block is highlighted.
- 6 In gen1_rtwdemo_reusable_sys_outputs, right-click one of the three clones. From the menu, select **Model Transformer > Traceability to original block**. In the original model rtwdemo_reusable_sys_outputs, the corresponding linked library block is highlighted.

Model Transformer Limitations

The Model Transformer tool has these limitations:

- If you want to run both the transformations, you must run **Transform the model to variant system** before **Identify subsystem clones and replace them with library blocks**.
- After you run one or more tasks, you cannot rerun tasks because the **Run this Task** and **Run All** buttons are deactivated. If you want to rerun a task, reset the Model Transformer by right-clicking **Model Transformer** and selecting **Reset**.
- Do not change a model in the middle of a transformation. If you want to change the model, close the **Model Transformer**, modify the model, and then reopen the **Model Transformer**.
- For the hyperlinks in the Model Transformer to work, you must have the model to which the links point to open.
- If you run both transformations, the Model Transformer creates one model. The name of this model begins with the prefixes from both transformations plus the original model name.

Related Examples

- “Libraries”
- “Reusable Library Subsystem”

Limit Model Checks

What Is a Model Advisor Exclusion?

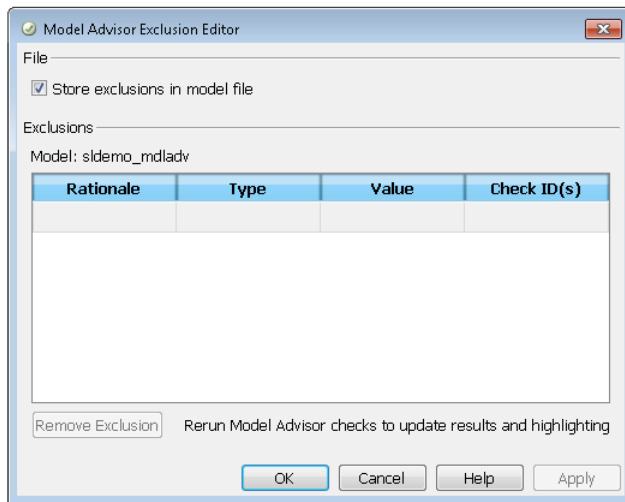
To save time during model development and verification, you can limit the scope of a Model Advisor analysis of your model. You can create a *Model Advisor exclusion* to exclude blocks in the model from selected checks. You can exclude all or selected checks from:

- Simulink blocks
- Stateflow charts

After you specify the blocks to exclude, Model Advisor uses the exclusion information to exclude blocks from specified checks during analysis. By default, Model Advisor exclusion information is stored in the model SLX file. Alternately, you can store the information in an exclusion file.

Note: Edit-time checking does not support Model Advisor exclusions.

To view exclusion information for the model, right-click in the model window or right-click a block and select **Model Advisor > Open Model Advisor Exclusion Editor**. The Model Advisor Exclusion Editor dialog box includes the following information for each exclusion.



Field	Description
Rationale	A description of why this object is excluded from Model Advisor checks. The rationale field is the only field that you can edit.
Type	Whether a specific block is excluded or all blocks of a given type are excluded.
Value	Name of excluded block or blocks.
Check ID (s)	Names of checks for which the block exclusion applies.

Note: If you comment out blocks, they are excluded from both simulation and Model Advisor analysis.

Save Model Advisor Exclusions in a Model File

To save Model Advisor exclusions to the model SLX file, in the Model Advisor Exclusion Editor dialog box, select **Store exclusions in model file**. When you open the model SLX file, the model contains the exclusions.

Save Model Advisor Exclusions in Exclusion File

A *Model Advisor exclusion file* specifies the collection of blocks to exclude from specified checks in an exclusion file. You can create exclusions and save them in an exclusion file. To use an exclusion file, in the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**. The **Exclusion File** field is enabled.

The **Exclusion File** contains the exclusion file name and location associated with the model. You can use an exclusion file with several models. However, a model can have only one exclusion file.

Unless you specify a different folder, the Model Advisor saves exclusion files in the current folder. The default name for an exclusion file is `<model_name>_exclusions.xml`.

If you create an exclusion file and save your model, you attach the exclusion file to your model. Each time that you open the model, the blocks and checks specified in the exclusion file are excluded from the analysis.

Create Model Advisor Exclusions

- 1 In the model window, right-click a block and select **Model Advisor**. Select the menu option for the type of exclusion that you want to do.

To	Select Model Advisor >
Exclude the block from all checks.	Exclude block only > All Checks
Exclude all blocks of this type from all checks.	Exclude all blocks with type <block_type> > All Checks
Exclude the block from selected checks.	<ul style="list-style-type: none"> • Exclude block only > Select Checks. • In the Check Selector dialog box, select the checks. Click OK.
Exclude all blocks of this type from selected checks.	<ul style="list-style-type: none"> • Exclude all blocks with type <block_type> > Select Checks. • In the Check Selector dialog box, select the checks. Click OK.
Exclude the block from all failed checks. After a	Exclude block only > Only failed checks

To	Select Model Advisor >
Model Advisor analysis, this option is available.	
Exclude all blocks of this type from all failed checks. After a Model Advisor analysis, this option is available.	Exclude all blocks with type <block_type> > Only failed checks
Exclude the block from a failed check. After a Model Advisor analysis, this option is available.	Exclude block only > <name of failed check>
Exclude all blocks of this type from a failed check. After a Model Advisor analysis, this option is available.	Exclude all blocks with type <block_type> > <name of failed check>


- 2 In the Model Advisor Exclusion Editor dialog box, to:
 - Store exclusions in model file, select **Store exclusions in model file**. Click **OK** or **Apply** to create the exclusion.
 - Save the information to an exclusion file, clear **Store exclusions in model file**. Click **OK** or **Apply**. If this exclusion is the first one, a Save Exclusion File as dialog box opens. In this dialog box, click **Save** to create a exclusion file with the default name *<model_name>_exclusions.xml* in the current folder. Optionally, you can select a different file name or location.
- 3 Optionally, if you want to change the exclusion file name or location:
 - a In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**.
 - b In the Model Advisor Exclusion Editor dialog box, select **Change**.
 - c In the Change Exclusion File dialog box, select **Save as**.
 - d In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.
 - e In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information to an exclusion file.

You can create as many Model Advisor exclusions as you want by right-clicking model blocks and selecting **Model Advisor**. Each time that you create an exclusion, the Model Advisor Exclusion Editor dialog box opens. In the **Rationale** field, you can specify a reason for excluding blocks or checks from the Model Advisor analysis. The rationale is useful to others who review your model.

If you create an exclusion file and save your model, you attach the exclusion file to your model. Each time that you open the model, the blocks and checks specified in the exclusion file are excluded from the analysis.

Review Model Advisor Exclusions

You can review the exclusions associated with your model. Before or after a Model Advisor analysis, to view exclusions information:

- Right-click in the model window or right-click a block and select **Model Advisor > Open Model Advisor Exclusion Editor**. The Model Advisor Exclusion Editor dialog box lists the exclusions for your model.
- On the Model Advisor toolbar, select **Settings > Preferences**. In the Model Advisor Preferences dialog box, select **Show Exclusion tab**. In the right pane of the Model Advisor window, select the **Exclusions** tab to display checks that are excluded from the Model Advisor analysis.
- In the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
 - 1 On the Model Advisor window toolbar, select **Highlighting > Highlight Exclusions**. By default, this menu option is selected.
 - 2 In the Model Advisor window, click **Enable highlighting** .
 - 3 In the left pane of the Model Advisor window, select a check. The blocks excluded from the check appear in the model window, highlighted in gray with a black border.

After the Model Advisor analysis, you can view exclusion information for individual checks in the:

- HTML report. Before the analysis, in the Model Advisor window, make sure that you select the **Show report after run** check box.
- Model Advisor window. In the left pane of the Model Advisor window, select **By Product > Simulink > < name of check >**. If the **By Product** folder is not

displayed, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

If the check	The HTML report and Model Advisor window
Has no exclusions rules applied.	Show that no exclusions were applied to this check.
Does not support exclusions.	Shows that the check does not support exclusions.
Is excluded from a block.	Lists the check exclusion rules.

Manage Exclusions

Save Exclusions in a File

- 1 In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file** and click **OK** or **Apply**. If this exclusion is the first one, a Save Exclusion File as dialog box opens. In this dialog box, click **Save** to create an exclusion file with the default name `<model_name>_exclusions.xml` in the current folder. Optionally, you can select a different file name or location.
- 2 If you want to change the exclusion file name or location:
 - a In the Model Advisor Exclusion Editor dialog box, select **Change**.
 - b In the Change Exclusion File dialog box, select **Save as**.
 - c In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.
 - d In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information in an exclusion file.

Load an Exclusion File

To load an existing exclusion file for use with your model:

- 1 In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**. Click **Change**.
- 2 In the Change Exclusion File dialog box, click **Load**.
- 3 Navigate to the exclusion file that you want to use with your model. Select **Open**.
- 4 In the Model Advisor Exclusion Editor dialog box, click **OK** to associate the exclusion file with your model.

Detach an Exclusion File

To detach an exclusion file associated with your model:

- 1 In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**. Click **Change**.
- 2 In the Change Exclusion File dialog box, click **Detach**.
- 3 In the Model Advisor Exclusion Editor dialog box, click **OK**.

Remove an Exclusion

- 1 In the Model Advisor Exclusion Editor dialog box, select the exclusions that you want to remove.
- 2 Click **Remove Exclusion**.

Add a Rationale to an Exclusion

You can add text that describes why you excluded a particular block or blocks from selected checks during Model Advisor analysis. A description is useful to others who review your model.

- 1 In the Model Advisor Exclusion Editor dialog box, double-click the **Rationale** field for the exclusion.
- 2 Delete the existing text.
- 3 Add the rationale for excluding this object.

Programmatically Specify an Exclusion File

You can use the `MAModelExclusionFile` method to programmatically specify the name of an exclusion file.

- 1 Use `get_param` to obtain the model object. For example, for `sldemo_mdldadv`:

```
mo = get_param('sldemo_mdldadv','Object')
```
- 2 Use `MAModelExclusionFile` to specify the name of an exclusion file. For example, to specify exclusion file `my_exclusion.xml` in `S:\work`:

```
mo.MAModelExclusionFile = ['S:\work\','my_exclusion.xml']
```
- 3 Open the Model Advisor Exclusion Editor dialog box. The **Exclusion File** field contains the specified exclusion file and path.

Related Examples


- “Limit Model Checks By Excluding Gain and Outport Blocks” on page 24-27
- “Exclude Blocks From Custom Checks” on page 28-54

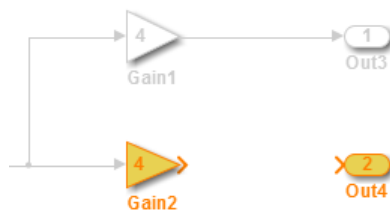
More About

- “Consulting the Model Advisor”

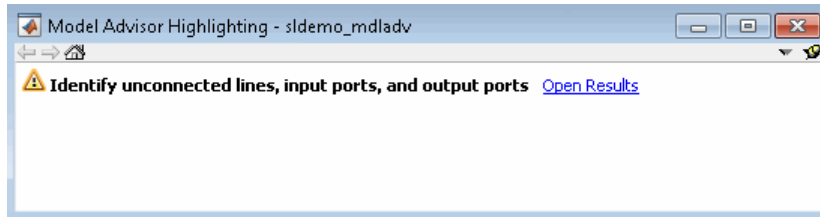
Limit Model Checks By Excluding Gain and Output Blocks

This example shows how to exclude a Gain block and all Output blocks from a Model Advisor check during a Model Advisor analysis. By excluding individual blocks from checks, you limit the scope of the analysis and might save time during model development and verification.

- 1 At the MATLAB command line, type `sldemo_mdadv`.
- 2 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 3 A System Selector — Model Advisor dialog box opens. Click **OK**.
- 4 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 5 In the left pane of the Model Advisor window, expand **By Product > Simulink**. Select the **Show report after run** check box to see an HTML report of check results after you run the checks.
- 6 Run the selected checks by clicking the **Run selected checks** button. After the Model Advisor runs the checks, an HTML report displays the check results in a browser window. The check **Identify unconnected lines, input ports, and output ports** triggers a warning.
- 7 In the left pane of the Model Advisor window, select the check **By Product > Simulink > Identify unconnected lines, input ports, and output ports**.
- 8 In the Model Advisor window, click the **Enable highlighting** button .
 - The model window opens. The blocks causing the **Identify unconnected lines, input ports, and output ports** check warning are highlighted in yellow.

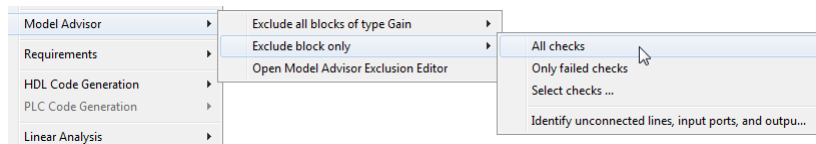


- The Model Advisor Highlighting window opens with a link to the Model Advisor window. In the Model Advisor window, you can find more information about the check results and how to fix the warning condition.

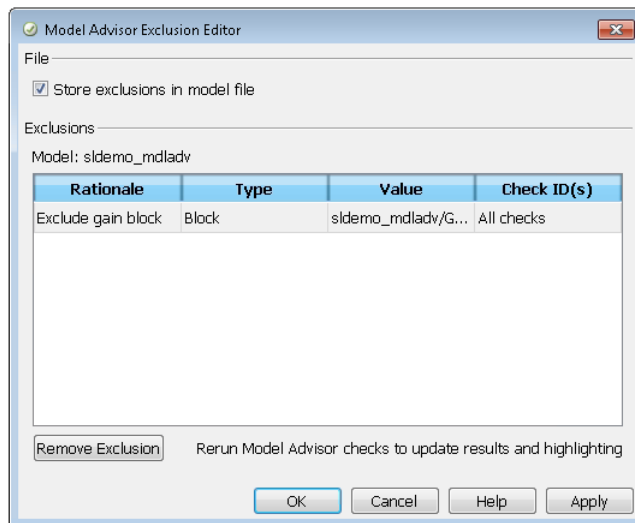


9 After reviewing the check results, exclude the Gain2 block from all Model Advisor checks:

a In the model window, right-click the Gain2 block and select **Model Advisor > Exclude block only > All Checks** .



b In the Model Advisor Exclusion Editor dialog box, double-click in the first row of the **Rationale** field, and enter **Exclude gain block**.



c Click **OK** to store the exclusion in the model file.

- 10** After reviewing the check results, exclude all Outport blocks from the Identify unconnected lines, input ports, and output ports check:
- a** Right-click the Out4 block and select **Model Advisor > Exclude all blocks of type Output > Identify unconnected lines, input ports, and output ports**.
 - b** In the Model Advisor Exclusion Editor dialog box, click **OK** to store the exclusion in the model file.
- 11** In the left pane of the Model Advisor window, select **By Product > Simulink** and then:
- Select the **Show report after run** check box.
 - Click **Run Selected Checks** to run a Model Advisor analysis.
- 12** After the Model Advisor completes the analysis, you can view exclusion information for the Identify unconnected lines, input ports, and output ports check in the:
- HTML report:

✔ **Identify unconnected lines, input ports, and output ports**

Identify unconnected lines, input ports, and output ports in the model

Passed

There are no unconnected lines, input ports, and output ports in this model.

Check Exclusions Rules

Rationale	Exclusion Usage Count
Exclusion for all blocks of type Output	1
Exclude gain block	1


- **Model Advisor window.** In the left pane of the Model Advisor window, select **By Product > Simulink > Identify unconnected lines, input ports, and output ports**.

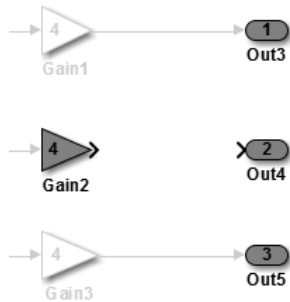
Identify unconnected lines, input ports, and output ports in the model

Passed
There are no unconnected lines, input ports, and output ports in this model.

Check Exclusions Rules

Rationale	Exclusion Usage Count
Exclusion for all blocks of type Output	1
Exclude gain block	1

- Model window. In the left pane of the Model Advisor window, select **By Product** > **Simulink** > **Identify unconnected lines, input ports, and output ports**. Then click the **Enable highlighting** button ().



13 Close `sldemo_mdadv`.

Related Examples

- “Exclude Blocks From Custom Checks” on page 28-54
- “Select Checks and Run Model Advisor”

More About

- “Limit Model Checks” on page 24-19
- “Consulting the Model Advisor”

Model Checks for DO-178C/DO-331 Standard Compliance

You can check that your model or subsystem complies with selected aspects of the DO-178C safety standard by running the Model Advisor. Navigate to **By Product > Simulink Verification and Validation > Modeling DO-178C/DO-331 Checks** and run the checks.

For information on the DO-178C Software Considerations in Airborne Systems and Equipment Certification and related standards, see Radio Technical Commission for Aeronautics (RTCA) .

The table lists the DO-178C/DO-331 checks, with applicable “High-Integrity System Modeling” guidelines.

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
“Check safety-related optimization settings”	<ul style="list-style-type: none"> • “hisl_0018: Usage of Logical Operator block” • “hisl_0045: Configuration Parameters > Optimization > Implement logic signals as Boolean data (vs. double)” • “hisl_0046: Configuration Parameters > Optimization > Block reduction” • “hisl_0048: Configuration Parameters > Optimization > Application lifespan (days)” • “hisl_0052: Configuration Parameters > Optimization > Data initialization” • “hisl_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values” • “hisl_0054: Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions”
“Check safety-related diagnostic settings for solvers”	“hisl_0043: Configuration Parameters > Diagnostics > Solver”

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
“Check safety-related diagnostic settings for sample time”	“hisl_0044: Configuration Parameters > Diagnostics > Sample Time”
“Check safety-related diagnostic settings for signal data”	“hisl_0005: Usage of Product blocks”
“Check safety-related diagnostic settings for parameters”	“hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters”
“Check safety-related diagnostic settings for data used for debugging”	“hisl_0305: Configuration Parameters > Diagnostics > Debugging”
“Check safety-related diagnostic settings for data store memory”	“hisl_0013: Usage of data store blocks”
“Check safety-related diagnostic settings for type conversions”	“hisl_0309: Configuration Parameters > Diagnostics > Type Conversion”
“Check safety-related diagnostic settings for signal connectivity”	“hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals”
“Check safety-related diagnostic settings for bus connectivity”	“hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses”
“Check safety-related diagnostic settings that apply to function-call connectivity”	“hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls”
“Check safety-related diagnostic settings for compatibility”	“hisl_0301: Configuration Parameters > Diagnostics > Compatibility”
“Check safety-related diagnostic settings for model initialization”	“hisl_0304: Configuration Parameters > Diagnostics > Model initialization”
“Check safety-related diagnostic settings for model referencing”	“hisl_0310: Configuration Parameters > Diagnostics > Model Referencing”
“Check safety-related model referencing settings”	“hisl_0037: Configuration Parameters > Model Referencing”

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
“Check safety-related code generation settings”	<ul style="list-style-type: none"> • “hisl_0038: Configuration Parameters > Code Generation > Comments” • “hisl_0039: Configuration Parameters > Code Generation > Interface” • “hisl_0047: Configuration Parameters > Code Generation > Code Style” • “hisl_0049: Configuration Parameters > Code Generation > Symbols”
“Check safety-related diagnostic settings for saving”	“hisl_0036: Configuration Parameters > Diagnostics > Saving”
“Check for blocks that do not link to requirements”	
“Check state machine type of Stateflow charts”	“hisf_0001: Mealy and Moore semantics”
“Check Stateflow charts for ordering of states and transitions”	“hisf_0002: User-specified state/transition execution order”
“Check Stateflow debugging options”	“hisf_0011: Stateflow debugging settings”
“Check usage of lookup table blocks”	“hisl_0033: Usage of Lookup Table blocks”
“Check for MATLAB Function interfaces with inherited properties”	“himl_0002: Strong data typing at MATLAB function boundaries”
“Check MATLAB Function metrics”	“himl_0003: Limitation of MATLAB function complexity”
“Check MATLAB Code Analyzer messages”	“himl_0004: MATLAB Code Analyzer recommendations for code generation”
“Check MATLAB code for global variables”	“himl_0005: Usage of global variables in MATLAB functions”
“Check for inconsistent vector indexing methods”	“hisl_0021: Consistent vector indexing method”
“Check for blocks not recommended for C/C++ production code deployment”	“hisl_0020: Blocks not recommended for MISRA C:2012 compliance”

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
“Check for variant blocks with 'Generate preprocessor conditionals' active”	“hisl_0023: Verification of model and subsystem variants”
“Check Stateflow charts for uniquely defined data objects”	“hisl_0061: Unique identifiers for clarity”
“Check usage of Math Operations blocks”	<ul style="list-style-type: none"> • “hisl_0001: Usage of Abs block” • “hisl_0002: Usage of Math Function blocks (rem and reciprocal)” • “hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)” • “hisl_0029: Usage of Assignment blocks”
“Check usage of Signal Routing blocks”	“hisl_0034: Usage of Signal Routing blocks”
“Check usage of Logic and Bit Operations blocks”	<ul style="list-style-type: none"> • “hisl_0016: Usage of blocks that compute relational operators” • “hisl_0017: Usage of blocks that compute relational operators (2)” • “hisl_0018: Usage of Logical Operator block”
“Check usage of Ports and Subsystems blocks”	<ul style="list-style-type: none"> • “hisl_0006: Usage of While Iterator blocks” • “hisl_0007: Usage of While Iterator subsystems” • “hisl_0008: Usage of For Iterator Blocks” • “hisl_0009: Usage of For Iterator Subsystem blocks” • “hisl_0010: Usage of If blocks and If Action Subsystem blocks” • “hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks”
“Check model object names”	“hisl_0032: Model object names”

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
“Display model version information”	

Related Examples

- “Select Checks and Run Model Advisor”

More About

- “Consulting the Model Advisor”

Model Checks for IEC 61508, IEC 62304, ISO 26262, and EN 50128 Standard Compliance

You can check that your model or subsystem complies with selected aspects of the following safety standards by running the Model Advisor:

- IEC 61508-3 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements
- IEC 62304 Medical device software - Software life cycle processes
- ISO 26262-6 Road vehicles - Functional safety - Part 6: Product development: Software level
- EN 50128 Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems

To check compliance, run the checks in the applicable Model Advisor folders.

- **By Task > Modeling Standards for IEC 61508**
- **By Task > Modeling Standards for IEC 62304**
- **By Task > Modeling Standards for ISO 26262**
- **By Task > Modeling Standards for EN 50128**

The table lists the IEC 61508, IEC 62304, ISO 26262, and EN 50128 checks, with applicable “High-Integrity System Modeling” guidelines.

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
“Display configuration management data”	
“Check model object names”	“hisl_0032: Model object names”
“Display model metrics and complexity report”	
“Check for unconnected objects”	
“Check for root Inports with missing properties”	“hisl_0024: Inport interface definition”
“Check for root Inports with missing range definitions”	“hisl_0025: Design min/max specification of input interfaces”

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
“Check for root Outports with missing range definitions”	“hisl_0026: Design min/max specification of output interfaces”
“Check for blocks not recommended for C/C++ production code deployment”	
“Check usage of Stateflow constructs”	<ul style="list-style-type: none"> • “hisf_0002: User-specified state/transition execution order” • “hisf_0009: Strong data typing (Simulink and Stateflow boundary)” • “hisf_0011: Stateflow debugging settings” • “hisl_0061: Unique identifiers for clarity”
“Check state machine type of Stateflow charts”	“hisf_0001: Mealy and Moore semantics”
“Check usage of Math Operations blocks”	<ul style="list-style-type: none"> • “hisl_0001: Usage of Abs block” • “hisl_0029: Usage of Assignment blocks”
“Check usage of Signal Routing blocks”	
“Check usage of Logic and Bit Operations blocks”	<ul style="list-style-type: none"> • “hisl_0016: Usage of blocks that compute relational operators” • “hisl_0017: Usage of blocks that compute relational operators (2)” • “hisl_0018: Usage of Logical Operator block”

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
“Check usage of Ports and Subsystems blocks”	<ul style="list-style-type: none"> • “hisl_0006: Usage of While Iterator blocks” • “hisl_0007: Usage of While Iterator subsystems” • “hisl_0008: Usage of For Iterator Blocks” • “hisl_0009: Usage of For Iterator Subsystem blocks” • “hisl_0010: Usage of If blocks and If Action Subsystem blocks” • “hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks”
“Check for inconsistent vector indexing methods”	“hisl_0021: Consistent vector indexing method”
“Check for model objects that do not link to requirements”	
“Check for MATLAB Function interfaces with inherited properties”	“himl_0002: Strong data typing at MATLAB function boundaries”
“Check MATLAB Function metrics”	“himl_0003: Limitation of MATLAB function complexity”
“Check MATLAB Code Analyzer messages”	“himl_0004: MATLAB Code Analyzer recommendations for code generation”
“Check MATLAB code for global variables”	“himl_0005: Usage of global variables in MATLAB functions”

Related Examples

- “Select Checks and Run Model Advisor”

More About

- “Consulting the Model Advisor”

Model Checks for MathWorks Automotive Advisory Board (MAAB) Guideline Compliance

You can check that your model or subsystem complies with MathWorks Automotive Advisory Board (MAAB) Guidelines by running the Model Advisor. Navigate to **By Task > Modeling Standards for MAAB** and run the checks.

The MAAB involves major automotive OEMs and suppliers in the process of evolving MathWorks controls, simulation, and code generation products, including Simulink, Stateflow, and Simulink Coder. An important result of the MAAB has been the MAAB Control Algorithm Modeling Guidelines.

The table lists the MAAB checks, with the applicable MAAB Control Algorithm Modeling Guideline. If the check has a ✓ in the JMAAB column, you can run the check to verify compliance with a Japan MBD Automotive Advisory Board (JMAAB) Control Algorithm Modeling Guideline.

By Task > Modeling Standards for MAAB subfolder	MathWorks Automotive Advisory Board (MAAB) Check	MAAB Control Algorithm Modeling Guidelines, Version 3.0 in the Simulink documentation	JMAAB Control Algorithm Modeling Guidelines, Version 4.0. To download the Guidelines, see the JMAAB website.
Naming Conventions	“Check file names”	ar_0001: Filenames	✓
	“Check folder names”	ar_0002: Directory names	✓
	“Check subsystem names”	jc_0201: Usable characters for Subsystem names	✓
	“Check port block names”	jc_0211: Usable characters for Inport blocks and Outport blocks	✓
	“Check character usage in signal labels”	jc_0221: Usable characters for signal line names	✓

By Task > Modeling Standards for MAAB subfolder	MathWorks Automotive Advisory Board (MAAB) Check	MAAB Control Algorithm Modeling Guidelines, Version 3.0 in the Simulink documentation	JMAAB Control Algorithm Modeling Guidelines, Version 4.0. To download the Guidelines, see the JMAAB website.
	“Check character usage in block names”	jc_0231: Usable characters for block names	✓
Model Architect	“Check for mixing basic blocks and subsystems”	db_0143: Similar block types on the model levels	✓
Model Configuration Options	“Check Implement logic signals as Boolean data (vs. double)”	jc_0011: Optimization parameters for Boolean data types	✓
	“Check model diagnostic parameters”	jc_0021: Model diagnostic settings	No applicable guideline
Simulink	“Check for Simulink diagrams using nonstandard display attributes”	na_0004: Simulink model appearance	✓
	“Check font formatting”	db_0043: Simulink font and font size	✓
	“Check positioning and configuration of ports”	db_0042: Port block in Simulink models	✓
	“Check visibility of block port names”	na_0005: Port block name visibility in Simulink models	
	“Check display for port blocks”	jc_0081: Icon display for Port block	
	“Check whether block names appear below blocks”	db_0142: Position of block names	✓
	“Check the display attributes of block names”	jc_0061: Display of block names	✓

By Task > Modeling Standards for MAAB subfolder	MathWorks Automotive Advisory Board (MAAB) Check	MAAB Control Algorithm Modeling Guidelines, Version 3.0 in the Simulink documentation	JMAAB Control Algorithm Modeling Guidelines, Version 4.0. To download the Guidelines, see the JMAAB website.
	“Check position of Trigger and Enable blocks”	db_0146: Triggered, enabled, conditional Subsystems	✓
	“Check for nondefault block attributes”	db_0140: Display of basic block parameters	✓
	“Check for matching port and signal names”	jm_0010: Port block names in Simulink models	
	“Check Trigger and Enable block names”	jc_0281: Naming of Trigger Port block and Enable Port block	✓
	“Check signal line labels”	na_0008: Display of labels on signals	✓
	“Check for propagated signal labels”	na_0009: Entry versus propagation of signal labels	✓
	“Check for unconnected ports and signal lines”	db_0081: Unconnected signals, block inputs and block outputs	✓
	“Check for prohibited blocks in discrete controllers”	jm_0001: Prohibited Simulink standard blocks inside controllers	✓
	“Check for blocks not recommended for C/ C++ production code deployment”		
	“Check for prohibited sink blocks”	hd_0001: Prohibited Simulink sinks	✓
	“Check scope of From and Goto blocks”	na_0011: Scope of Goto and From blocks	

By Task > Modeling Standards for MAAB subfolder	MathWorks Automotive Advisory Board (MAAB) Check	MAAB Control Algorithm Modeling Guidelines, Version 3.0 in the Simulink documentation	JMAAB Control Algorithm Modeling Guidelines, Version 4.0. To download the Guidelines, see the JMAAB website.
	“Check usage of Switch blocks”	jc_0141: Use of the Switch block	✓
	“Check usage of Relational Operator blocks”	jc_0131: Use of Relational Operator block	✓
	“Check for indexing in blocks”	db_0112: Indexing	✓
	“Check usage of buses and Mux blocks”	na_0010: Grouping data flows into signals	
	“Check usage of tunable parameters in blocks”	db_0110: Tunable parameters in basic blocks	✓
	“Check orientation of Subsystem blocks”	jc_0111: Direction of Subsystem	✓
Stateflow	“Check usage of exclusive and default states in state machines”	db_0137: States in state machines	
	“Check transition orientations in flow charts”	db_0132: Transitions in flow charts	✓
	“Check entry formatting in State blocks in Stateflow charts”	jc_0501: Format of entries in a State block	
	“Check return value assignments of graphical functions in Stateflow charts”	jc_0511: Setting the return value from a graphical function	✓

By Task > Modeling Standards for MAAB subfolder	MathWorks Automotive Advisory Board (MAAB) Check	MAAB Control Algorithm Modeling Guidelines, Version 3.0 in the Simulink documentation	JMAAB Control Algorithm Modeling Guidelines, Version 4.0. To download the Guidelines, see the JMAAB website.
	“Check default transition placement in Stateflow charts”	jc_0531: Placement of the default transition	✓
	“Check for Strong Data Typing with Simulink I/O”	db_0122: Stateflow and Simulink interface signals and parameters	✓
	“Check Stateflow data objects with local scope”	db_0125: Scope of internal signals and local auxiliary variables	✓
	“Check usage of return values from a graphical function in Stateflow charts”	jc_0521: Use of the return value from graphical functions	✓
	“Check for MATLAB expressions in Stateflow charts”	db_0127: MATLAB commands in Stateflow	✓
	“Check for pointers in Stateflow charts”	jm_0011: Pointers in Stateflow	✓
	“Check for event broadcasts in Stateflow charts”	jm_0012: Event broadcasts	✓
	“Check transition actions in Stateflow charts”	db_0151: State machine patterns for transition actions	✓
	“Check for bitwise operations in Stateflow charts”	na_0001: Bitwise Stateflow operators	✓
	“Check for unary minus operations on unsigned integers in Stateflow charts”	jc_0451: Use of unary minus on unsigned integers in Stateflow	✓

By Task > Modeling Standards for MAAB subfolder	MathWorks Automotive Advisory Board (MAAB) Check	MAAB Control Algorithm Modeling Guidelines, Version 3.0 in the Simulink documentation	JMAAB Control Algorithm Modeling Guidelines, Version 4.0. To download the Guidelines, see the JMAAB website.
	“Check for comparison operations in Stateflow charts”	na_0013: Comparison operation in Stateflow	✓
	“Check for equality operations between floating-point expressions in Stateflow charts”	jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow	
	“Check for mismatches between names of Stateflow ports and associated signals”	db_0123: Stateflow port names	✓
MATLAB Functions and Code	“Check input and output settings of MATLAB Functions”	na_0034: MATLAB Function block input/output settings	✓
	“Check MATLAB Function metrics”	<ul style="list-style-type: none"> • na_0016: Source lines of MATLAB Functions • na_0018: Number of nested if/else and case statement 	✓
	“Check MATLAB code for global variables”	na_0024: Global Variables	✓

Related Examples

- “Select Checks and Run Model Advisor”

More About

- “Consulting the Model Advisor”

Model Checks for MISRA C:2012 Compliance

You can check that your model or subsystem has a likelihood of generating MISRA C:2012 compliant code. Navigate to **By Task > Modeling Guidelines for MISRA C:2012** and run the checks:

- “Check usage of Assignment blocks”
- “Check for blocks not recommended for MISRA C:2012”
- “Check for unsupported block names”
- “Check configuration parameters for MISRA C:2012”
- “Check for equality and inequality operations on floating-point values”
- “Check for bitwise operations on signed integers”
- “Check for recursive function calls”
- “Check for switch case expressions without a default case”

Related Examples

- “Select Checks and Run Model Advisor”

More About

- “Consulting the Model Advisor”

Model Checks for Requirements Links

To check that every requirements link in your model has a valid target in a requirements document, navigate to **Analysis > Requirements Traceability > Check Consistency** and run the Model Advisor checks:

- “Identify requirement links with missing documents”
- “Identify requirement links that specify invalid locations within documents”
- “Identify selection-based links having descriptions that do not match their requirements document text”
- “Identify requirement links with path type inconsistent with preferences”

Related Examples

- “Validate Requirements Links in a Model” on page 6-4
- “Select Checks and Run Model Advisor”

More About

- “Consulting the Model Advisor”

Generate Model Advisor Reports in Adobe PDF and Microsoft Word Formats

By default, when the Model Advisor runs checks, it generates an HTML report of check results in the `slprj/modeladvisor/model_name` folder. On Windows platforms, you can generate Model Advisor reports in Adobe PDF and Microsoft Word .docx formats.

The beginning of the PDF and Microsoft Word versions of the Model Advisor reports contain the:

- Model name
- Simulink version
- System
- Treat as Referenced Model
- Model version
- Current run

To generate a Model Advisor report in Adobe PDF or Microsoft Word:

- 1 In the Model Advisor window, navigate to the folder that contains the checks that you ran.
- 2 Select the folder. The right pane of the Model Advisor window displays information about that folder. The pane includes a **Report** box.
- 3 In the Report box, click **Generate Report**.
- 4 In the Generate Model Advisor Report dialog box, enter the path to the folder where you want to generate the report. Provide a file name.
- 5 In the Generate Model Advisor Report dialog box **File format** field, select PDF or Word.
- 6 Click **OK**. The Model Advisor generates the report in PDF or Microsoft Word format to the location that you specified.

Modify Default Template

If you have a MATLAB Report Generator license, you can modify the default template that the Model Advisor uses to generate the report in PDF or Microsoft Word.

The default template contains holes that the Model Advisor uses to populate the generated report with information about the analysis. If you want your Model Advisor report to contain the analysis information, do not delete the holes. When the Model Advisor uses the template to generate the report, analysis information overrides the text that you enter in the template hole field.

Template Hole	In generated report, displays
ModelName	Model name
SimulinkVersion	Simulink version
SystemName	System name
TreatAsMdlRef	Whether or not model is treated as a referenced model
ModelVersion	Model version
CurrentRun	Model Advisor analysis time stamp
PassCount	Number of checks that pass
FailCount	Number of checks that fail
WarningCount	Number of checks that cause a warning
NrunCount	Number of checks that did not run
TotalCount	Total number of checks
CheckResults	Results for each check

This example shows how to add a header to a PDF version of a Model Advisor report.

- 1 Using Microsoft Word, open the default template `matlabroot/toolbox/simulink/simulink/modeladvisor/resources/templates/default.dotx`.
- 2 Rename and save the template `default.dotx` to a writable location. For example, save template `default.dotx` to `C:/work/ma_format/mytemplate.dotx`.
- 3 In the template `C:/work/ma_format/mytemplate.dotx` file, add a header. For example, in the template header, add the text **My Custom Header**. Save the template as a Microsoft Word `.dotx` file.

My Custom Header

Model Advisor Report – Model name

Simulink version: Simulink version

Model version: Model version

System: System name

Current run: Timestamp

Treat as Referenced Model: If it's treat as referenced model

Run Summary

Pass	Fail	Warning	Not Run	Total
 Passed check	 Failed check	 Warning check	 Not run check	Total number

Results of all checks

- 4 In the Model Advisor window Report pane, click **Generate Report**.
- 5 In the Generate Model Advisor Report dialog box:
 - Enter the path to the folder where you want to generate the report and provide a file name.
 - Set **File format** to PDF.
 - Select **View report after generation**.
 - Set **Report template** to C:\work\ma_format\mytemplate.dotx.
- 6 Click **OK**. The Model Advisor generates the report in PDF format with a custom header. Because the template `mytemplate.dotx` contains holes that Model Advisor uses to populate the generated report, the report contains information about the Model Advisor analysis. For example, the report contains the model name, model version, and number of checks that pass.

My Custom Header

Model Advisor Report – sldemo_mdldv

Simulink version: 8.5

Model version: 1.78

System: sldemo_mdldv

Current run: 13-Mar-2015 10:27:03

Treat as Referenced Model: off

Run Summary

Pass	Fail	Warning	Not Run	Total
 1	 0	 2	 30	33

Related Examples

- “Save and View Model Advisor Reports”
- “Customize Microsoft Word Part Templates”
- “Select Checks and Run Model Advisor”

More About

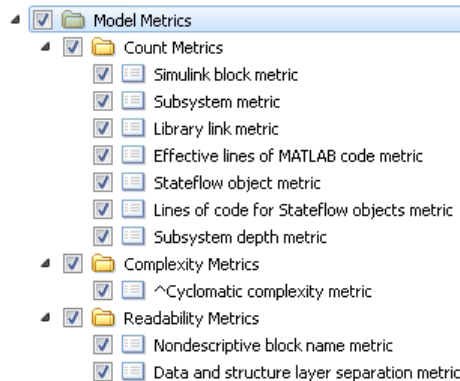
- “Consulting the Model Advisor”

Run Checks for Model Metrics

To help you assess your model for size, complexity, and readability, you can run the model metrics checks in the Model Advisor **By Task > Model Metrics** subfolder.

This example shows how to run model metrics checks on your model.

- 1 Open your model. For example, open the example model `sldemo_fuelsys`.
- 2 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor. A System Selector — Model Advisor dialog box opens. Click **OK**.
- 3 In the left pane of the Model Advisor window, navigate to **By Task > Model Metrics**. Select the model metric checks to run on your model. For example, select all the metrics checks.



- 4 Click **Run selected checks** to run the selected checks.
- 5 After the Model Advisor runs an analysis, in the left pane of the Model Advisor window, select a model metric check to explore the result. The result displays information about your model. Alternatively, you can view the analysis results in the Model Advisor report.

For example, in model `sldemo_fuelsys`:

- Select metric check **Simulink block count metric**. The metric result displays the block count. A summary table provides the number of blocks at the root model level and subsystem level.

Display number of blocks in the model.

Total Blocks: 175

Passed

Component	Blocks
.../fuel_rate_control/airflow_calc	24
sldemo_fuelsys	17
.../Throttle & Manifold/Throttle	14
sldemo_fuelsys/To Controller	12

▼ More (16 rows)

- Select metric check **Cyclomatic complexity metric**. The metric displays the cyclomatic complexity. A summary table provides the cyclomatic complexity for `sldemo_fuelsys` and subsystems in `sldemo_fuelsys`.
- 6 After reviewing the check results, you can update your model to meet size, complexity, and readability recommendations.

Related Examples

- “Select Checks and Run Model Advisor”

More About

- “Model Metrics” on page 26-2
- “Consulting the Model Advisor”

Check Systems Programmatically

Checking Systems Programmatically

The Simulink Verification and Validation product includes a programmable interface for scripting and for command-line interaction with the Model Advisor. Using this interface, you can:

- Create scripts and functions for distribution that check one or more systems using the Model Advisor.
- Run the Model Advisor on multiple systems in parallel on multicore machines (requires a Parallel Computing Toolbox™ license).
- Check one or more systems using the Model Advisor from the command line.
- Archive results for reviewing at a later time.

To define the workflow for running multiple checks on systems:

- 1 Specify a list of checks to run. Do one of the following:
 - Create a Model Advisor configuration file that includes only the checks that you want to run.
 - Create a list of check IDs.
- 2 Specify a list of systems to check.
- 3 Run the Model Advisor checks on the list of systems using the `ModelAdvisor.run` function.
- 4 Archive and review the results of the run.

See Also

`ModelAdvisor.run`

Related Examples

- “Archive and View Results” on page 25-10
- “Find Check IDs” on page 25-3

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 29-5

Find Check IDs

An *ID* is a unique identifier for a Model Advisor check. You find check IDs in the Model Advisor, using check context menus.

To Find	Do This
Check Title, ID, or location of the MATLAB source code	<ol style="list-style-type: none"> 1 On the model window toolbar, select Settings > Preferences. 2 In the Model Advisor Preferences dialog box, select Show Source Tab. 3 In the right pane of the Model Advisor window, click the Source tab. The Model Advisor window displays the check Title, TitleId, and location of the MATLAB source code for the check.
Check ID	<ol style="list-style-type: none"> 1 In the left pane of the Model Advisor, select the check. 2 Right-click the check name and select Send Check ID to Workspace. The ID is displayed in the Command Window and sent to the base workspace.
Check IDs for selected checks in a folder	<ol style="list-style-type: none"> 1 In the left pane of the Model Advisor, select the checks for which you want IDs. Clear the other checks in the folder. 2 Right-click the folder and select Send Check ID to Workspace. An array of the selected check IDs are sent to the base workspace.

If you know a check ID from a previous release, you can find the current check ID using the `ModelAdvisor.lookupCheckID` function. For example, the check ID for **By Product > Simulink Verification and Validation > Modeling Standards > DO-178C/DO-331 Checks > Check safety-related optimization settings** prior to Release 2010b was `D0178B:OptionSet`. Using the `ModelAdvisor.lookupCheckID` function returns:

```
>> NewID = ModelAdvisor.lookupCheckID('D0178B:OptionSet')

NewID =

mathworks.do178.OptionSet
```

Note: If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

See Also

`ModelAdvisor.lookupCheckID`

Create a Function for Checking Multiple Systems

The following tutorial guides you through creating and testing a function to run multiple checks on any model. The function returns the number of failures and warnings.

- 1 In the MATLAB window, select **New > Function**.
- 2 Save the function as `run_configuration.m`.
- 3 In the MATLAB Editor, specify `[output_args]` as `[fail, warn]`.
- 4 Rename the function `run_configuration`.
- 5 Specify `input_args` to `SysList`.
- 6 Inside the function, specify the list of checks to run using the example Model Advisor configuration file:

```
fileName = 'slvndemo_mdldv_config.mat';
```

- 7 Call the `ModelAdvisor.run` function:

```
SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName);
```

- 8 Determine the number of checks that return warnings and failures:

```
fail=0;
warn=0;

for i=1:length(SysResultObjArray)
    fail = fail + SysResultObjArray{i}.numFail;
    warn = warn + SysResultObjArray{i}.numWarn;
end
```

The function should now look like this:

```
function [fail, warn] = run_configuration(SysList)
%RUN_CONFIGURATION Check systems with Model Advsiior
% Check systems given as input and return number of warnings and
% failures.

fileName = 'slvndemo_mdldv_config.mat';
fail=0;
warn=0;

SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName);

for i=1:length(SysResultObjArray)
    fail = fail + SysResultObjArray{i}.numFail;
    warn = warn + SysResultObjArray{i}.numWarn;
end

end
```

- 9 Save the function.

- 10** Test the function. In the MATLAB Command Window, run `run_configuration.m` on the `sldemo_auto_climatecontrol/Heater Control` subsystem:

```
[failures, warnings] = run_configuration(...  
    'sldemo_auto_climatecontrol/Heater Control');
```

- 11** Review the results. Click the Summary Report link to open the Model Advisor Command-Line Summary report.

See Also

`ModelAdvisor.run`

Related Examples

- “Check Multiple Systems in Parallel” on page 25-7
- “Create a Function for Checking Multiple Systems in Parallel” on page 25-8

Check Multiple Systems in Parallel

Checking multiple systems in parallel reduces the processing time required by the Model Advisor to check multiple systems. If you have a Parallel Computing Toolbox license, you can check multiple systems in parallel on a multicore host machine. The Parallel Computing Toolbox does not support 32-bit Windows machines. Each parallel process runs checks on one model at a time.

To enable parallel processing, use the `ModelAdvisor.run` function with `'ParallelMode'` set to `'On'`. By default, `'ParallelMode'` is set to `'Off'`. When you use `ModelAdvisor.run` with `'ParallelMode'` set to `'On'`, MATLAB automatically creates a parallel pool.

See Also

`ModelAdvisor.run`

Related Examples

- “Create a Function for Checking Multiple Systems in Parallel” on page 25-8

Create a Function for Checking Multiple Systems in Parallel

If you have a Parallel Computing Toolbox license and a multicore host machine, you can create the following function to check multiple systems in parallel:

- 1 Create the `run_configuration` function.
- 2 Save the function as `run_fast_configuration.m`.
- 3 In the Editor, change the name of the function to `run_fast_configuration`.
- 4 In the `ModelAdvisor.run` function, set `'ParallelMode'` to `'On'`. When you use `ModelAdvisor.run` with `'ParallelMode'` set to `'On'`, MATLAB automatically creates a parallel pool.

```
SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName,...
    'ParallelMode','On');
```

The function should now look like this:

```
function [fail, warn] = run_fast_configuration(SysList)
%RUN_FAST_CONFIGURATION Check systems in parallel with Model Advisor
% Return number of warnings and failures.
fileName = 'slvndemo_mdldv_config.mat';
fail=0;
warn=0;

SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName,...
    'ParallelMode','On');

for i=1:length(SysResultObjArray)
    fail = fail + SysResultObjArray{i}.numFail;
    warn = warn + SysResultObjArray{i}.numWarn;
end

end
```

- 5 Save the function.
- 6 Test the function. In the MATLAB Command Window, create a list of systems:

```
SysList={'sldemo_auto_climatecontrol/Heater Control',...
    'sldemo_auto_climatecontrol/AC Control','rtwdemo_iec61508'};
```

- 7 Run `run_fast_configuration` on the list of systems:

```
[failures, warnings] = run_fast_configuration(SysList);
```

- 8 Review the results. Click the Summary Report link to open the Model Advisor Command-Line Summary report.

See Also

`ModelAdvisor.run`

Related Examples

- “Check Multiple Systems in Parallel” on page 25-7

Archive and View Results

Archive Results

After you run the Model Advisor programmatically, you can archive the results. The `ModelAdvisor.run` function returns a cell array of `ModelAdvisor.SystemResult` objects, one for each system run. If you save the objects, you can use them to view the results at a later time without rerunning the Model Advisor.

View Results in Command Window

When you run the Model Advisor programmatically, the system-level results of the run are displayed in the Command Window. For example:

```
Systems passed: 0 of 1
Systems with warnings: 1 of 1
Systems failed: 0 of 1
Summary Report
```

The [Summary Report](#) link provides access to the Model Advisor Command-Line Summary report.

You can review additional results in the Command Window by calling the `DisplayResults` parameter when you run the Model Advisor. For example, run the Model Advisor as follows:

```
SysResultObjArray = ModelAdvisor.run('sldemo_auto_climatecontrol/Heater Control',...
    'Configuration','slvndemo_mdadv_config.mat','DisplayResults','Details');
```

The results displayed in the Command Window are:

```
Running Model Advisor
Running Model Advisor on sldemo_auto_climatecontrol/Heater Control
=====
Model Advisor run: 29-Oct-2012 16:30:00
Configuration: slvndemo_mdadv_config.mat
System: sldemo_auto_climatecontrol/Heater Control
System version: 8.1
Created by: The MathWorks Inc.
=====
(1) Warning: Check model diagnostic parameters [check ID: mathworks.maab.jc_0021]
-----
(2) Warning: Check for fully defined interface [check ID: mathworks.iec61508.RootLevelInports]
-----
(3) Pass: Check for unconnected objects [check ID: mathworks.iec61508.UnconnectedObjects]
-----
(4) Pass: Check for blocks not recommended for C/C++ production code deployment
[check ID: mathworks.iec61508.PCGSupport]
-----
Summary:    Pass    Warning    Fail    Not Run
           2         2         0         0
```

```

=====

Systems passed: 0 of 1

Systems with warnings: 1 of 1

Systems failed: 0 of 1
Summary Report

```

To display the results in the Command Window after loading an object, use the `viewReport` function.

View Results in Model Advisor Command-Line Summary Report

When you run the Model Advisor programmatically, a Summary Report link is displayed in the Command Window. Clicking this link opens the Model Advisor Command-Line Summary report. The following graphic is the report that the Model Advisor generates for `run_configuration`.

Model Advisor Command-Line Summary

Simulink version: **7.7** Current run: **09-Feb-2011**
15:29:49

Configuration file: **C:\matlabwork\slvnmvdemo_mdadv_config.mat** Number of systems: **1**

Run Summary

Systems passed	0 of 1
Systems with warnings	1 of 1
Systems failed	0 of 1

Systems Run

System	Passed	Failed	Warnings	Not Run	Model Advisor Report
slidemo_auto_climatecontrol/Heater Control	2	0	2	0	../Report.html

To view the Model Advisor Command-Line Summary report after loading an object, use the `summaryReport` function.

View Results in Model Advisor GUI

In the Model Advisor window, you can view the results of running the Model Advisor programmatically using the `viewReport` function. In the Model Advisor window, you can review results, run checks, fix warnings and failures, and view and save Model Advisor reports.

Tip To fix warnings and failures, you must rerun the check in the Model Advisor window.

View Model Advisor Report

For a single system or check, you can view the same Model Advisor report that you access from the Model Advisor GUI.

To view the Model Advisor report for a system:

- Open the Model Advisor Command-Line Summary report. In the Systems Run table, click the link for the Model Advisor report.
- Use the `viewReport` function.

To view individual check results:

- In the Command Window, generate a detailed report using the `viewReport` function with the `DisplayResults` parameter set to `Details`, and then click the Pass, Warning, or Fail link for the check. The Model Advisor report for the check opens.
- Use the `view` function.

See Also

`ModelAdvisor.run` | `ModelAdvisor.summaryReport` | `view` | `viewReport`

Related Examples

- “Archive and View Model Advisor Run Results” on page 25-14
- “Check Multiple Systems in Parallel” on page 25-7

- “Create a Function for Checking Multiple Systems in Parallel” on page 25-8

More About

- “Run Model Checks”
- “Save and Load Process for Objects”

Archive and View Model Advisor Run Results

This example guides you through archiving the results of running checks so that you can review them at a later time. To simulate archiving and reviewing, the steps in the tutorial detail how to save the results, clear out the MATLAB workspace (simulates shutting down MATLAB), and then load and review the results.

- 1 Call the `ModelAdvisor.run` function:

```
SysResultObjArray = ModelAdvisor.run({'sldemo_auto_climatecontrol/Heater Control'},...  
    'Configuration','slvndemo_mdldv_config.mat');
```

- 2 Save the `SysResultObj` for use at a later time:

```
save my_model_advisor_run SysResultObjArray
```

- 3 Clear the workspace to simulate viewing the results at a different time:

```
clear
```

- 4 Load the results of the Model Advisor run:

```
load my_model_advisor_run SysResultObjArray
```

- 5 View the results in the Model Advisor:

```
viewReport(SysResultObjArray{1},'MA')
```

See Also

`ModelAdvisor.run`

Related Examples

- “Archive and View Results” on page 25-10

Model Metrics

Model Metrics

The DO-331, IEC 61508, IEC 62304, ISO 26262, and EN 50128 standards recommend that your model and code comply with size, complexity, and readability requirements. To help you assess your model, you can use the model metrics available with Simulink Verification and Validation. You can also use APIs to create your own model metrics, compute metrics, and export metric data.

Action	Use	See
Programmatically assess the size, complexity, and readability of your model	Model metric APIs	<ul style="list-style-type: none"> “Collect Model Metrics Programmatically” on page 26-16 “Model Metrics Results API”
Interactively assess the size, complexity, and readability of your model	Model Advisor metric checks	“Run Checks for Model Metrics” on page 24-51
Create your own model metrics	Model metric customization APIs	<ul style="list-style-type: none"> “Create Model Metrics by Using APIs” on page 26-13 “Create Model Metric for Nonvirtual Block Count” on page 26-20

Available Model Metrics

Metric Type	Metric	Description
Count	“Simulink block metric” on page 26-3	Calculates the number of blocks in the model.
	“Subsystem metric” on page 26-4	Calculates the number of subsystems in the model.
	“Library link metric” on page 26-5	Calculates the number of library-linked blocks in the model.

Metric Type	Metric	Description
	“Effective lines of MATLAB code metric” on page 26-5	Calculates the number of effective lines of MATLAB code.
	“Stateflow chart objects metric” on page 26-6	Calculates the number of Stateflow objects.
	“Lines of code for Stateflow blocks metric” on page 26-7	Calculates the number of code lines for the following Stateflow blocks in the model: <ul style="list-style-type: none"> • States • Transitions • Truth tables
	“Subsystem depth metric” on page 26-8	Calculates the subsystem depth of the model.
Complex	“Cyclomatic complexity metric” on page 26-9	Calculates the cyclomatic complexity of the model.
Readability	“Nondescriptive block name metric” on page 26-10	Determines nondescriptive Inport, Outport, and Subsystem block names.
	“Data and structure layer separation metric” on page 26-11	Calculates the data and structure layer separation.

Count Metrics

Simulink block metric

Use this metric to calculate the number of blocks in the model. The results provide the number of blocks at the model and subsystem level.

Action	Use
Programmatically calculate the block count in the model.	Model metric APIs. The MetricID is <code>mathworks.metrics.SimulinkBlockCount</code> .

Action	Use
	<p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Number of blocks • AggregateValue: Number of blocks for component and its descendents • Measures: Same as Value
Interactively calculate the block count in the model.	Model Advisor check “Simulink block metric”.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Subsystem metric

Use this metric to calculate the number of subsystems in the model. The results provide the number of subsystems at the model and subsystem level.

Action	Use
Programmatically calculate the subsystem count in the model.	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.SubSystemCount</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Number of subsystems • AggregateValue: Number of subsystems for component and its descendents • Measures: Same as Value
Interactively calculate the subsystem count in the model.	Model Advisor check “Subsystem metric”.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Library link metric

Use this metric to calculate the number of library-linked blocks in the model. The results provide the number of library-linked blocks at the model and subsystem level.

Action	Use
Programmatically calculate the number of library-linked blocks in the model.	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.LibraryLinkCount</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Number of library linked blocks • AggregateValue: Number of library linked blocks for component and its descendents • Measures: Same as Value
Interactively calculate the number of library-linked blocks in the model.	Model Advisor check “Library link metric”.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Effective lines of MATLAB code metric

Run this metric to calculate the number of effective lines of MATLAB code. The results provide the number of effective lines of MATLAB code for each MATLAB function block and for MATLAB functions in Stateflow charts. Effective lines of MATLAB code are lines of executable code. Empty lines, lines that contain only comments, and lines that contain only an end statement are not considered effective lines of code.

Action	Use
Programmatically calculate the effective lines of code in the model.	Model metric APIs. The MetricID is <code>mathworks.metrics.MatlabLOCCount</code> . For this metric, instances of <code>slmetric.metric.Result</code> provide the following results: <ul style="list-style-type: none"> • Value: Number of effective lines of MATLAB code • AggregateValue: Number of effective lines of MATLAB code for component and its descendents • Measures: Same as Value
Interactively calculate the effective lines of code in the model.	Model Advisor check “Effective lines of MATLAB code metric”.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.
- Does not analyze the content of MATLAB code in external files.

Stateflow chart objects metric

Run this metric to calculate the number of Stateflow objects. For each chart in the model, the results provide the number of the following Stateflow objects:

- Atomic subcharts
- Boxes
- Data objects
- Events
- Graphical functions
- Junctions
- Linked charts
- MATLAB functions
- Notes
- Simulink functions

- States
- Transitions
- Truth tables

Action	Use
Programmatically calculate the number of Stateflow objects	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.StateflowChartObjectCount</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Number of Stateflow objects • AggregateValue: Number of Stateflow objects for component and its descendents • Measures: Not applicable
Interactively calculate the number of Stateflow objects	Model Advisor check “Stateflow chart objects metric”.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Lines of code for Stateflow blocks metric

Use this metric to calculate the number of code lines for the following Stateflow blocks in the model.

- States
- Transitions
- Truth tables

Action	Use
Programmatically calculate the number of code lines for Stateflow blocks in the model.	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.StateflowLOCCount</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p>

Action	Use
	<ul style="list-style-type: none"> • Value: Number of Stateflow block code lines • AggregateValue: Number of Stateflow block code lines for component and its descendents • Measures: Not applicable
Interactively calculate the number of code lines for Stateflow blocks in the model.	Model Advisor check “Lines of code for Stateflow blocks metric”.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Subsystem depth metric

Use this metric to calculate the subsystem depth of the model. The results provide the subsystem depth for each subsystem in the model.

Action	Use
Programmatically calculate the subsystem depth of the model.	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.SubSystemDepth</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Subsystem level, starting from <code>AnalysisRoot</code> • AggregateValue: Not applicable • Measures: Array [maximum depth starting from the component to its leaf nodes in the subsystem hierarchy, same as Value]
Interactively calculate the subsystem depth of the model.	Model Advisor check “Subsystem depth metric”.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Complexity Metrics

Cyclomatic complexity metric

Use this metric to calculate the cyclomatic complexity of the model. The results provide the local and aggregated cyclomatic complexity for the:

- Model
- Subsystems
- Charts
- States in charts
- MATLAB functions

Local complexity is the cyclomatic complexity for objects at their hierarchical level. Aggregated cyclomatic complexity is the cyclomatic complexity of an object and its descendants.

Action	Use
Programmatically calculate the cyclomatic complexity of the model.	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.CyclomaticComplexity</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Local cyclomatic complexity • AggregateValue: Aggregated cyclomatic complexity • Measures: Not applicable
Interactively calculate the cyclomatic complexity of the model.	Model Advisor check “Cyclomatic complexity metric”.

The metric:

- Does not run on library models.

- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models. However, if a block contains a library-linked block, the metric does report the aggregated cyclomatic complexity of the library-linked block.

Readability Metrics

Nondescriptive block name metric

Run this metric to determine nondescriptive Inport, Outport, and Subsystem block names. Default names appended with an integer are nondescriptive block names. The results provide the nondescriptive block names at the model and subsystem level.

Action	Use
Programmatically determine nondescriptive Inport, Outport, and Subsystem block names.	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.DescriptiveBlockNames</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Number of nondescriptive Inport, Outport, and Subsystem block names • AggregateValue: Number of nondescriptive Inport, Outport, and Subsystem block names for component and its descendents • Measures: 1-D vector [total number of Inport blocks, number of Inport blocks with nondescriptive names, total number of Outport blocks, number of Outport blocks with nondescriptive names, total number of Subsystem blocks, number of Subsystem blocks with nondescriptive names]
Interactively determine nondescriptive Inport, Outport, and Subsystem block names.	Model Advisor check “Nondescriptive block name metric”.

The metric:

- Runs on library models.

- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Data and structure layer separation metric

Run this metric to calculate the data and structure layer separation. The results provide the separation at the model and subsystem level.

Action	Use
Programmatically calculate the data and structure layer separation.	<p>Model metric APIs. The MetricID is <code>mathworks.metrics.LayerSeparation</code>.</p> <p>For this metric, instances of <code>slmetric.metric.Result</code> provide the following results:</p> <ul style="list-style-type: none"> • Value: Number of basic blocks on a structural level • AggregateValue: Number of basic blocks on a structural level for component and its descendents • Measures: Not applicable
Interactively calculate the data and structure layer separation.	Model Advisor check “Data and structure layer separation metric”.

For guidelines about blocks on model levels, see MAAB 3.0 guideline db_0143: Similar block types on the model levels.

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of library-linked blocks or referenced models.

Related Examples

- “Run Checks for Model Metrics” on page 24-51
- “Collect Model Metrics Programmatically” on page 26-16
- “Create Model Metric for Nonvirtual Block Count” on page 26-20

More About

- “Create Model Metrics by Using APIs” on page 26-13
- “Model Metrics Results API”
- “Consulting the Model Advisor”

Create Model Metrics by Using APIs

To create your own model metrics, you first construct a new metric class derived from the base class `slmetric.metric.Metric`. The new metric class implements a constructor method and a metric algorithm method.

For an example, see “Create Model Metric for Nonvirtual Block Count” on page 26-20.

Metric Class Constructor

The metric class constructor specifies the following required and optional properties that the new metric class inherits from `slmetric.metric.Metric`.

Inherited Class	Property		Description
slmetric.metric.Metric	Required	ID	Unique metric identifier that retrieves the new metric data.
		Component - Scope	Model components for which the metric is calculated.
	Optional	Compile - Context	Compile mode for metric calculation. If it is not explicitly set, the metric analyzes the noncompiled model. Collecting metric data for compiled models impacts performance.
		Description	Description of the metric.
		Version	Metric version

Metric Algorithm

To develop your own model metrics, create a metric algorithm that calculates the metric data and generates one or more `slmetric.metric.Result` objects with the results. The algorithm calculates data specified by the `Advisor.component.Component` class. The `Advisor.component.Types` class specifies the types of model objects for which you can calculate metric data.

In your metric algorithm, set the `slmetric.metric.Result` object properties as indicated in the table.

Class	Property	Description
slmetric.metric.Result	ComponentID	Set for each result object. You can retrieve the ComponentID from the specifiedAdvisor.component.Component object. Each call to the metric algorithm must return only one result object per ComponentID.
	ComponentPath	If your algorithm returns more than one slmetric.metric.Result object, for each object, set ComponentPath. Do not use the ComponentID property. The ComponentPath must point to an object that is a descendant of the Advisor.component.Component object.
	MetricID	Set to your metric ID.
	Value	Set to the metric value calculated by your algorithm. Each returned result object must have a set value.
	Measures	Optionally, set to return information about the measures used to calculate the metric value.
	Aggregated-Value	Do not set. The metric engine implicitly aggregates the metric data across the model hierarchy.
	Aggregated-Measures	
	UserData	Optionally, set to return additional information.

See Also

Advisor.component.Component | Advisor.component.Types | slmetric.Engine

Related Examples

- “Collect Model Metrics Programmatically” on page 26-16

More About

- “Model Metrics” on page 26-2
- “Model Metrics Results API”

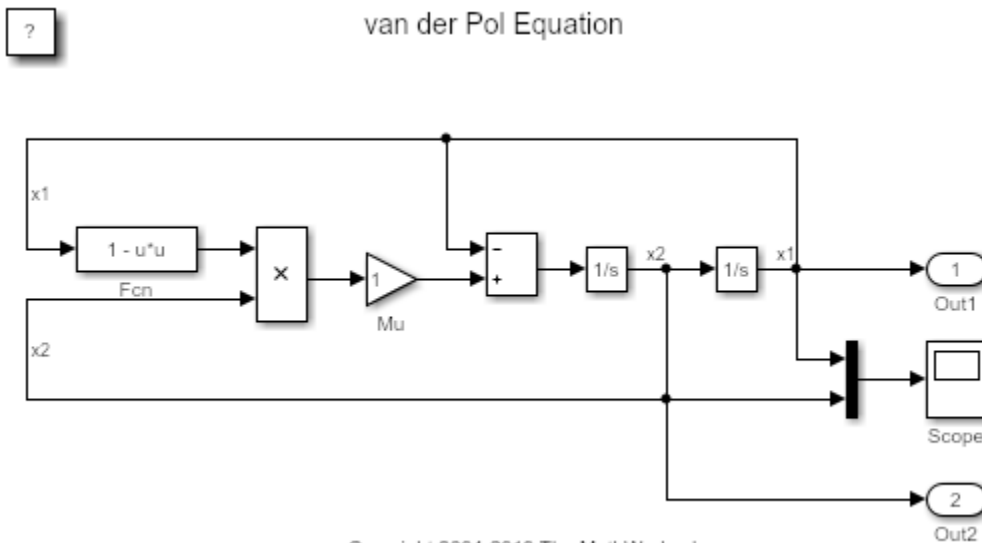
Collect Model Metrics Programmatically

This example shows how to use the metric APIs to programmatically collect subsystem and block count metrics for a model. After collecting metrics for the model, the example shows how to access and export the results.

Example Model

Open model vdp and close the scope.

```
model = 'vdp';
open_system(model);
%
shh = get(0, 'ShowHiddenHandles');
set(0, 'ShowHiddenHandles', 'On');
hscope = findobj(0, 'Type', 'Figure', 'Tag', 'SIMULINK_SIMSCOPE_FIGURE');
close(hscope);
set(0, 'ShowHiddenHandles', shh);
```



Collect Metrics

To collect metric data on models, use instances of `slmetric.Engine`. Using the `getMetrics` method, specify the metrics you want to collect. For this example, specify the block count and subsystem count metrics for the `vdp` model.

```
e=slmetric.Engine();
setAnalysisRoot(e,'Root','vdp','RootType','Model');
execute(e);
rc = getMetrics(e,{'mathworks.metrics.SimulinkBlockCount',...
    'mathworks.metrics.SubSystemCount'});
```

Access Results

To access the metrics for your model, use instances of `slmetric.metric.Result`. In this example, access the block count and subsystem count metrics for the `vdp` model.

Create cell array `metricData` to store the `MetricID`, `ComponentPath`, and `Value` for the metric results. The `MetricID` is the metric, the `ComponentPath` is the path to component for which the metric is calculated, and the `Value` is the metric value. You can use the cell array to export the results to a spreadsheet.

```
metricData ={'MetricID','ComponentPath','Value'};
cnt = 1;
```

For each result, display the `MetricID`, `ComponentPath`, and `Value`. Store the `MetricID`, `ComponentPath`, and `Value` results in `metricData`.

The results show that the:

- Block count for component `vdp/More Info` is 1.
- Block count for component `vdp` is 11.
- Subsystem count for component `vdp/More Info` is 1.
- Subsystem count for component `vdp` is 1.

```
for n=1:length(rc)
    if rc(n).Status == 0
        results = rc(n).Results;

        for m=1:length(results)
            disp(['MetricID: ',results(m).MetricID]);
```

```
        disp([' ComponentPath: ',results(m).ComponentPath]);
        disp([' Value: ',num2str(results(m).Value)]);
        metricData{cnt+1,1} = results(m).MetricID;
        metricData{cnt+1,2} = results(m).ComponentPath;
        metricData{cnt+1,3} = results(m).Value;
        cnt = cnt + 1;
    end
else
    disp(['No results for:',rc(n).MetricID]);
end
disp(' ');
end
```

```
MetricID: mathworks.metrics.SimulinkBlockCount
ComponentPath: vdp/More Info
Value: 1
```

```
MetricID: mathworks.metrics.SimulinkBlockCount
ComponentPath: vdp
Value: 11
```

```
MetricID: mathworks.metrics.SubSystemCount
ComponentPath: vdp/More Info
Value: 1
```

```
MetricID: mathworks.metrics.SubSystemCount
ComponentPath: vdp
Value: 1
```

Export Results

To export the `metricData` results `MetricID`, `ComponentPath`, and `Value` to a spreadsheet, use `xlswrite` to write the contents of `metricData` to `MySpreadsheet.xlsx`.

```
filename = 'MySpreadsheet.xlsx';
xlswrite(filename,metricData)
```

	A	B	C
	MetricID	ComponentPath	Value
Cell	Cell	Number	
1	MetricID	ComponentPath	Value
2	mathworks.metrics.SimulinkBlockCount	vdp/More Info	1
3	mathworks.metrics.SimulinkBlockCount	vdp	11
4	mathworks.metrics.SubSystemCount	vdp/More Info	1
5	mathworks.metrics.SubSystemCount	vdp	1

To export all the metric results to an XML file, use the `exportMetrics` method. For each metric result, the XML file includes the `ComponentID`, `ComponentPath`, `MetricID`, `Value`, `AggregatedValue`, and `Measure`.

```
filename='MyXMLfile.xml';
exportMetrics(e,filename)
```

Close the model vdp.

```
bdclose(model);
```

See Also

[slmetric.Engine](#) | [slmetric.metric.Result](#) | [slmetric.metric.ResultCollection](#)

Related Examples

- “Run Checks for Model Metrics” on page 24-51
- “Create Model Metric for Nonvirtual Block Count” on page 26-20

More About

- “Model Metrics” on page 26-2
- “Model Metrics Results API”

Create Model Metric for Nonvirtual Block Count

This example shows how to use the metric APIs to create a metric for counting nonvirtual blocks in a model. After creating the metric, the example shows how to access and export the results.

Create Metric Class

Using the `createNewMetricClass` function, create a new metric algorithm class named `nonvirtualblockcount`. The function creates file `nonvirtualblockcount.m` in the current working folder. The file contains an empty metric algorithm method and a constructor implementation. For this example, create a working directory.

```
mkdir C:work  
cd C:work
```

```
className = 'nonvirtualblockcount';  
slmetric.metric.createNewMetricClass(className);
```

Create Nonvirtual Block Count Metric

To create the metric algorithm, open the file that the `createNewMetricClass` function created. Using the `slmetric.metric.Metric.algorithm` method, add the metric logic to the file. For example, to edit file, use the command `edit(className)`.

For this example, the file `nonvirtualblockcount_orig.m` contains the logic to create a metric that counts the nonvirtual blocks.

```
copyfile(fullfile(matlabroot, 'examples', 'slvsv', 'nonvirtualblockcount_orig.m'), ...  
          'nonvirtualblockcount_orig.m', 'f');
```

Close the files. Copy and save `nonvirtualblockcount_orig.m` to `nonvirtualblockcount.m`.

```
copyfile('nonvirtualblockcount_orig.m', 'nonvirtualblockcount.m');
```

Register the new metric. For this example, register the nonvirtual block count metric.

```
id_metric = slmetric.metric.registerMetric(className);
```

Collect Metrics

To collect metric data on models, use instances of `slmetric.Engine`. Using the `getMetrics` method, specify the metrics you want to collect. For this example, specify the nonvirtual block count metric for the `sf_car` model.

Load the `sf_car` model.

```
model = 'sf_car';
load_system(model);
```

Create a metric engine object and set the analysis root.

```
e = slmetric.Engine();
setAnalysisRoot(e, 'Root', model, 'RootType', 'Model');
```

Collect metric data for the nonvirtual block count metric.

```
execute(e);
rc = getMetrics(e, id_metric);
```

Display and Export Results

To access the metrics for your model, use instances of `slmetric.metric.Result`. In this example, display the nonvirtual block count metrics for the `sf_car` model. For each result, display the `MetricID`, `ComponentPath`, and `Value`.

```
for n=1:length(rc)
    if rc(n).Status == 0
        results = rc(n).Results;

        for m=1:length(results)
            disp(['MetricID: ', results(m).MetricID]);
            disp([' ComponentPath: ', results(m).ComponentPath]);
            disp([' Value: ', num2str(results(m).Value)]);
            disp(' ');
        end
    else
        disp(['No results for:', rc(n).MetricID]);
    end
end
disp(' ');
```

```
MetricID: nonvirtualblockcount
ComponentPath: sf_car/Engine
Value: 4
```

```
MetricID: nonvirtualblockcount
ComponentPath: sf_car/Vehicle
Value: 0
```

```
MetricID: nonvirtualblockcount
  ComponentPath: sf_car/shift_logic
  Value: NaN

MetricID: nonvirtualblockcount
  ComponentPath: sf_car/shift_logic/selection_state.calc_th
  Value: 5

MetricID: nonvirtualblockcount
  ComponentPath: sf_car/transmission
  Value: 0

MetricID: nonvirtualblockcount
  ComponentPath: sf_car/transmission/Torque
  Converter
  Value: 0

MetricID: nonvirtualblockcount
  ComponentPath: sf_car/transmission/transmission
  ratio
  Value: 3

MetricID: nonvirtualblockcount
  ComponentPath: sf_car
  Value: 3
```

To export all the metric results to an XML file, use the `exportMetrics` method. For each metric result, the XML file includes the `ComponentID`, `ComponentPath`, `MetricID`, `Value`, `AggregatedValue`, and `Measure`.

```
filename='MyXMLfile.xml';
exportMetrics(e,filename);
```

For this example, unregister the nonvirtual block count metric.

```
slmetric.metric.unregisterMetric(id_metric);
```

Close the model `sf_car`.

```
clear;
```



```
bdclose('all');
```

See Also

`slmetric.metric.Metric` | `slmetric.Engine` | `slmetric.metric.Result` |
`slmetric.metric.createNewMetricClass`

Related Examples

- “Collect Model Metrics Programmatically” on page 26-16

More About

- “Model Metrics” on page 26-2
- “Model Metrics Results API”

Customizing the Model Advisor

Overview of Customizing the Model Advisor

Model Advisor Customization

Using Model Advisor APIs and the Model Advisor Configuration Editor, you can:

- Create your own Model Advisor checks.
- Create custom configurations.
- Specify the order in which you make changes to your model.
- Create multiple custom configurations for different projects or modeling guidelines, and switch between these configurations in the Model Advisor.
- Deploy custom configurations to your users.

To	See
Create Model Advisor checks.	“Create Model Advisor Checks”
Format check results.	“Format Check Results” on page 28-57
Create custom Model Advisor configurations.	“Create Custom Configurations” on page 29-2
Specify the order in which you make changes to your model.	“Organize and Deploy Model Advisor Checks”
Deploy custom configurations to your users.	“Organize and Deploy Model Advisor Checks”
Verify that models comply with modeling guidelines.	“Check Model Compliance”

Requirements for Customizing the Model Advisor

Before customizing the Model Advisor:

- If you want to create checks, know how to create a MATLAB script. For more information, see “Create Scripts” in the MATLAB documentation.
- Understand how to access model constructs that you want to check. For example, know how to find block and model parameters. For more information on using utilities for creating check callbacks, see “Common Utilities for Creating Checks” on page 28-6.

Create Model Advisor Checks

Create Model Advisor Checks Workflow

- 1 On your MATLAB path, create a *customization file* named `sl_customization.m`. In this file, create a `sl_customization()` function to register the custom checks that you create and optional process callbacks with the Model Advisor. For detailed information, see “Register Checks and Process Callbacks” on page 28-29.
- 2 Define custom checks and where they appear in the Model Advisor. For detailed information, see “Define Custom Checks” on page 28-33.
- 3 Specify what actions you want the Model Advisor to take for the custom checks by creating a check callback function for each custom check. For detailed information, see “Create Callback Functions and Results” on page 28-41.
- 4 Optionally, specify what automatic fix operations the Model Advisor performs by creating an action callback function. For detailed information, see “Action Callback Function” on page 28-48.
- 5 Optionally, specify startup and post-execution actions by creating a process callback function. For detailed information, see “Define Startup and Post-Execution Actions Using Process Callback Functions” on page 28-30.

Customization File Overview

A *customization file* is a MATLAB file that you create and name `sl_customization.m`. The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

Function	Description	When Required
<code>sl_customization()</code>	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at startup. See “Register Checks and Process Callbacks” on page 28-29.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Define Custom Checks” on page 28-33.	Required for custom checks and to add custom checks to the By Product folder. If the By Product folder is not displayed in the Model Advisor window, select Show By Product Folder from the Settings > Preferences dialog box.
Check callback functions	Defines the actions of the custom checks. See “Create Callback Functions and Results” on page 28-41.	Required for custom checks. You must write one callback function for each custom check.
One or more calls to check input parameters	Specifies input parameters to custom checks. See “Define Check Input Parameters” on page 28-36.	Optional
One or more calls to check list views	Specifies calls to the Model Advisor Result Explorer for custom checks. See “Define Model Advisor Result Explorer Views” on page 28-38.	Optional

Function	Description	When Required
One or more calls to check actions	Specifies actions the software performs for custom checks. See “Define Check Actions” on page 28-39 and “Action Callback Function” on page 28-48.	Optional
One process callback function	Specifies actions to be performed at startup and post-execution time. See “Define Startup and Post-Execution Actions Using Process Callback Functions” on page 28-30.	Optional

This example shows a custom configuration of the Model Advisor that has custom checks defined in custom folders and procedures. The selected check includes input parameters, list view parameters, and actions.

The screenshot displays a software interface with a left-hand navigation pane and a main configuration area on the right.

Left Panel: Model Advisor

- Model Advisor
 - By Product
 - Demo
 - Check Simulink
 - Check Simulink
 - Check model o
 - By Task
 - Demo Factory Gro
 - Check Simulink
 - Check Simulink
 - Check model o
 - My Group
 - Example task with
 - Example task 2
 - Example task 3
 - My Procedure

Example task with input parameter and auto-fix ability

Analysis

Example style three callback

Input Parameters

Skip font checks.

Standard font size Valid font

Result: Not Run

Click **Run This Check**.

Action

Click the button to update all blocks with specified font

Common Utilities for Creating Checks

When you create a custom check, there are common Simulink utilities that you can use to make the check perform different actions. Following is a list of utilities and when to use them. In the Utility column, click the link for more information about the utility.

Utility	Used for..
find_system	Getting handle or path to: <ul style="list-style-type: none"> • Blocks • Lines • Annotations When getting the object, you can: <ul style="list-style-type: none"> • Specify a search depth • Search under masks and libraries
get_param / set_param	Getting and setting system and block parameter values.
inspect	Getting object properties. First you must get a handle to the object.
evalin	Working in the base workspace.
Simulink identifier (SID)	Identifying Simulink blocks, model annotations or Stateflow objects. The SID is a unique number within the model, assigned by Simulink. For details, see “Locate Diagram Components Using Simulink Identifiers”.
Stateflow API	Programmatic access to Stateflow objects.

Create and Add Custom Checks - Basic Examples

To	See
Add a customized check to a Model Advisor By Product > Demo subfolder.	“Add Custom Check to By Product Folder” on page 28-7
Create a Model Advisor pass/fail check.	“Create Customized Pass/Fail Check” on page 28-8
Create a Model Advisor pass/fail check with a fix action.	“Create Customized Pass/Fail Check with Fix Action” on page 28-11

Add Custom Check to By Product Folder

This example shows how to add a custom check to a Model Advisor **By Product > Demo** subfolder. In this example, the customized check does not check model elements.

- 1 In your working directory, create the `sl_customization.m` file, as shown below. This file registers and creates the check registration function `defineModelAdvisorChecks`, which in turn registers the check callback function `SimpleCallback`. The function `defineModelAdvisorChecks` uses a `ModelAdvisor.Root` object to define the check interface.

```
function sl_customization(cm)

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

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Example of a customized check';
rec.TitleTips = 'Added customized check to Product Folder';
rec.setCallbackFcn(@SimpleCallback, 'None', 'StyleOne');
mdladvRoot.publish(rec, 'Demo');

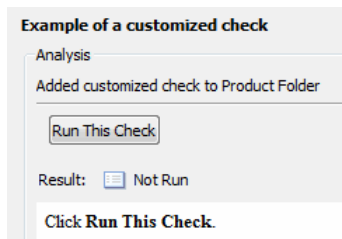
% --- creates SimpleCallback function
function result = SimpleCallback(system);
result={};
```

- 2 Close the Model Advisor and your model if either are open.
- 3 In the Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

- 4 From the MATLAB window, select **New > Simulink Model** to open a new Simulink model window.
- 5 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 6 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.
- 7 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 8 In the left pane, expand the **By Product** folder to display the subfolders. The customized check **Example of a customized check** appears in the **By Product > Demo** subfolder.

The following commands in the `sl_customization.m` file create the right pane of the Model Advisor.



```
rec.Title = 'Example of a customized check';
rec.TitleTips = 'Added customized check to Product Folder';
```

Create Customized Pass/Fail Check

This example shows how to create a Model Advisor pass/fail check. In this example, the Model Advisor checks Constant blocks. If a Constant blocks value is numeric, the check fails.

- 1 In your working directory, update the `sl_customization.m` file, as shown below. This file registers and creates the check registration function `defineModelAdvisorChecks`, which also registers the check callback function `SimpleCallback`. The function `SimpleCallback` creates a check that finds Constant blocks that have numeric values. `SimpleCallback` uses the Model Advisor format template.

```

function sl_customization(cm)

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

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne')

mdladvRoot.publish(rec, 'Demo');

% --- SimpleCallback function that checks constant blocks
function result = SimpleCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
result = {};

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');

ft = ModelAdvisor.FormatTemplate('ListTemplate');
ft.setInformation(['This check looks for constant blocks that...'
    ' use numeric values']);
if ~isempty(blk_with_value)
    ft.setSubResultStatusText(['Check has failed. The following '...
        'Constant blocks have numeric values:']);
    ft.setListObj(blk_with_value);
    ft.setSubResultStatus('warn');
    ft.setRecAction('Parameterize the constant block');
    mdladvObj.setCheckResultStatus(false);
else
    ft.setSubResultStatusText(['Check has passed. No constant blocks'...
        ' with numeric values were found.']);
    ft.setSubResultStatus('pass');
    mdladvObj.setCheckResultStatus(true);
end
ft.setSubBar(0);
result{end+1} = ft;

```

- 2 Close the Model Advisor and your model if either are open.
- 3 In the Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

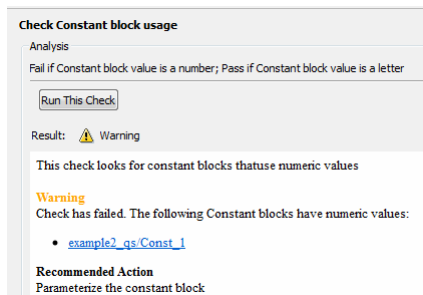
- 4 From the MATLAB window, select **New > Simulink Model** to open a new Simulink model window.

- 5 In the Simulink model window, create two Constant blocks named Const_one and Const_1:
 - Right-click the Const_one block, choose **Constant Parameters**, and assign a **Constant value** of one.
 - Right-click the Const_1 block, choose **Constant Parameters**, and assign a **Constant value** of 1.
 - Save your model as example2_qs



- 6 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 7 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.
- 8 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 9 In the left pane, select **By Product > Demo > Check Constant block usage**.
- 10 Select **Run This Check**. The Model Advisor check fails for the Const_1 block and displays a **Recommended Action** to parametrize the constant block.

The following commands in the `sl_customization.m` file create the right pane in the Model Advisor.



- **Check Constant block usage**

```
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
```

- **Recommended Action**

```
ft.setInformation(['This check looks for constant blocks that'...
    ' use numeric values']);
ft.setSubResultStatusText(['Check has failed. The following '...
    'Constant blocks have numeric values:']);
ft.setListObj(blk_with_value);
ft.setSubResultStatus('warn');
ft.setRecAction('Parameterize the constant block');
```

11 Follow the **Recommended Action** to fix the failed Constant block. In the Model Advisor dialog box:

- Double-click the `example2_qs/Const_1` hyperlink.
- Change **Constant Parameters > Constant value** to two, or a nonnumeric value.
- Rerun the Model Advisor check. Both Constant blocks now pass the check.

Create Customized Pass/Fail Check with Fix Action

This example shows how to create a Model Advisor pass/fail check with a fix action. In this example, the Model Advisor checks Constant blocks. If a Constant block value is numeric, the check fails. The Model Advisor is also customized to create a fix action for the failed checks.

1 In your working directory, update the `sl_customization.m` file, as shown below. This file contains three functions, each of which use the Model Advisor format template:

- `defineModelAdvisorChecks` — Defines the check, creates input parameters, and defines the fix action.
- `simpleCallback` — Creates the check that finds Constant blocks with numeric values.
- `simpleActionCallback` — Creates the fix for Constant blocks that fail the check.

```

function sl_customization(cm)

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

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if '...
    'Constant block value is a letter'];
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne')

% --- input parameters
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Text entry example';
inputParam1.Value='VarNm';
inputParam1.Type='String';
inputParam1.Description='sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec.setInputParameters({inputParam1});

% -- set fix operation
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@simpleActionCallback);
myAction.Name='Fix Constant blocks';
myAction.Description=['Click the button to update all blocks with'...
    ' Text entry example'];
rec.setAction(myAction);

mdladvRoot.publish(rec, 'Demo');

% --- SimpleCallback function that checks constant blocks
function result = SimpleCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
result = {};

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');

ft = ModelAdvisor.FormatTemplate('ListTemplate');
ft.setInformation(['This check looks for constant blocks that'...
    ' use numeric values']);
if ~isempty(blk_with_value)
    ft.setSubResultStatusText(['Check has failed. The following '...
        'Constant blocks have numeric values:']);
    ft.setListObj(blk_with_value);
    ft.setSubResultStatus('warn');
    ft.setRecAction('Parameterize the constant block');
    mdladvObj.setCheckResultStatus(false);

```

```

        mdladvObj.setActionEnable(true);
    else
        ft.setSubResultStatusText(['Check has passed. No constant blocks'...
            'with numeric values were found.']);
        ft.setSubResultStatus('pass');
        mdladvObj.setCheckResultStatus(true);
    end
    ft.setSubBar(0);
    result{end+1} = ft;

    % --- creates SimpleActionCallback function that fixes failed check
    function result = simpleActionCallback(taskobj)
        mdladvObj = taskobj.MAObj;
        result = {};

        system = getfullname(mdladvObj.System);

        % Get the string from the input parameter box.
        inputParams = mdladvObj.getInputParameters;
        textEntryEx = inputParams{1}.Value;

        all_constant_blk=find_system(system,'LookUnderMasks','all',...
            'FollowLinks','on','BlockType','Constant');
        blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');
        ft = ModelAdvisor.FormatTemplate('TableTemplate');
        % Define table col titles
        ft.setColTitles({'Block','Old Value','New Value'})
        for inx=1:size(blk_with_value)
            oldVal = get_param(blk_with_value{inx},'Value');
            ft.addRow({blk_with_value{inx},oldVal,textEntryEx});
            set_param(blk_with_value{inx},'Value',textEntryEx);
        end

        ft.setSubBar(0);
        result = ft;
        mdladvObj.setActionEnable(false);

```

2 Close the Model Advisor and your model if either are open.

3 At the command prompt, enter:

```
Advisor.Manager.refresh_customizations
```

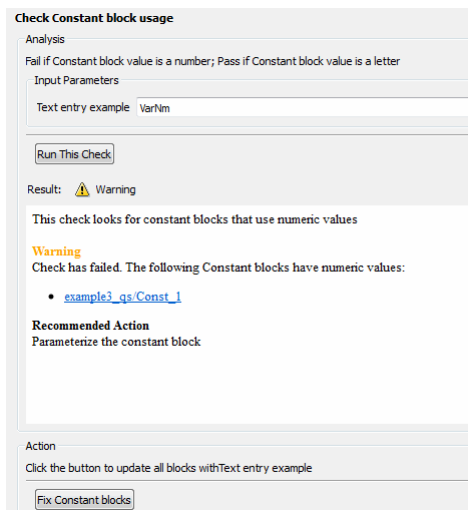
4 From the Command Window, select **New > Simulink Model** to open a new model.

5 In the Simulink model window, create two Constant blocks named Const_one and Const_1:

- Right-click the Const_one block, choose **Constant Parameters**, and assign a **Constant value** of one.
- Right-click the Const_1 block, choose **Constant Parameters**, and assign a **Constant value** of 1.
- Save your model as `example3_qs`.

- 6 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 7 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.
- 8 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 9 In the left pane, select **By Product > Demo > Check Constant block usage**.
- 10 Select **Run This Check**. The Model Advisor check fails for the Const_1 block.

The following commands in the `sl_customization.m` file create the right pane in the Model Advisor.



- **Check Constant block usage**

```

rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if '...
                'Constant block value is a letter'];
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Text entry example';
inputParam1.Value='VarNm';
inputParam1.Type='String';
inputParam1.Description='sample tooltip';

```

```
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec.setInputParameters({inputParam1});
```

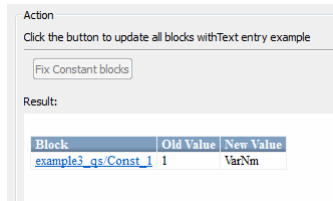
- **Action**

```
myAction.Name='Fix Constant blocks';
myAction.Description=[ 'Click the button to update all blocks with'...
    'Text entry example'];
```

The Model Advisor box has a **Fix Constant blocks** button in the **Action** section of the Model Advisor dialog box.

- 11 In the Model Advisor Dialog box, enter a nonnumeric value in the **Text entry example** parameter field in the **Analysis** section of the Model Advisor dialog box. In this example, the value is VarNm.
- 12 Click **Fix Constant blocks**. The **Const_1 Constant block value** changes from 1 to the nonnumeric value that you entered in step 10. The **Result** section of the dialog box lists the Old Value and New Value of the Const_1 block.

The following commands in the `sl_customization.m` file create the right pane in the Model Advisor.



- **Action**

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');

ft.setColTitles({'Block', 'Old Value', 'New Value'})
for inx=1:size(blk_with_value)
    oldVal = get_param(blk_with_value{inx}, 'Value');
    ft.addRow({blk_with_value{inx}, oldVal, textEntryEx});
    set_param(blk_with_value{inx}, 'Value', textEntryEx);
end
```

- 13 In the Model Advisor dialog box, click **Run This Check**. Both constant blocks now pass the check.

See Also

ModelAdvisor.Action class | ModelAdvisor.FormatTemplate class

More About

- “Register Checks and Process Callbacks” on page 28-29
- “Define Check Input Parameters” on page 28-36

Create Check for Model Configuration Parameters

To verify the configuration parameters for your model, you can create a configuration parameter check.

Decide which configuration parameter settings to use for your model.

Guidelines	See
MathWorks Automotive Advisory Board (MAAB) Control Algorithm Modeling Guidelines	“Model Configuration Options”
High-Integrity System Modeling Guideline	“Configuration Parameter Considerations”
Code Generation Modeling Guidelines	“Code Generation”

- 1 Create an XML data file containing the configuration parameter settings you want to check. You can use `Advisor.authoring.generateConfigurationParameterDataFile` or manually create the file yourself.
- 2 Register the model configuration parameter check using an `sl_customization.m` file.
- 3 Run the check on your models.

Create Data File for Diagnostics Pane Configuration Parameter Check

This example shows how to create a data file for **Diagnostics** pane model configuration parameter check that warns when:

- **Algebraic loop** is set to `none`
- **Minimize algebraic loop** is not set to `error`
- **Block Priority Violation** is not set to `error`

In the example, to create the data file, you use the `Advisor.authoring.generateConfigurationParameterDataFile` function.

At the command prompt, type `vdp`.

In the model window, select **Simulation > Model Configuration Parameters**. In the **Diagnostics** pane, set the configuration parameters as follows:

- **Algebraic loop** to none
- **Minimize algebraic loop** to error
- **Block Priority Violation** to error

Use `Advisor.authoring.generateConfigurationParameterDataFile` to create a data file specifying configuration parameter constraints in the **Diagnostics** pane. Additionally, to create a check with a fix action, set `FixValue` to true. At the command prompt, type:

```
model='vdp';
dataFileName = 'ex_DataFile.xml';
Advisor.authoring.generateConfigurationParameterDataFile(dataFileName,...
model, 'Pane', 'Diagnostics', 'FixValues', true);
```

In the Command Window, select `ex_DataFile.xml`. The data file opens in the MATLAB editor.

- The **Minimize algebraic loop** (command-line: `ArtificialAlgebraicLoopMsg`) configuration parameter tagging specifies a value of error with a fixvalue of error. When you run the configuration parameter check using `ex_DataFile.xml`, the check fails if the **Minimize algebraic loop** setting is not error. The check fix action modifies the setting to error.
- The **Block Priority Violation** (command-line: `BlockPriorityViolationMsg`) configuration parameter tagging specifies a value of error with a fixvalue of error. When you run the configuration parameter check using `ex_DataFile.xml`, the check fails if the **Block Priority Violation** setting is not error. The check fix action modifies the setting to error.

In `ex_DataFile.xml`, edit the **Algebraic loop** (command-line: `AlgebraicLoopMsg`) parameter tagging so that the check warns if the value is none. Because you are specifying a configuration parameter that you do not want, you need a `NegativeModelParameterConstraint`. Additionally, to create a subcheck that does not have a fix action, remove the line with `<fixvalue>` tagging. The tagging for the configuration parameter looks as follows:

```
<!-- Algebraic loop: (AlgebraicLoopMsg)-->
  <NegativeModelParameterConstraint>
```



```

    <parameter>AlgebraicLoopMsg</parameter>
    <value>none</value>
</NegativeModelParameterConstraint>

```

In `ex_DataFile.xml`, delete the lines with tagging for configuration parameters that you do not want to check. The data file `ex_DataFile.xml` has tagging only for **Algebraic loop**, **Minimize algebraic loop** and **Block Priority Violation**. `ex_DataFile.xml` looks similar to:

```

<?xml version="1.0" encoding="utf-8"?>
<customcheck xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.w3schools.com
MySchema.xsd">
  <checkdata>
<!-- Algebraic loop: (AlgebraicLoopMsg)-->
  <NegativeModelParameterConstraint>
    <parameter>AlgebraicLoopMsg</parameter>
    <value>none</value>
  </NegativeModelParameterConstraint>
<!-- Minimize algebraic loop: (ArtificialAlgebraicLoopMsg)-->
  <PositiveModelParameterConstraint>
    <parameter>ArtificialAlgebraicLoopMsg</parameter>
    <value>error</value>
    <fixvalue>error</fixvalue>
  </PositiveModelParameterConstraint>
<!-- Block priority violation: (BlockPriorityViolationMsg)-->
  <PositiveModelParameterConstraint>
    <parameter>BlockPriorityViolationMsg</parameter>
    <value>error</value>
    <fixvalue>error</fixvalue>
  </PositiveModelParameterConstraint>
</checkdata>
</customcheck>

```

Verify the data syntax with `Advisor.authoring.DataFile.validate`. At the command prompt, type:

```

dataFile = 'myDataFile.xml';
msg = Advisor.authoring.DataFile.validate(dataFile);

if isempty(msg)
    disp('Data file passed the XSD schema validation.');
```

```

else
    disp(msg);

```

```
end
```

Create Check for Diagnostics Pane Model Configuration Parameters

This example shows how to create a check for **Diagnostics** pane model configuration parameters using a data file and an `sl_customization` file. First, you register the check using an `sl_customization` file. Using `ex_DataFile.xml`, the check warns when:

- **Algebraic loop** is set to none
- **Minimize algebraic loop** is not set to error
- **Block Priority Violation** is not set to error

The check fix action modifies the **Minimize algebraic loop** and **Block Priority Violation** settings to error.

The check uses the `ex_DataFile.xml` data file created in “Create Data File for Diagnostics Pane Configuration Parameter Check” on page 28-17.

Close the Model Advisor and your model if either are open.

Use the following `sl_customization.m` file to specify and register check **Example: Check model configuration parameters**.

```
function sl_customization(cm)

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

% register items to factory group.
cm.addModelAdvisorTaskFcn(@defineModelAdvisorGroups);

%% defineModelAdvisorChecks
function defineModelAdvisorChecks

    rec = ModelAdvisor.Check('com.mathworks.Check1');
    rec.Title = 'Example: Check model configuration parameters';
    rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
        (system)), 'None', 'StyleOne');
    rec.TitleTips = 'Example check for model configuration parameters';

    % --- data file input parameters
    rec.setInputParametersLayoutGrid([1 1]);
    inputParam1 = ModelAdvisor.InputParameter;
    inputParam1.Name = 'Data File';
    inputParam1.Value = 'ex_DataFile.xml';
    inputParam1.Type = 'String';
    inputParam1.Description = 'Name or full path of XML data file.';
```

```

inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec.setInputParameters({inputParam1});

% -- set fix operation
act = ModelAdvisor.Action;
act.setCallbackFcn(@(task) (Advisor.authoring.CustomCheck.actionCallback...
                        (task)));

act.Name = 'Modify Settings';
act.Description = 'Modify model configuration settings.';
rec.setAction(act);

mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

%% defineModelAdvisorGroups
function defineModelAdvisorGroups
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group 1
rec = ModelAdvisor.FactoryGroup('com.mathworks.Test.factoryGroup');
rec.DisplayName='Example: My Group';
rec.addCheck('com.mathworks.Check1');
mdladvRoot.publish(rec);

```

Create the **Example: Check model configuration parameters**. At the command prompt, enter:

```
Advisor.Manager.refresh_customizations
```

At the command prompt, type vdp.

In the model window, select **Simulation > Model Configuration Parameters**. In the **Diagnostics** pane, to trigger check warnings, set the configuration parameters as follows:

- **Algebraic loop** to none
- **Minimize algebraic loop** to warning
- **Block Priority Violation** to warning

From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.

In the left pane, select **By Task > Example: My Group > Example: Check model configuration parameters**. In the right pane, **Data File** is set to `ex_DataFile.xml`.

Click **Run This Check**. The Model Advisor check warns that the configuration parameters are not set to the values specified in `ex_DataFile.xml`. For configuration

parameters with positive constraint tagging (`PositiveModelParameterConstraint`), the recommended values are obtained from the `value` tagging. For configuration parameters with negative constraint tagging (`NegativeModelParameterConstraint`), the values not recommended are obtained from the `value` tagging.

- **Algebraic loop** (`AlgebraicLoopMsg`) - the `ex_DataFile.xml` tagging does not specify a fix action for `AlgebraicLoopMsg`. The subcheck passes only when the setting is not set to `none`.
- **Minimize algebraic loop** (`ArtificialAlgebraicLoopMsg`) - the `ex_DataFile.xml` tagging specifies a subcheck with a fix action for `ArtificialAlgebraicLoopMsg` that passes only when the setting is `error`. The fix action modifies the setting to `error`.
- **Block priority violation** (`BlockPriorityViolationMsg`) - the `ex_DataFile.xml` tagging specifies a subcheck with a fix action for `BlockPriorityViolationMsg` that does not pass when the setting is `warning`. The fix action modifies the setting to `error`.

In the **Action** section of the Model Advisor dialog box, click **Modify Settings**. Model Advisor updates the configuration parameters for **Block priority violation** and **Minimize algebraic loop**.

Run By Task > Example: My Group > Example: Check model configuration parameters. The check warns because **Algebraic loop** is set to `none`.

In the right pane of the Model Advisor window, use the **Algebraic loop** (`AlgebraicLoopMsg`) link to open the **Simulation > Model Configuration Parameters > Diagnostics**. Set **Algebraic loop** to `warning` or `error`.

Run By Task > Example: My Group > Example: Check model configuration parameters. The check passes.

Data File for Configuration Parameter Check

You use an XML data file to create a configuration parameter check. To create the data file, you can use `Advisor.authoring.generateConfigurationParameterDataFile` or manually create the file yourself. The data file contains tagging that specifies check behavior. Each model configuration parameter specified in the data file is a subcheck. The structure for the data file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:noNamespaceSchemaLocation="http://www.w3schools.com
MySchema.xsd">
<messages>
  <Description>Description of check</Description>
  <PassMessage>Pass message</PassMessage>
  <FailMessage>Fail message</FailMessage>
  <RecommendedActions>Recommended action</RecommendedActions>
</messages>
<checkdata>
<!-- Command-line name of configuration parameter-->
  <PositiveModelParameterConstraint>
    <parameter>Command-line name of configuration parameter</parameter>
    <value>Value that you want configuration parameter to have</value>
    <fixvalue>Specify value for a fix action</fixvalue>
    <dependson>ID of configuration parameter subcheck that must pass
      before this subcheck runs</value>
  </PositiveModelParameterConstraint>
<!-- Command-line name of configuration parameter-->
  <NegativeModelParameterConstraint>
    <parameter>Command-line name of configuration parameter</parameter>
    <value>Value that you do not want configuration parameter to have</value>
    <fixvalue>Specify value for a fix action</fixvalue>
    <dependson>ID of configuration parameter subcheck that must pass
      before this subcheck runs</value>
  </NegativeModelParameterConstraint>
</checkdata>
</customcheck>

```

The `<messages>` tagging is optional.

`Advisor.authoring.generateConfigurationParameterDataFile` does not generate `<messages>` tagging. The `<messages>` tagging contains:

- `<Description>` - Description of the check. Displayed in Model Advisor window. Optional.
- `<PassMessage>` - Pass message displayed in Model Advisor window. Optional.
- `<FailMessage>` - Fail message displayed in Model Advisor window. Optional.
- `<RecommendedActions>` - Recommended actions displayed in Model Advisor window when check does not pass. Optional.

Within the `<checkdata>` tagging, the data file specifies two types of constraints:

- `<PositiveModelParameterConstraint>` - Specifies the configuration parameter setting that you want.
- `<NegativeModelParameterConstraint>` - Specifies the configuration parameter setting that you do not want.

Within the tagging for each of the two types of constraints, for each configuration parameter that you want to check, the data file has the following tags:

- `parameter` - Specifies the configuration parameter that you want to check. The tagging uses the command-line name for the configuration parameter. For example:
 - `<parameter>AlgebraicLoopMsg</parameter>`
 - `<parameter>BlockPriorityViolationMsg</parameter>`
- `value` - For `PositiveModelParameterConstraint` constraints, specifies the setting(s) that you want for the configuration parameter. For `NegativeModelParameterConstraint`, specifies the setting(s) you that do not want for the configuration parameter. You can specify more than one value tag. Values can have the following formats:
 - For a scalar value:
`<value>xyz</value>`
 - For a structure or object:
`<value><param1>xyz</param1><param2>yza</param2></value>`
 - For an array:
`<value><element>value</element><element>value</element></value>`

For example:

- `PositiveModelParameterConstraint` constraints that warn when the setting is not error:
`<value>error</value>`
- `NegativeModelParameterConstraint` constraints that warn when the setting is none or warning:
`<value>none</value>`
`<value>warning</value>`
- `PositiveModelParameterConstraint` constraints that warn when the setting is not a valid structure:
`<value>`
 `<double>a</double>`
 `<single>b</single>`
`</value>`
- `NegativeModelParameterConstraint` constraints that warn when the setting is an invalid array:

```
<value>
  <element>A</element>
  <element>B</element>
</value>
```

- **fixvalue** - Specifies the setting to use when applying the Model Advisor fix action. Optional. Fix values can have the following formats:

- For a scalar value:

```
<fixvalue>xyz</fixvalue>
```

- For a structure or object:

```
<fixvalue><param1>xyz</param1><param2>yza</param2></fixvalue>
```

- For an array:

```
<fixvalue><element>value</element><element>value</element></fixvalue>
```

For example:

- Fix action to modify configuration parameter setting to **error**:

```
<fixvalue>error</fixvalue>
```

- Fix action to modify configuration parameter setting to **warning**:

```
<fixvalue>warning</fixvalue>
```

- Fix action to modify configuration parameter setting to a structure:

```
<fixvalue>
  <double>a</double>
  <single>b</single>
</fixvalue>
```

- Fix action to modify configuration parameter setting to an array:

```
<fixvalue>
  <element>A</element>
  <element>B</element>
</fixvalue>
```

- **dependson** - Specifies a prerequisite subcheck. Optional. For example, to specify that configuration parameter subcheck ID_B must pass before configuration parameter subcheck ID_A is run, use this tagging:

```
<PositiveModelParameterConstraint id="ID_A">
  <dependson>ID_B</value>
</PositiveModelParameterConstraint>
```

Data file tagging specifying a configuration parameter

The following tagging specifies a subcheck for configuration parameter `SolverType`. If the configuration parameter is set to `Fixed-Step`, the subcheck passes.

```
<PositiveModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
</PositiveModelParameterConstraint>
```

Data file tagging specifying configuration parameter with fix action

The following tagging specifies a subcheck for configuration parameter `AlgebraicLoopMsg`. If the configuration parameter is set to `none` or `warning`, the subcheck passes. If the subcheck does not pass, the check fix action modifies the configuration parameter to `error`.

```
<PositiveModelParameterConstraint id="ID_A">
  <parameter>AlgebraicLoopMsg</parameter>
  <value>none</value>
  <value>warning</value>
  <fixvalue>error</value>
</PositiveModelParameterConstraint>
```

Data file tagging specifying an array type configuration parameter

```
<PositiveModelParameterConstraint id="A">
  <parameter>ReservedNameArray</parameter>
  <value>
    <element>A</element>
    <element>B</element>
  </value>
  <value>
    <element>A</element>
    <element>C</element>
  </value>
</PositiveModelParameterConstraint>
```

Data file tagging specifying a structure type configuration parameter with fix action

```
<PositiveModelParameterConstraint id="A">
```



```

<parameter>ReplacementTypes</parameter>
<value>
  <double>a</double>
  <single>b</single>
</value>
<value>
  <double>c</double>
  <single>b</single>
</value>
<fixvalue>
  <double>a</double>
  <single>b</single>
</fixvalue>
</PositiveModelParameterConstraint>

```

Data file tagging specifying configuration parameter with fix action and prerequisite check

The following tagging specifies a subcheck for configuration parameter `SolverType`. The subcheck for `SolverType` runs only after the `ID_B` subcheck passes. If the `ID_B` subcheck does not pass, the subcheck for `SolverType` does not run. The Model Advisor reports that the prerequisite constraint is not met.

If the `SolverType` subcheck runs and `SolverType` is set to `Fixed-Step`, the `SolverType` subcheck passes. If the subcheck runs and does not pass, the check fix action modifies the configuration parameter to `Fixed-Step`.

```

<PositiveModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
  <fixvalue>Fixed-step</value>
  <dependson>ID_B</value>
</PositiveModelParameterConstraint>

```

Data file tagging specifying unwanted configuration parameter

The following tagging specifies a subcheck for configuration parameter `SolverType`. The subcheck does not pass if the configuration parameter is set to `Fixed-Step`.

```

<NegativeModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
</NegativeModelParameterConstraint>

```

Data file tagging specifying unwanted configuration parameter with fix action

The following tagging specifies a subcheck for configuration parameter `SolverType`. If the configuration parameter is set to `Fixed-Step`, the subcheck does not pass. If the subcheck does not pass, the check fix action modifies the configuration parameter to `Variable-Step`.

```
<NegativeModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
  <fixvalue>Variable-step</value>
</NegativeModelParameterConstraint>
```

Data file tagging specifying unwanted configuration parameter with fix action and prerequisite check

The following tagging specifies a check for configuration parameter `SolverType`. The subcheck for `SolverType` runs only after the `ID_B` subcheck passes. If the `ID_B` subcheck does not pass, the subcheck for `SolverType` does not run. The Model Advisor reports that the prerequisite constraint is not met.

If the `SolverType` subcheck runs and `SolverType` is set to `Fixed-Step`, the subcheck does not pass. The check fix action modifies the configuration parameter to `Variable-Step`.

```
<NegativeModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
  <fixvalue>Variable-step</value>
  <dependson>ID_B</value>
</NegativeModelParameterConstraint>
```

See Also

[Advisor.authoring.CustomCheck.actionCallback](#) |
[Advisor.authoring.CustomCheck.checkCallback](#) | [Advisor.authoring.DataFile.validate](#) |
[Advisor.authoring.generateConfigurationParameterDataFile](#)

More About

- “Organize and Deploy Model Advisor Checks”

Register Checks and Process Callbacks

Create `sl_customization` Function

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

Tip

- You can have more than one `sl_customization.m` file on your MATLAB path.
 - Do not place an `sl_customization.m` file that customizes checks and folders in the Model Advisor in your root MATLAB folder or its subfolders, except for the *matlabroot/work* folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.
 - The Model Advisor registers only one process callback function. If you have more than one `sl_customization.m` file on your MATLAB path, the Model Advisor registers the process callback function from the `sl_customization.m` file that has the highest priority.
-

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

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

Register Checks and Process Callbacks

To register custom checks and process callbacks, the customization manager includes the following methods:

- `addModelAdvisorCheckFcn (@checkDefinitionFcn)`

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

The *checkDefinitionFcn* argument is a handle to the function that defines custom checks that you want to add to the Model Advisor as instances of the `ModelAdvisor.Check` class.

- `addModelAdvisorProcessFcn (@modelAdvisorProcessFcn)`

Registers the process callback function for the Model Advisor checks.

This example shows how to register custom checks and a process callback function:

```
function sl_customization(cm)

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

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

Note: If you add custom tasks and folders within the `sl_customization.m` file, include methods for registering the tasks and folders in the `sl_customization` function.

Define Startup and Post-Execution Actions Using Process Callback Functions

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

- **configure** stage: The Model Advisor executes **configure** actions at startup, after checks and tasks have been initialized. At this stage, you can customize how the Model Advisor constructs lists of checks and tasks by modifying **Visible**, **Enable**, and **Value** properties. For example, you can remove, rename, and selectively display checks and tasks in the **By Task** folder.
- **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.

Process Callback Function Arguments

The process callback function uses the following arguments.

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

Process Callback Function

This example shows a process callback function that specifies actions in the **configure** stage, to make only custom checks visible. In the `process_results` stage, this function displays information at the command prompt for checks that do not pass.

```
% Process Callback Function
% Defines actions to execute at startup and post-execution
function [checkCellArray taskCellArray] = ...
    ModelAdvisorProcessFunction(stage, system, checkCellArray, taskCellArray)
switch stage
    % Specify the appearance of the Model Advisor window at startup
    case 'configure'
        for i=1:length(checkCellArray)
            % Hide all checks that do not belong to custom folder
            if isempty(strfind(checkCellArray{i}.ID, 'mathworks.example'))
                checkCellArray{i}.Visible = false;
                checkCellArray{i}.Value = false;
            end
        end
    end
end
```

```
        end
    end
    % Specify actions to perform after the Model Advisor completes execution
    case 'process_results'
        for i=1:length(checkCellArray)
            % Print message if check does not pass
            if checkCellArray{i}.Selected && (strcmp(checkCellArray{i}.Title, ...
                'Check Simulink window screen color'))
                mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
                % Verify whether the check was run and if it failed
                if mdladvObj.verifyCheckRan(checkCellArray{i}.ID)
                    if ~mdladvObj.getCheckResultStatus(checkCellArray{i}.ID)
                        % Display text in MATLAB Command Window
                        disp(['Example message from Model Advisor Process'...
                            ' callback.']);
                    end
                end
            end
        end
    end
end
```

See Also

ModelAdvisor.Check

Related Examples

- Registering Tasks and Folders on page 29-12

More About

- “Define Custom Checks” on page 28-33

Define Custom Checks

About Custom Checks

You can create a custom check to use in the Model Advisor. Creating custom checks provides you with the ability to specify which conditions and configuration settings the Model Advisor reviews.

You define custom checks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Check` class. Define one instance of this class for each custom check that you want to add to the Model Advisor, and register the custom check.

Tip You can add a check to multiple folders by creating a *task*.

Contents of Check Definitions

When you define a Model Advisor check, it contains the information listed in the following table.

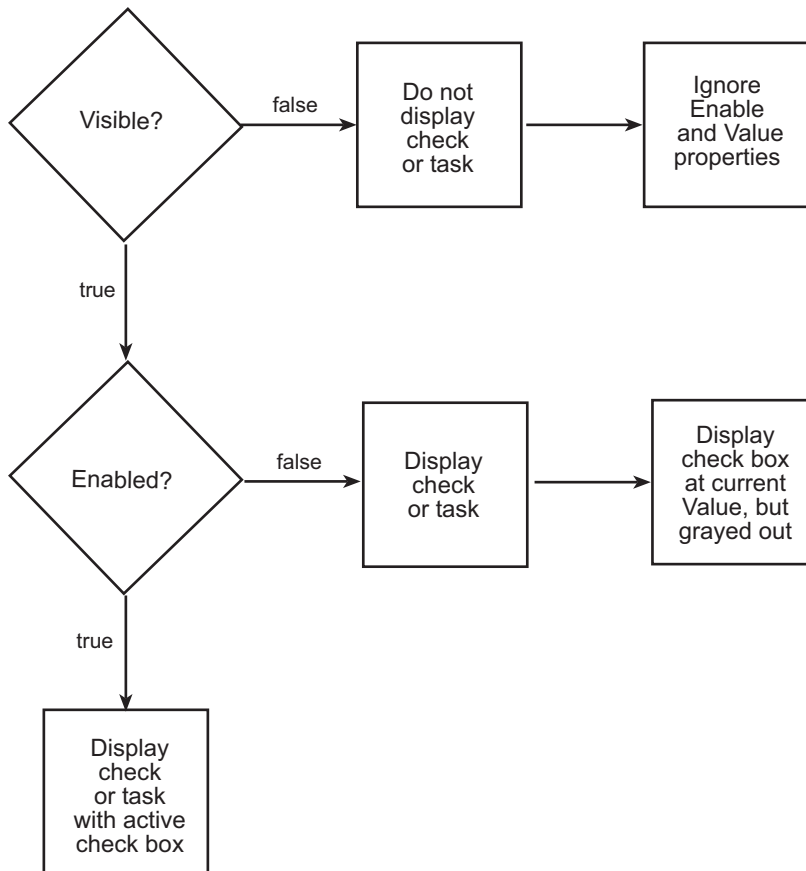
Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this id to access the check.
Handle to check callback function (required)	Function that specifies the contents of a check.
Check name (recommended)	Creates a name for the check that the Model Advisor displays.
Check properties (optional)	Creates a user interface with the check. When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for <code>Visible</code> and <code>LicenseName</code> .
Input Parameters (optional)	Adds input parameters that request input from the user. The Model Advisor uses the input to perform the check.
Action (optional)	Adds automatic fixing action.
Explore Result button (optional)	Adds the Explore Result button that the user clicks to open the Model Advisor Result Explorer.

Display and Enable Checks

You can create a check and specify how it appears in the Model Advisor. You can define when to display a check, or whether a user can select or clear a check using the `Visible`, `Enable`, and `Value` properties of the `ModelAdvisor.Check` class.

Note: When adding checks to the Model Advisor as tasks, specify these properties in the `ModelAdvisor.Task` class. If you specify the properties in both `ModelAdvisor.Check` and `ModelAdvisor.Task`, the `ModelAdvisor.Task` properties take precedence, except for the `Visible` and `LicenseName` properties.

Modify the behavior of the `Visible`, `Enable`, and `Value` properties in a process callback function. The following chart illustrates how these properties interact.



Define Where Custom Checks Appear

Specify where the Model Advisor places custom checks using the following guidelines:

- To place a check in a new folder in the **Model Advisor** root, use the `ModelAdvisor.Group` class.
- To place a check in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.
- To place a check in the **By Product** folder, use the `ModelAdvisor.Root.publish` method. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

Check Definition Function

This example shows a function that defines the custom checks associated with the callback functions described in “Create Callback Functions and Results” on page 28-41. The check definition function returns a cell array of custom checks to be added to the Model Advisor.

The check definitions in the example use the tasks described in Defining Custom Groups on page 29-13.

```
% Defines custom Model Advisor checks
function defineModelAdvisorChecks

% Sample check 1: Informational check
rec = ModelAdvisor.Check('mathworks.example.configManagement');
rec.Title = 'Informational check for model configuration management';
setCallbackFcn(rec, @modelVersionChecksumCallbackUsingFT, 'None', 'StyleOne');
rec.CallbackContext = 'PostCompile';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample check 2: Basic Check with Pass/Fail Status
rec = ModelAdvisor.Check('mathworks.example.unconnectedObjects');
rec.Title = 'Check for unconnected objects';
setCallbackFcn(rec, @unconnectedObjectsCallbackUsingFT, 'None', 'StyleOne');
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample Check 3: Check with Subchecks and Actions
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
rec.Title = 'Check safety-related optimization settings';
setCallbackFcn(rec, @OptimizationSettingCallback, 'None', 'StyleOne');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
setCallbackFcn(modifyAction, @modifyOptimizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
    ' settings that can impact safety.'];

modifyAction.Enable = true;
setAction(rec, modifyAction);
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);
```

Define Check Input Parameters

With input parameters, the you can request input before running the check. Define input parameters using the `ModelAdvisor.InputParameter` class inside a custom check function. You must define one instance of this class for each input parameter that you want to add to a Model Advisor check.

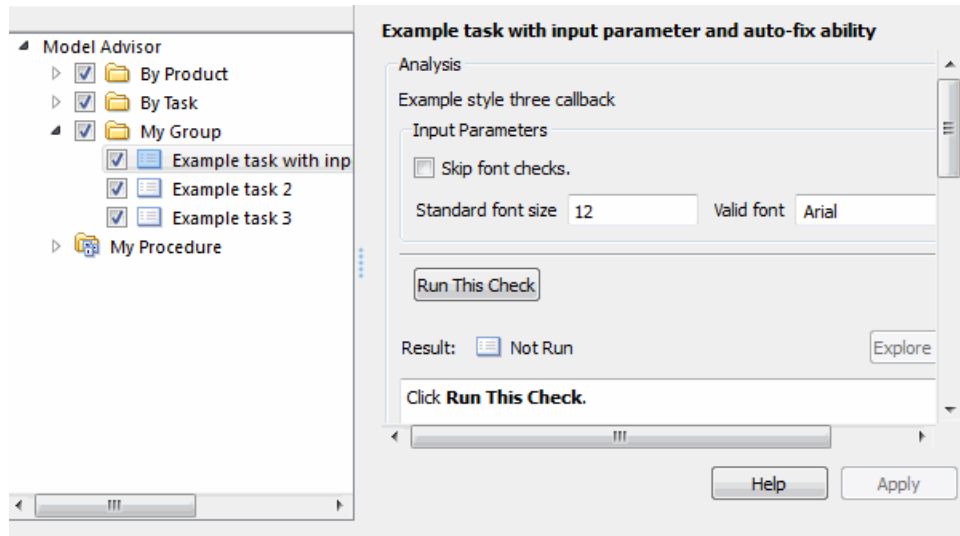
Specify the layout of input parameters with the following methods.

Method	Description
<code>ModelAdvisor.Check.setInputParametersLayoutGrid</code>	Specifies the size of the input parameter grid.
<code>ModelAdvisor.InputParameter.setRowSpan</code>	Specifies the number of rows the parameter occupies in the Input Parameter layout grid.
<code>ModelAdvisor.InputParameter.setColSpan</code>	Specifies the number of columns the parameter occupies in the Input Parameter layout grid.

This example shows how to define input parameters that you add to a custom check. You must include input parameter definitions inside a custom check definition. The following code, when included in a custom check definition, creates three input parameters.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.setInputParametersLayoutGrid([3 2]);
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
```

The Model Advisor displays these input parameters in the right pane, in an **Input Parameters** box.



Define Model Advisor Result Explorer Views

A *list view* provides a way for users to fix check warnings and failures using the Model Advisor Result Explorer. Creating a list view allows you to:

- Add the **Explore Result** button to the custom check in the Model Advisor window.
- Provide the information to populate the Model Advisor Result Explorer.

This example shows how to define 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. You must define one instance of this class for each list view that you want to add to a Model Advisor Result Explorer window.

The following code, when included in a check definition function, adds the **Explore Result** button to the check in the Model Advisor.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
% add 'Explore Result' button
rec.ListViewVisible = true;
```

The following code, when included in a check callback function, provides the information to populate the Model Advisor Result Explorer.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object');
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

Define Check Actions

An *action* provides a way for you to specify an action that the Model Advisor performs to fix a Model Advisor check. When you define an action, the Model Advisor window includes an **Action** box below the **Analysis** box.

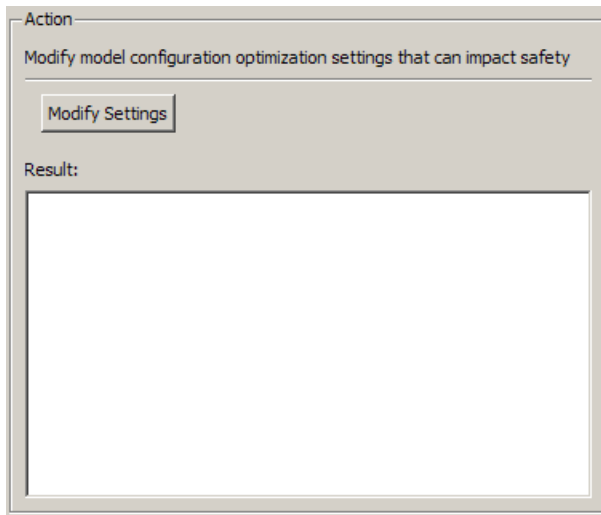
You define actions using the `ModelAdvisor.Action` class inside a custom check function. You must define:

- One instance of this class for each action that you want to take.
- One action callback function for each action.

This example shows the information you need to populate the **Action** box in the Model Advisor. Include this in the check definition function.

```
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
modifyAction.setCallbackFcn(@modifyOptimizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
    ' settings that can impact safety'];
modifyAction.Enable = true;
rec.setAction(modifyAction);
```

The Model Advisor, in the right pane, displays an **Action** box.



See Also

[ModelAdvisor.Action](#) | [ModelAdvisor.Check](#) | [ModelAdvisor.FactoryGroup](#) | [ModelAdvisor.Group](#) | [ModelAdvisor.InputParameter](#) | [ModelAdvisor.Root.publish](#) | [ModelAdvisor.Task](#)

Related Examples

- “Organize Customization File Checks and Folders” on page 29-11

More About

- “Batch-Fix Warnings or Failures”
- “Create Callback Functions and Results” on page 28-41
- Defining Custom Groups on page 29-13
- “Define Startup and Post-Execution Actions Using Process Callback Functions” on page 28-30
- “Register Checks and Process Callbacks” on page 28-29

Create Callback Functions and Results

About Callback Functions

A *callback function* specifies the actions that the Model Advisor performs on a model or subsystem, based on the check or action that the user runs. You must create a callback function for each custom check and action so that the Model Advisor can execute the function when you run the check. All types of callback functions provide one or more return arguments for displaying the results after executing the check or action. In some cases, return arguments are character vectors or cell arrays of character vectors that support embedded HTML tags for text formatting.

To	See
Create in <i>informational callback function</i> for a custom check that finds and displays the model configuration and checksum information.	“Informational Check Callback Function” on page 28-42
Create a <i>simple callback function</i> that indicates if the model passed a check, or to recommend fixing the issue.	“Simple Check Callback Function” on page 28-43
Create <i>detailed check callback function</i> to return and organize results as strings in a layered, hierarchical fashion.	“Detailed Check Callback Function” on page 28-44
Create a callback function that automatically displays hyperlinks for every object returned by the check.	“Check Callback Function with Hyperlinked Results” on page 28-45
Create an <i>action callback function</i> that specifies the actions that the Model Advisor performs on a model or subsystem when you click the action button.	“Action Callback Function” on page 28-48
Create a callback function for a custom check with two subchecks.	“Check With Subchecks and Actions” on page 28-49
Create a callback function for a custom basic check with pass/fail status.	“Basic Check with Pass/Fail Status” on page 28-51

Informational Check Callback Function

This example shows how to create a callback function for a custom *informational* check that finds and displays the model configuration and checksum information. The informational check uses the Result Template API to format the check result.

An *informational* check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.

An informational check does not include the following items in the results:

- The check status. The Model Advisor displays the overall check status, but the status is not in the result.
- A description of the status.
- The recommended action to take when the check does not pass.
- Subcheck results.
- A line below the results.

```
% Sample Check 1 Callback Function: Informational Check
% Find and display model configuration and checksum information
% Informational checks do not have a passed or warning status in the results

function resultDescription = modelVersionChecksumCallbackUsingFT(system)
resultDescription = [];
system = getfullname(system);
model = bdroot(system);

% Format results in a list using Model Advisor Result Template API
ft = ModelAdvisor.FormatTemplate('ListTemplate');
% Add See Also section for references to standards
docLinksFunction{1} = {'IEC 61508-3, Table A.8 (5)' ...
    'Software configuration management'};
setRefLink(ft,docLinksFunction);

% Description of check in results
desc = 'Display model configuration and checksum information.';
% If running the Model Advisor on a subsystem, add note to description
if strcmp(system, model) == false
    desc = strcat(desc, ['<br/>NOTE: The Model Advisor is reviewing a' ...
        ' sub-system, but these results are based on root-level settings.']);
end
setCheckText(ft, desc);

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% If err, use these values
```



```

mdlver = 'Error - could not retrieve Version';
mdlauthor = 'Error - could not retrieve Author';
mdldate = 'Error - could not retrieve Date';
mdlsum = 'Error - could not retrieve CheckSum';

% Get model configuration and checksum information
try
    mdlver = get_param(model, 'ModelVersion');
    mdlauthor = get_param(model, 'LastModifiedBy');
    mdldate = get_param(model, 'LastModifiedDate');
    mdlsum = Simulink.BlockDiagram.getChecksum(model);
    mdlsum = [num2str(mdlsum(1)) ' ' num2str(mdlsum(2)) ' ' ...
              num2str(mdlsum(3)) ' ' num2str(mdlsum(4))];
    mdladvObj.setCheckResultStatus(true); % init to true
catch err
    mdladvObj.setCheckResultStatus(false);
    setSubResultStatusText(ft,err.message);
    resultDescription{end+1} = ft;
    return
end

% Display the results
lbStr = '<br/>';
resultStr = ['Model Version: ' mdlver lbStr 'Author: ' mdlauthor lbStr ...
             'Date: ' mdldate lbStr 'Model Checksum: ' mdlsum];
setSubResultStatusText(ft,resultStr);

% Informational checks do not have subresults, suppress line
setSubBar(ft,false);
resultDescription{end+1} = ft;

```

Simple Check Callback Function

This example shows how to create a simple check callback function. Use a simple check callback function with results formatted using the Result Template API to indicate whether the model passed or failed the check, or to recommend fixing an issue. The keyword for this callback function is `StyleOne`. The check definition requires this keyword.

The check callback function takes the following arguments.

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

Detailed Check Callback Function

This example shows how to create a detailed check callback function. Use the detailed check callback function to return and organize results as strings in a layered, hierarchical fashion. The function provides two output arguments so you can associate text descriptions with one or more paragraphs of detailed information. The keyword for the detailed callback function is `StyleTwo`. The check definition requires this keyword.

The detailed callback function takes the following arguments.

Argument	I/O Type	Description
<code>system</code>	Input	Path to the model or system analyzed by the Model Advisor.
<code>ResultDescription</code>	Output	Cell array of MATLAB character vectors that supports Model Advisor Formatting API calls or embedded HTML tags for text formatting. The Model Advisor concatenates the <code>ResultDescription</code> character vector with the corresponding array of <code>ResultDetails</code> character vectors.
<code>ResultDetails</code>	Output	Cell array of cell arrays, each of which contains one or more character vectors.

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

This example shows 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 example shows how to create a callback function with hyperlinked results. This callback function automatically displays hyperlinks for every object returned by the check so that you can easily locate problem areas in your model or subsystem. The keyword for this type of callback function is `StyleThree`. The check definition requires this keyword.

This callback function takes the following arguments.

Argument	I/O Type	Description
system	Input	Path to the model or system analyzed by the Model Advisor.
ResultDescription	Output	Cell array of MATLAB character vectors 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

Argument	I/O Type	Description
		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 character vector 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.

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

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

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

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

```

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

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

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font for blocks for a uniform appearance in the model. '...
        'The following blocks use a font other than ' regularFontName ': ']);
    ResultDetails{end+1} = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult, 'object');
    myLVParam.Attributes = {'FontName'}; % name is default property
    mdladvObj.setListViewParameters({myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font names '...
        'are identical.']);
    ResultDetails{end+1} = {};
end

% find regular font size blocks
regularBlks = find_system(allBlks, 'FontSize', regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font size for blocks for a uniform appearance in the model. '...
        'The following blocks use a font size other than ' ...
        num2str(regularFontSize) ': ']);
    ResultDetails{end+1} = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font size blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult, 'object');
    myLVParam.Attributes = {'FontSize'}; % name is default property
    mdladvObj.setListViewParameters...
        ({mdladvObj.getListViewParameters{:}, myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font sizes '...
        'are identical.']);
    ResultDetails{end+1} = {};
end

mdladvObj.setActionEnable(needEnableAction);
mdladvObj.setCheckErrorSeverity(1);

```

In the Model Advisor, if you run **Example task with input parameter and auto-fix ability** for the `slvnvdemo_mdadv` model, you can view the hyperlinked results. Clicking the first hyperlink, `slvnvdemo_mdadv/Input`, displays the Simulink model with the Input block highlighted.

Action Callback Function

This example shows how to create an action callback function. An *action callback function* specifies the actions that the Model Advisor performs on a model or subsystem when the user clicks the action button. You must create one callback function for the action that you want to take.

The action callback function takes the following arguments.

Argument	I/O Type	Description
<code>taskobj</code>	Input	The <code>ModelAdvisor.Task</code> object for the check that includes an action definition.
<code>result</code>	Output	MATLAB character vector that supports Model Advisor Formatting API calls or embedded HTML tags for text formatting.

This example shows an action callback function that contains result details.

```
% Get Simulink.ModelAdvisor object
mdladvObj = taskobj.MAObj;
% get result of 'MyCheck'
myResult = mdladvObj.getCheckResult('MyCheck');
% if the check is in style three
[ResultDescription, ResultDetails] = myResult;
```

This example shows an action callback function that fixes the optimization settings that the Model Advisor reviews as defined in “Check With Subchecks and Actions” on page 28-49.

```
% Sample Check 3 Action Callback Function: Check with Subresults and Actions
% Fix optimization settings
function result = modifyOptimizationSetting(taskobj)
% Initialize variables
result = ModelAdvisor.Paragraph();
mdladvObj = taskobj.MAObj;
system = bdroot(mdladvObj.System);

% 'Block reduction' is selected
```

```

% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'BlockReduction'),'off')
    set_param(system,'BlockReduction','off');
    result.addItem(ModelAdvisor.Text( ...
        'Cleared the 'Block reduction' check box.', {'Pass'}));
    result.addItem(ModelAdvisor.LineBreak);
end
% 'Conditional input branch execution' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    set_param(system,'ConditionallyExecuteInputs','off');
    result.addItem(ModelAdvisor.Text( ...
        'Cleared the 'Conditional input branch execution' check box.', ...
        {'Pass'}));
end

```

Action Callback Function with Result Details

This example shows an action callback function that contains result details.

```

% Get Simulink.ModelAdvisor object
mdladvObj = taskobj.MAObj;
% get result of 'MyCheck'
myResult = mdladvObj.getCheckResult('MyCheck');
% if the check is in style three
[ResultDescription, ResultDetails] = myResult;

```

Check With Subchecks and Actions

This example shows how to create a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting, and the second reviews the **Conditional input branch execution** optimization setting.

A *check with subchecks* includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.
- Recommended actions to take when the subcheck does not pass.

- A line between the subcheck results.

```

% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
function ResultDescription = OptimizationSettingCallback(system)
% Initialize variables
system = getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};
system = bdroot(system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,['Check model configuration for optimization settings that...'
    'can impact safety.']);

% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction setting');
setInformation(ft1,'Check whether the ''Block reduction'' check box is cleared. ');
% Add See Also section with references to applicable standards
docLinks{1} = {'Reference DO-178B Section 6.3.4e - Source code ' ...
    'is traceable to low-level requirements'}];
% Review 'Block reduction' optimization
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is cleared. ');
    ResultStatus = true;
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is selected. ');
    setRecAction(ft1,['Clear the ''Optimization > Block reduction'' ...
        'check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check whether the ''Conditional input branch execution''...'
    'check box is cleared.'])
% Add See Also section and references to applicable standards
docLinks{1} = {'Reference DO-178B Section 6.4.4.2 - Test coverage ' ...
    'of software structure is achieved'}];
setRefLink(ft2,docLinks);

```



```

% Last subcheck, suppress line
setSubBar(ft2,false);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The 'Conditional input branch execution' ...
        'check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The 'Conditional input branch execution' ...
        'check box is selected.']);
    setRecAction(ft2,['Clear the 'Optimization > Conditional input branch ' ...
        'execution' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);

```

Basic Check with Pass/Fail Status

This example shows a callback function for a custom *basic* check that finds and reports unconnected lines, input ports, and output ports.

A *basic* check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.
- The status of the check.
- A description of the status.
- Results for the check.
- The recommended actions to take when the check does not pass.

A basic check does not include the following items in the results:

- Subcheck results.
- A line below the results.

```

% Sample Check 2 Callback Function: Basic Check with Pass/Fail Status
% Find and report unconnected lines, input ports, and output ports

```

```

function ResultDescription = unconnectedObjectsCallbackUsingFT(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% Initialize variables
mdladvObj.setCheckResultStatus(false);
ResultDescription = {};
ResultStatus = false; % Default check status is 'Warning'
system = getfullname(system);
isSubsystem = strcmp(bdroot(system), system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object
ft = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
if isSubsystem
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                   'output ports in the subsystem.'];
else
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                   'output ports in the model.'];
end
setCheckText(ft,checkDescStr);

% Add See Also section with references to applicable standards
checkStdRef = 'IEC 61508-3, Table A.3 (3) 'Language subset' ';
docLinkSfunction{1} = {checkStdRef};
setRefLink(ft,docLinkSfunction);

% Basic checks do not have subresults, suppress line
setSubBar(ft,false);

% Check for unconnected lines, inputs, and outputs
sysHandle = get_param(system, 'Handle');
uLines = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'line', ...
    'Connected', 'off');
uPorts = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'port', ...
    'Line', -1);

% Use parents of port objects for the correct highlight behavior
if ~isempty(uPorts)
    for i=1:length(uPorts)
        uPorts(i) = get_param(get_param(uPorts(i), 'Parent'), 'Handle');
    end
end

% Create cell array of unconnected object handles
modelObj = {};
searchResult = union(uLines, uPorts);
for i = 1:length(searchResult)

```

```

        modelObj{i} = searchResult(i);
    end

    % No unconnected objects in model
    % Set result status to 'Pass' and display text describing the status
    if isempty(modelObj)
        setSubResultStatus(ft, 'Pass');
        if isSubsystem
            setSubResultStatusText(ft, ['There are no unconnected lines, ' ...
                'input ports, and output ports in this subsystem.']);
        else
            setSubResultStatusText(ft, ['There are no unconnected lines, ' ...
                'input ports, and output ports in this model.']);
        end
        ResultStatus = true;
    % Unconnected objects in model
    % Set result status to 'Warning' and display text describing the status
    else
        setSubResultStatus(ft, 'Warn');
        if ~isSubsystem
            setSubResultStatusText(ft, ['The following lines, input ports, ' ...
                'or output ports are not properly connected in the system: ' system]);
        else
            setSubResultStatusText(ft, ['The following lines, input ports, or ' ...
                'output ports are not properly connected in the subsystem: ' system]);
        end
        % Specify recommended action to fix the warning
        setRecAction(ft, 'Connect the specified blocks. ');
        % Create a list of handles to problem objects
        setListObj(ft, modelObj);
        ResultStatus = false;
    end
end
% Pass the list template object to the Model Advisor
ResultDescription{end+1} = ft;
% Set overall check status
mdladvObj.setCheckResultStatus(ResultStatus);

```

See Also

ModelAdvisor.Check | ModelAdvisor.FormatTemplate | ModelAdvisor.Task

More About

- Defining Custom Groups on page 29-13
- “Define Custom Checks” on page 28-33
- “Format Check Results” on page 28-57
- “Register Checks and Process Callbacks” on page 28-29

Exclude Blocks From Custom Checks

This example shows how to exclude blocks from custom checks. To save time during model development and verification, you might decide to exclude individual blocks from custom checks in a Model Advisor analysis. By modifying the `sl_customization.m` file to include the `ModelAdvisor.Check.supportExclusion` and `Simulink.ModelAdvisor.filterResultWithExclusion` functions, you can exclude your custom checks from:

- Simulink blocks
- Stateflow charts

This example shows how to exclude blocks from a custom check.

- 1 At the command prompt, type `slvndemo_mdadv`.
- 2 In the model window, double-click **View demo sl_customization.m**.
- 3 To exclude the custom check **Check Simulink block font** from blocks during Model Advisor analysis, make three modifications to the `sl_customization.m` file.
 - a Enable the **Check Simulink block font** check to support check exclusions by using the `ModelAdvisor.Check.supportExclusion` property. You can now exclude the check from model blocks. After `rec.setInputParametersLayoutGrid([3 2]);`, add `rec.supportExclusion = true;`. The check 1 section of the function `defineModelAdvisorChecks` now looks like:


```
% --- sample check 1
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback, 'None', 'StyleThree');
rec.setInputParametersLayoutGrid([3 2]);
rec.supportExclusion = true;
```
 - b Use the `Simulink.ModelAdvisor.filterResultWithExclusion` function to filter model objects causing a check warning or failure with checks that have exclusions enabled. To do this, there are two locations in the `sl_customization.m` file to modify, both in the `[ResultDescription, ResultDetails] = SampleStyleThreeCallback(system)` function:


- After both instances of `searchResult = mdladvObj.filterResultWithExclusion(searchResult);`, add `searchResult = setdiff(allBlks, regularBlks);`.
- In the first location, the function now looks like:

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

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
if ~isempty(searchResult)
```

- In the second location, the function now looks like:

```
% find regular font size blocks
regularBlks = find_system(allBlks, 'FontSize', regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
if ~isempty(searchResult)
```

- 4 Save the `sl_customization.m` file. If you are asked if it is ok to overwrite the file, click **OK**.
- 5 In the model window, double-click **Launch Model Advisor**.
- 6 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 7 In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check fails.
- 8 In the Model Advisor window, click the **Enable highlighting** button (). The blocks causing the **Check Simulink block font** check failure are highlighted in yellow.
- 9 In the model window, right-click the X block and select **Model Advisor > Exclude block only > Check Simulink block font**.
- 10 In the Model Advisor Exclusion Editor, click **OK** to create the exclusion file.
- 11 In the model window, right-click the Input block and select **Model Advisor > Exclude block only > Check Simulink block font**.
- 12 In the Model Advisor Exclusion Editor, click **OK** to update the exclusion file.

- 13** In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check now passes. In the right-pane of the Model Advisor window, you can see the **Check Exclusion Rules** that the Model Advisor during the analysis.
- 14** Close `slvndemo_md1adv`.

See Also

`ModelAdvisor.Check.supportExclusion` | `Simulink.ModelAdvisor`

Related Examples

- “Select Checks and Run Model Advisor”
- Example of Excluding Gain and Outport Blocks From Checks on page 24-27

More About

- Excluding Blocks From Model Advisor Checks on page 24-19
- “Run Model Checks”
- “Highlight Model Check Results”

Format Check Results

Format Results

You can make the analysis results of your custom checks appear similar to each other with minimal scripting using the `ModelAdvisor.FormatTemplate` class.

If this format template does not meet your needs, or if you want to format action results, use the Model Advisor Formatting API to produce formatted outputs in the Model Advisor. The following constructors of the `ModelAdvisor` class allow you to format the output.

Constructor	Description
<code>ModelAdvisor.Text</code>	Create Model Advisor text output.
<code>ModelAdvisor.List</code>	Create list.
<code>ModelAdvisor.Table</code>	Create table.
<code>ModelAdvisor.Paragraph</code>	Create and format paragraph.
<code>ModelAdvisor.LineBreak</code>	Insert line break.
<code>ModelAdvisor.Image</code>	Include image in Model Advisor output.

Format Text

Text is the simplest form of output. You can format text in many different ways. The default text formatting is:

- Empty
- Default color (black)
- Unformatted (not bold, italicized, underlined, linked, subscripted, or superscripted)

To change text formatting, use the `ModelAdvisor.Text` constructor. When you want one type of formatting for all text, use this syntax:

```
ModelAdvisor.Text(content, {attributes})
```

When you want multiple types of formatting, you must build the text.

```
t1 = ModelAdvisor.Text('It is ');
```

```
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' for a uniform appearance in the model.');
```

```
result = [t1, t2, t3, t4, t5];
```

Add ASCII and Extended ASCII characters using the MATLAB `char` command. For more information, see the `ModelAdvisor.Text` class page.

Format 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. You can create lists with indented subsections, formatted as either numbered or bulleted.

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

```
topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1',{'keyword','bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2',{'keyword','bold'}), subList]);
```

Format Tables

The default table formatting is:

- Default color (black)
- Left justified
- Bold title, row, and column headings

Change table formatting using the `ModelAdvisor.Table` constructor.

This example creates a subtable within a table.

```
table1 = ModelAdvisor.Table(1,1);
table2 = ModelAdvisor.Table(2,3);
table2.setHeading('Table 2');
table2.setHeadingAlign('center');
table2.setColHeading(1, 'Header 1');
table2.setColHeading(2, 'Header 2');
table2.setColHeading(3, 'Header 3');
table1.setHeading('Table 1');
table1.setEntry(1,1,table2);
```


Table 1		
Table 2		
Header 1	Header 2	Header 3

This example creates a table with five rows and five columns containing randomly generated numbers. Use the MATLAB code in a callback function to create the table. The Model Advisor displays `table1` in the results.

```
% ModelAdvisor.Table example

matrixData = rand(5,5) * 10^5;

% initialize a table with 5 rows and 5 columns (heading rows not counting)
table1 = ModelAdvisor.Table(5,5);

% set column headings
for n=1:5
    table1.setColHeading(n, ['Column ', num2str(n)]);
end

% set alignment of second column heading
table1.setColHeadingAlign(2, 'center');

% set column width of second column
table1.setColWidth(2, 3);

% set row headings
for n=1:5
    table1.setRowHeading(n, ['Row ', num2str(n)]);
end

% set Table content
for rowIndex=1:5
    for colIndex=1:5
        table1.setEntry(rowIndex, colIndex, ...
            num2str(matrixData(rowIndex, colIndex)));

        % set alignment of entries in second row
        if colIndex == 2
            table1.setEntryAlign(rowIndex, colIndex, 'center');
        end
    end
end

% overwrite content of cell 3,3 with a ModelAdvisor.Text
```

```
text = ModelAdvisor.Text('Example Text');
table1.setEntry(3,3, text)
```

	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	81472.3686	9754.0405	15761.3082	14188.6339	65574.0699
Row 2	90579.1937	27849.8219	97059.2782	42176.1283	3571.1679
Row 3	12698.6816	54688.1519	Example Text	91573.5525	84912.9306
Row 4	91337.5856	95750.6835	48537.5649	79220.733	93399.3248
Row 5	63235.9246	96488.8535	80028.0469	95949.2426	67873.5155

Format Paragraphs

You must handle paragraphs explicitly because most markup languages do not support line breaks. The default paragraph formatting is:

- Empty
- Default color (black)
- Unformatted, (not bold, italicized, underlined, linked, subscripted, or superscripted)
- Aligned left

If you want to change paragraph formatting, use the `ModelAdvisor.Paragraph` class.

Formatted Output

The following is the example from “Simple Check Callback Function” on page 28-43, reformatted using the Model Advisor Formatting API.

```
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
if strcmp(get_param(bdroot(system), 'ScreenColor'),'white')
    result = ModelAdvisor.Text('Passed',{ 'pass' });
    mdladvObj.setCheckResultStatus(true);
else
    msg1 = ModelAdvisor.Text(...
        ['It is recommended to select a Simulink window screen color'...
        ' of white for a readable and printable model. Click ']);
    msg2 = ModelAdvisor.Text('here');
    msg2.setHyperlink('matlab: set_param(bdroot, 'ScreenColor', 'white')');
    msg3 = ModelAdvisor.Text(' to change screen color to white. ');
    result = [msg1, msg2, msg3];
    mdladvObj.setCheckResultStatus(false);
```

end

Format Linebreaks

You can add a line break between two lines of text with the `ModelAdvisor.LineBreak` constructor.

```
result = ModelAdvisor.Paragraph;  
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

Format Images

To include an image in Model Advisor output, use the `ModelAdvisor.Image` constructor. To create an `Image` object, use this syntax.

```
image_obj = ModelAdvisor.Image;
```

See Also

[ModelAdvisor.Check](#) | [ModelAdvisor.FormatTemplate](#) | [ModelAdvisor.Task](#)

Related Examples

- “Simple Check Callback Function” on page 28-43

More About

- [Defining Custom Groups](#) on page 29-13
- [“Define Custom Checks”](#) on page 28-33

Create Custom Configurations by Organizing Checks and Folders

Create Custom Configurations

You can use the Model Advisor APIs and Model Advisor Configuration Editor available with Simulink Verification and Validation to do the tasks listed in the following table.

To	See
Create custom configurations by organizing Model Advisor checks and folders.	“Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 29-5
Specify the order in which you make changes to your model.	“Create a Procedural-Based Configuration” on page 30-5
Deploy custom configuration to your users.	“How to Deploy Custom Configurations” on page 31-3

Create Configurations by Organizing Checks and Folders

To customize the Model Advisor with MathWorks and custom checks, perform the following tasks:

- 1** Review the information in “Requirements for Customizing the Model Advisor” on page 27-2.
- 2** Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
- 3** Organize the checks into new and existing folders to create custom configurations. See “Organize and Deploy Model Advisor Checks”.
 - a** Identify which checks you want to include in your custom Model Advisor configuration. You can use MathWorks checks and/or custom checks.
 - b** Create the custom configurations using either of the following:
 - Model Advisor Configuration Editor - “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 29-5.
 - A customization file - “Organize Customization File Checks and Folders” on page 29-11.
 - c** Verify the custom configuration. See “Verify and Use Custom Configurations” on page 29-19.
- 4** Optionally, deploy the custom configurations to your users. See “Organize and Deploy Model Advisor Checks”.
- 5** Verify that models comply with modeling guidelines. See “Run Model Checks”.

Create Procedural-Based Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

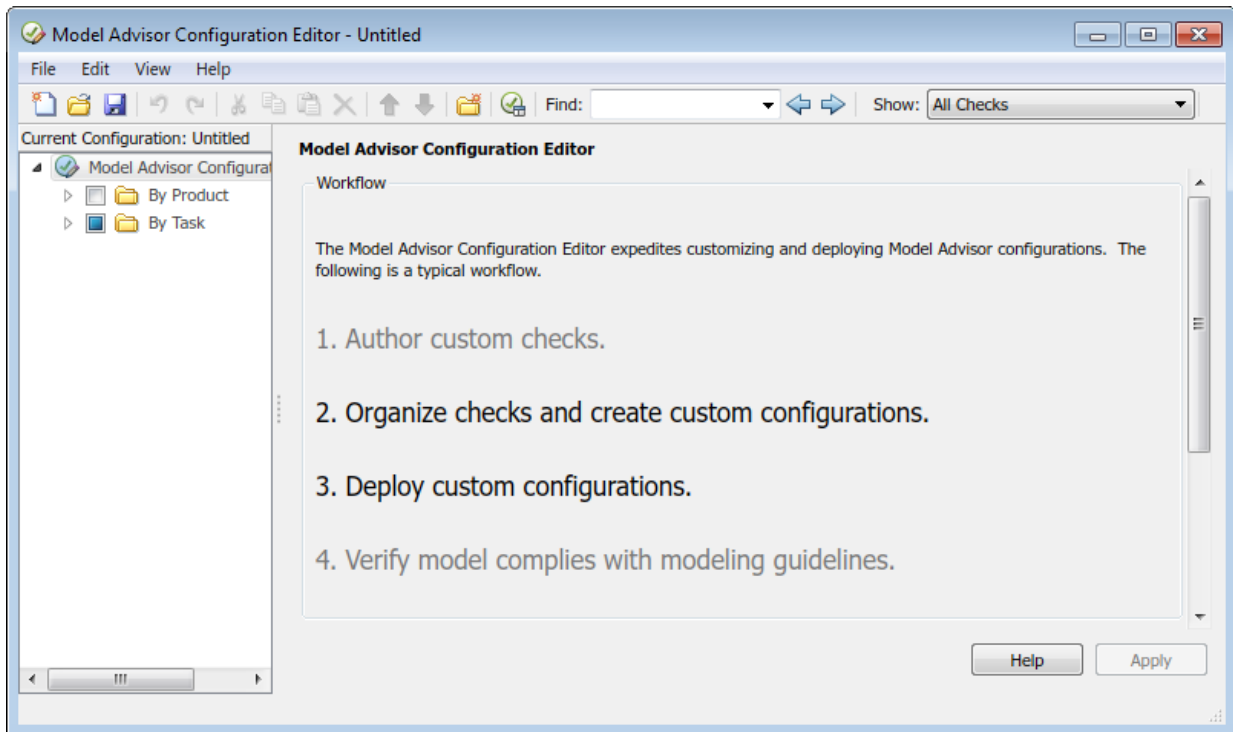
To create a procedural-based configuration, perform the following tasks:

- 1 Review the information in “Requirements for Customizing the Model Advisor” on page 27-2.
- 2 Decide on order of changes to your model.
- 3 Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.
- 4 Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
- 5 Organize the checks into procedures for a procedural-based configuration. See “Create a Procedural-Based Configuration” on page 30-5.
 - a Create procedures using the procedure API. For detailed information, see “Create Procedures Using the Procedures API” on page 30-2.
 - b Create the custom configuration by using a customization file. See “Organize Customization File Checks and Folders” on page 29-11.
 - c Verify the custom configuration as described in “Verify and Use Custom Configurations” on page 29-19.
- 6 Optionally, deploy the custom configurations to your users. For detailed information, see “Organize and Deploy Model Advisor Checks”.
- 7 Verify that models comply with modeling guidelines. For detailed information, see “Run Model Checks”.

Organize Checks and Folders Using the Model Advisor Configuration Editor

Overview of the Model Advisor Configuration Editor

When you start the Model Advisor Configuration Editor, two windows open; the Model Advisor Configuration Editor and the Model Advisor Check Browser. The Configuration Editor window consists of two panes: the Model Advisor Configuration Editor hierarchy and the Workflow. The Model Advisor Configuration Editor hierarchy lists the checks and folders in the current configuration. The Workflow on the right shows the common workflow you use to create a custom configuration.

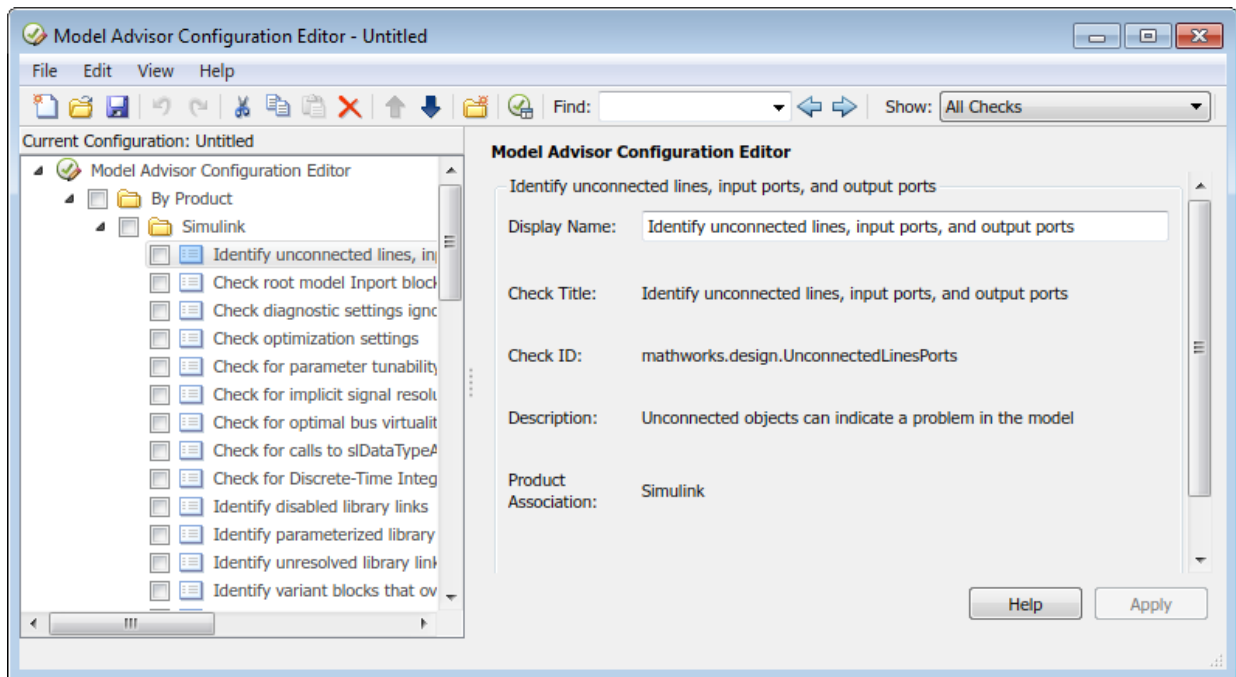


Model Advisor Configuration Editor

If you want the Model Advisor Configuration Editor hierarchy to list only the checks configured for edit-time checking, in the **Show** field, select **Edit-Time Supported**

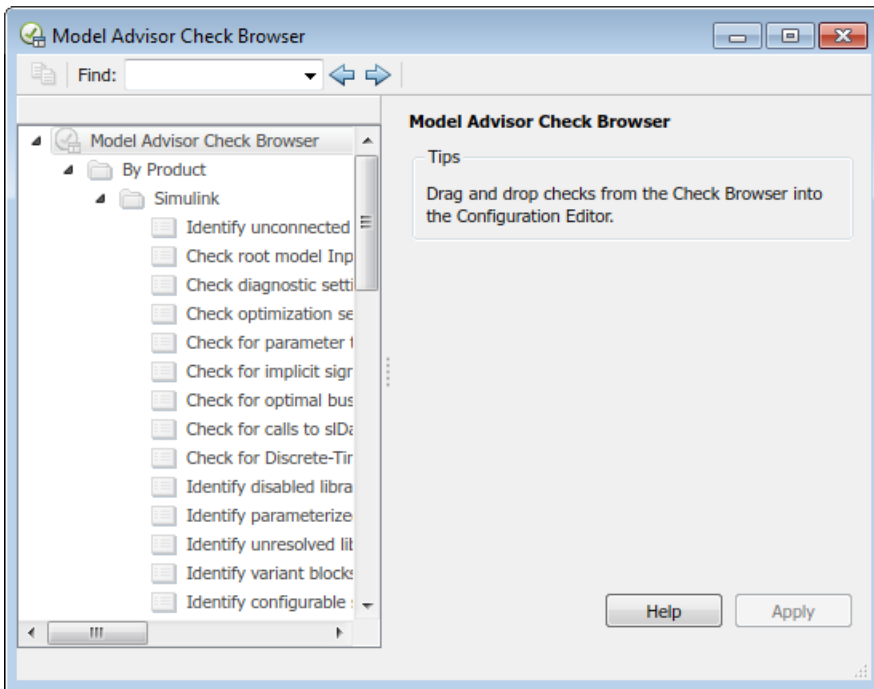
Checks. Or, in the model window, select **Analysis > Model Advisor > Configure Advisor Edit-Time Checks**.

When you select a folder or check in the Model Advisor Configuration Editor hierarchy, the Workflow pane changes to display information about the check or folder. You can change the display name of the check or folder in this pane.



The Model Advisor Check Browser window includes a read-only list of available checks. If you delete a check in the Model Advisor Configuration Editor, you can retrieve a copy of it from the Model Advisor Check Browser.

Tip If you use a process callback function in a `sl_customization` file to hide checks and folders, the Model Advisor Configuration Editor and Model Advisor Check Browser do not display the hidden checks and folders. For a complete list of checks and folders, remove process callback functions and update the Simulink environment.



Model Advisor Check Browser

Using the Model Advisor Configuration Editor, you can perform the following actions.

To...	Select...
Create new configurations	File > New
Find checks and folders in the Model Advisor Check Browser	View > Check Browser
Add checks and folders to the configuration	Edit > Copy Edit > Paste Edit > New folder The check or folder and drag and drop
Remove checks and folders from the configuration	Edit > Delete Edit > Cut
Reorder checks and folders	Edit > Move up

To...	Select...
	Edit > Move down The check or folder and drag and drop
Rename checks and folders	The check or folder and edit Display Name in right pane.
Note: MathWorks folder display names are restricted. When you rename a folder, you cannot use the restricted display names.	
Allow or gray out the check box control for checks and folders	Edit > Enable Edit > Disable
Tip This capability is equivalent to enabling checks, described in “Display and Enable Checks” on page 28-34.	
Save the configuration as a MAT file for use and distribution	File > Save File > Save As
Set the configuration so it opens by default in the Model Advisor	File > Set Current Configuration as Default
Restore the MathWorks default configuration	File > Restore Default Configuration
Load and edit saved configurations	File > Open

Start the Model Advisor Configuration Editor

Before starting the Model Advisor Configuration Editor, verify that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.

Note:

- The Model Advisor Configuration Editor uses the `slprj` folder in the code generation folder. If the `slprj` folder does not exist in the code generation folder, the Model Advisor Configuration Editor creates it.
-

- 1 To include custom checks in the new Model Advisor configuration, update the Simulink environment to include your `sl_customization.m` file.
- 2 Start the Model Advisor Configuration Editor.

To start the Model Advisor Configuration Editor...	Do this:
Programmatically	At the MATLAB command line, enter <code>Simulink.ModelAdvisor.openConfigUI</code> .
From the Model Advisor	<ol style="list-style-type: none"> a Start the Model Advisor. b Select Settings > Open Configuration Editor.

The Model Advisor Configuration Editor and Model Advisor Check Browser windows open.

- 3 Optionally, to edit an existing configuration in the Model Advisor Configuration Editor window:
 - a Select **File > Open**.
 - b In the Open dialog box, navigate to the configuration file that you want to edit.
 - c Click **Open**.

Organize Checks and Folders Using the Model Advisor Configuration Editor

The following tutorial steps you through creating a custom configuration.

- 1 Open the Model Advisor Configuration Editor at the MATLAB command line by entering `Simulink.ModelAdvisor.openConfigUI` .
- 2 In the Model Advisor Configuration Editor, in the left pane, delete the **By Product** and **By Task** folders, to start with a blank configuration.
- 3 Select the root node which is labeled Model Advisor Configuration Editor.
- 4 In the toolbar, click the **New Folder** button to create a folder.
- 5 In the left pane, select the new folder.
- 6 In the right pane, edit **Display Name** to rename the folder. For the purposes of this tutorial, rename the folder to **Review Optimizations**.
- 7 In the Model Advisor Check Browser window, in the **Find** field, enter `optimization` to find **Simulink > Check optimization settings**.

- 8** Drag and drop **Check optimization settings** into **Review Optimizations**.
- 9** In the Model Advisor Check Browser window, find **Simulink Verification and Validation > Modeling Standards > DO-178C/DO-331Checks > Check safety-related optimization settings**.
- 10** Drag and drop **Check safety-related optimization settings** into **Review Optimizations**.
- 11** In the Model Advisor Configuration Editor window, expand **Review Optimizations**.
- 12** Rename **Check optimization settings** to **Check Simulink optimization settings**.
- 13** Select **File > Save As** to save the configuration.
- 14** Name the configuration `optimization_configuration.mat`.
- 15** Close the Model Advisor Configuration Editor window.

Tip To move a check to the first position in a folder:

- 1** Drag the check to the second position.
 - 2** Right-click the check and select **Move up**.
-

See Also

ModelAdvisor.Check | Simulink.ModelAdvisor

Related Examples

- “Update the Environment to Include Your `sl_customization` File” on page 29-19

Organize Customization File Checks and Folders

Abstract

Use the `sl_customization.m` file to organize checks and folders.

Customization File Overview

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

Function	Description	Required or Optional
<code>sl_customization()</code>	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at startup. See “Register Checks and Process Callbacks” on page 28-29.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Define Custom Checks” on page 28-33.	Required for custom checks and to add custom checks to the By Product folder.
One or more task definitions	Defines custom tasks. See “Define Custom Tasks” on page 29-13.	Required to add custom checks to the Model Advisor, except when adding the checks to the By Product folder. Write one task for each check that you add to the Model Advisor.
One or more groups	Defines custom groups. See “Define Custom Tasks” on page 29-13.	Required to add custom tasks to new folders in the Model Advisor, except when adding a new subfolder to the By Product folder. Write one group definition for each new folder.
One process callback function	Specifies actions that Simulink performs at startup and post-execution time. See “Define	Optional.

Function	Description	Required or Optional
	Startup and Post-Execution Actions Using Process Callback Functions” on page 28-30.	

If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

Register Tasks and Folders

Create `sl_customization` Function

To add tasks and folders to the Model Advisor, create the `sl_customization.m` file on your MATLAB path. Then create the `sl_customization()` function in the `sl_customization.m` file on your MATLAB path.

Tip

- You can have more than one `sl_customization.m` file on your MATLAB path.
 - Do not place an `sl_customization.m` file that customizes the Model Advisor in your root MATLAB folder or its subfolders, except for the `matlabroot/work` folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.
-

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

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

Register Tasks and Folders

The customization manager provides the following methods for registering custom tasks and folders:

- `addModelAdvisorTaskFcn (@factorygroupDefinitionFcn)`

Registers the tasks that you define in *factorygroupDefinitionFcn* to the **By Task** folder of the Model Advisor.

The *factorygroupDefinitionFcn* argument is a handle to the function that defines the checks to add to the Model Advisor as instances of the `ModelAdvisor.FactoryGroup` class.

- `addModelAdvisorTaskAdvisorFcn (@taskDefinitionFcn)`

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

The *taskDefinitionFcn* argument is a handle to the function that defines custom tasks and folders. Simulink adds the checks and folders to the Model Advisor as instances of the `ModelAdvisor.Task` or `ModelAdvisor.Group` classes.

The following example shows how to register custom tasks and folders:

```
function sl_customization(cm)

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

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

Note: If you add custom checks and process callbacks within the `sl_customization.m` file, include methods for registering the checks and process callbacks in the `sl_customization` function.

Define Custom Tasks

Add Check to Custom or Multiple Folders Using Tasks

You can use custom tasks for adding checks to the Model Advisor, either in multiple folders or in a single, custom folder. You define custom tasks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Task` class. Define one instance of this class for each custom task that you want to add to the Model Advisor. Then register the custom task. The following sections describe how to define custom tasks.

To add a check to multiple folders or a single, custom folder:

- 1 Create a check using the `ModelAdvisor.Check` class.
- 2 Register a task wrapper for the check.
- 3 If you want to add the check to folders that are not already present, register and create the folders using the `ModelAdvisor.Group` class.
- 4 Add a check to the task using the `ModelAdvisor.Task.setCheck` method.
- 5 Add the task to each folder using the `ModelAdvisor.Task.addTask` method and the task ID.

Create Custom Tasks Using MathWorks Checks

You can add MathWorks checks to your custom folders by defining the checks as custom tasks. When you add the checks as custom tasks, you identify checks by the check ID.

To find MathWorks check IDs:

- 1 In the Model Advisor, select **View > Source** tab.
- 2 Navigate to the folder that contains the MathWorks check.
- 3 In the right pane, click **Source**. The Model Advisor displays the **Title**, **TitleID**, and **Source** information for each check in the folder.
- 4 Select and copy the **TitleID** of the check that you want to add as a task.

Display and Enable Tasks

The `Visible`, `Enable`, and `Value` properties interact the same way for tasks as they do for checks.

Define Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor using the following guidelines:

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Task Definition Function

The following example shows a task definition function. This function defines three tasks.

```

% Defines Model Advisor tasks and a custom folder
% Add checks to a custom folder using task definitions
function defineTaskAdvisor
mdladvRoot = ModelAdvisor.Root;

% Define task that uses Sample Check 1: Informational check
MAT1 = ModelAdvisor.Task('mathworks.example.task.configManagement');
MAT1.DisplayName = 'Informational check for model configuration management';
MAT1.Description = 'Display model configuration and checksum information.';
setCheck(MAT1, 'mathworks.example.configManagement');
mdladvRoot.register(MAT1);

% Define task that uses Sample Check 2: Basic Check with Pass/Fail Status
MAT2 = ModelAdvisor.Task('mathworks.example.task.unconnectedObjects');
MAT2.DisplayName = 'Check for unconnected objects';
setCheck(MAT2, 'mathworks.example.unconnectedObjects');
MAT2.Description = ['Identify unconnected lines, input ports, and output ' ...
                    'ports in the model or subsystem.'];
mdladvRoot.register(MAT2);

% Define task that uses Sample Check 3: Check with Subresults and Actions
MAT3 = ModelAdvisor.Task('mathworks.example.task.optimizationSettings');
MAT3.DisplayName = 'Check safety-related optimization settings';
MAT3.Description = ['Check model configuration for optimization ' ...
                    'settings that can impact safety.'];
MAT3.setCheck('mathworks.example.optimizationSettings');
mdladvRoot.register(MAT3);

% Custom folder definition
MAG = ModelAdvisor.Group('mathworks.example.ExampleGroup');
MAG.DisplayName = 'My Group';
% Add tasks to My Group folder
addTask(MAG, MAT1);
addTask(MAG, MAT2);
addTask(MAG, MAT3);
% Add My Group folder to the Model Advisor under 'Model Advisor' (root)
mdladvRoot.publish(MAG);

```

Define Custom Folders

About Custom Folders

Use folders to group checks in the Model Advisor by functionality or usage. You define custom folders in:

- A factory group definition function that specifies the properties of each instance of the `ModelAdvisor.FactoryGroup` class.
- A task definition function that specifies the properties of each instance of the `ModelAdvisor.Group` class.

Define one instance of the group classes for each folder that you want to add to the Model Advisor.

Add Custom Folders

To add a custom folder:

- 1 Create the folder using the `ModelAdvisor.Group` or `ModelAdvisor.FactoryGroup` classes.
- 2 Register the folder.

Define Where Custom Folders Appear

You can specify the location of custom folders within the Model Advisor using the following guidelines:

- To define a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To define a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Note: To define a new folder in the **By Product** folder, use the `ModelAdvisor.Root.publish` method within a custom check. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

Group Definition

The following examples shows a group definition. The definition places the tasks inside a folder called **My Group** under the **Model Advisor** root. The task definition function includes this group definition.

```
% Custom folder definition
MAG = ModelAdvisor.Group('mathworks.example.ExampleGroup');
MAG.DisplayName='My Group';
% Add tasks to My Group folder
MAG.addTask(MAT1);
MAG.addTask(MAT2);
MAG.addTask(MAT3);
% Add My Group folder to the Model Advisor under 'Model Advisor' (root)
mdladvRoot.publish(MAG);
```

The following example shows a factory group definition function. The definition places the checks into a folder called **Demo Factory Group** inside of the **By Task** folder.

```
function defineModelAdvisorTasks
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='Demo Factory Group';
rec.Description='Demo Factory Group';
rec.addCheck('mathworks.example.configManagement');
rec.addCheck('mathworks.example.unconnectedObjects');
rec.addCheck('mathworks.example.optimizationSettings');
mdladvRoot.publish(rec); % publish inside By Task
```

Customization Example

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

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

The example also provides the source code of the `sl_customization.m` file that executes the customizations.

To run the example:

- 1 At the MATLAB command line, type `slvndemo_mdladv`.
- 2 Follow the instructions in the model.

See Also

[ModelAdvisor.Check](#) | [ModelAdvisor.FactoryGroup](#) | [ModelAdvisor.Group](#) | [ModelAdvisor.Root.publish](#) | [ModelAdvisor.Task](#)

Related Examples

- “Update the Environment to Include Your sl_customization File” on page 29-19

More About

- “Define Custom Checks” on page 28-33
- “Define Startup and Post-Execution Actions Using Process Callback Functions” on page 28-30
- “Display and Enable Checks” on page 28-34
- “Register Checks and Process Callbacks” on page 28-29

Verify and Use Custom Configurations

Update the Environment to Include Your `sl_customization` File

When you start Simulink, it reads customization (`sl_customization.m`) files. If you change the contents of your customization file, update your environment by performing these tasks:

- 1 If you previously started the Model Advisor:
 - a Close the model from which you started the Model Advisor
 - b Clear the data associated with the previous Model Advisor session by removing the `slprj` folder from your code generation folder.
- 2 At the MATLAB command line, enter:

```
sl_refresh_customizations
```

If you have created custom checks, at the MATLAB command line, then also enter:

```
Advisor.Manager.refresh_customizations
```
- 3 Open your model.
- 4 Start the Model Advisor.

Verify Custom Configurations

To verify a custom configuration:

- 1 If you created custom checks, or created the custom configuration using the `sl_customization` method, update the Simulink environment.
- 2 Open a model.
- 3 From the model window, start the Model Advisor.
- 4 Select **Settings > Load Configuration**. If you see a warning that the Model Advisor report corresponds to a different configuration, click **Load** to continue.
- 5 In the Open dialog box, navigate to and select your custom configuration.
- 6 When the Model Advisor reopens, verify that the configuration contains the new folders and checks. For example, the **Review Optimizations** folder and the **Check Simulink optimization settings** and **Check safety-related optimization settings** checks.

7 Optionally, run the checks.

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 29-5

Create Procedural-Based Model Advisor Configurations

Create Procedures

What Is a Procedure?

A procedure is a series of checks. The checks in a procedure depend on passing the previous checks. If Check A is the first check in a procedure and Check B follows, the Model Advisor does not run Check B until Check A passes. Checks A and B can be either custom or provided by MathWorks.

You create procedures with the `ModelAdvisor.Procedure` class API. You first add the checks to tasks, which are wrappers for the checks. The tasks are added to procedures.

When creating procedural checks, be aware of potential conflicts with the checks. Verify that it is possible to pass both checks.

Create Procedures Using the Procedures API

You use the `ModelAdvisor.Procedure` class to create procedural checks.

- 1 Add each check to a task using the `ModelAdvisor.Task.setCheck` method. The task is a wrapper for the check. You cannot add checks directly to procedures.
- 2 Add each task to a procedure using the `ModelAdvisor.Procedure.addTask` method.

Define Procedures

You define procedures in a procedure definition function that specifies the properties of each instance of the `ModelAdvisor.Procedure` class. Define one instance of the procedure class for each procedure that you want to add to the Model Advisor. Then register the procedure using the `ModelAdvisor.Root.register` method.

Add Subprocedures and Tasks to Procedures

You can add subprocedures or tasks to a procedure. The tasks are wrappers for checks.

- Use the `ModelAdvisor.Procedure.addProcedure` method to add a subprocedure to a procedure.
- Use the `ModelAdvisor.Procedure.addTask` method to add a task to a procedure.

Define Where Procedures Appear

You can specify where the Model Advisor places a procedure using the `ModelAdvisor.Group.addProcedure` method.

Procedure Definition

The following code example adds procedures to a group:

```
%Create three procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure3');

%Create a group
MAG = ModelAdvisor.Group('com.mathworks.sample.myGroup');

%Add the three procedures to the group
addProcedure(MAG, MAP1);
addProcedure(MAG, MAP2);
addProcedure(MAG, MAP3);

%register the group and and procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAG);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

The following code example adds subprocedures to a procedure:

```
%Create a procedures
MAP = ModelAdvisor.Procedure('com.mathworks.example.Procedure');

%Create 3 sub procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub1');
MAP2=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub2');
MAP3=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub3');

%Add sub procedures to procedure
addProcedure(MAP, MAP1);
addProcedure(MAP, MAP2);
addProcedure(MAP, MAP3);

%register the procedures
```

```
mdladvRoot = ModelAdvisor.Root;  
mdladvRoot.register(MAP);  
mdladvRoot.register(MAP1);  
mdladvRoot.register(MAP2);  
mdladvRoot.register(MAP3);
```

The following code example adds tasks to a procedure:

```
%Create three tasks  
MAT1=ModelAdvisor.Task('com.mathworks.tasksample.myTask1');  
MAT2=ModelAdvisor.Task('com.mathworks.tasksample.myTask2');  
MAT3=ModelAdvisor.Task('com.mathworks.tasksample.myTask3');  
  
%Create a procedure  
MAP = ModelAdvisor.Procedure('com.mathworks.tasksample.myProcedure');  
  
%Add the three tasks to the procedure  
addTask(MAP, MAT1);  
addTask(MAP, MAT2);  
addTask(MAP, MAT3);  
  
%register the procedure and tasks  
mdladvRoot = ModelAdvisor.Root;  
mdladvRoot.register(MAP);  
mdladvRoot.register(MAT1);  
mdladvRoot.register(MAT2);  
mdladvRoot.register(MAT3);
```

See Also

[ModelAdvisor.Procedure](#) | [ModelAdvisor.Procedure.addProcedure](#)
| [ModelAdvisor.Procedure.addTask](#) | [ModelAdvisor.Root.register](#) |
[ModelAdvisor.Task.setCheck](#)

Related Examples

- “Create a Procedural-Based Configuration” on page 30-5

More About

- “Define Custom Tasks” on page 29-13

Create a Procedural-Based Configuration

In this example, you examine a procedural-based configuration.

- 1 At the MATLAB command line, type `slvndemo_mdadv`.
- 2 In the model window, select **View demo sl_customization.m**. The `sl_customization.m` file opens in the MATLAB Editor window.

The file contains three checks created in the function `defineModelAdvisorChecks`:

- `ModelAdvisor.Check('com.mathworks.sample.Check1')` - Checks Simulink block fonts.
- `ModelAdvisor.Check('com.mathworks.sample.Check2')` - Checks Simulink window screen color.
- `ModelAdvisor.Check('com.mathworks.sample.Check3')` - Checks model optimization settings.

Each check has a set of fix actions.

- 3 In the `sl_customization.m` file, examine the function `defineTaskAdvisor`.

- The `ModelAdvisor.Procedure` class API creates procedures `My Procedure` and `My sub_Procedure`:

```
% Define procedures
MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');
MAP.DisplayName='My Procedure';

MAP_sub = ModelAdvisor.Procedure('com.mathworks.sample.sub_ProcedureSample');
MAP_sub.DisplayName='My sub_Procedure';
```

- The `ModelAdvisor.Task` class API creates tasks `MAT4`, `MAT5`, and `MAT6`. The `ModelAdvisor.Task.setCheck` method adds the checks to the tasks:

```
% Define tasks
MAT4 = ModelAdvisor.Task('com.mathworks.sample.TaskSample4');
MAT4.DisplayName='Check Simulink block font';
MAT4.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT4);

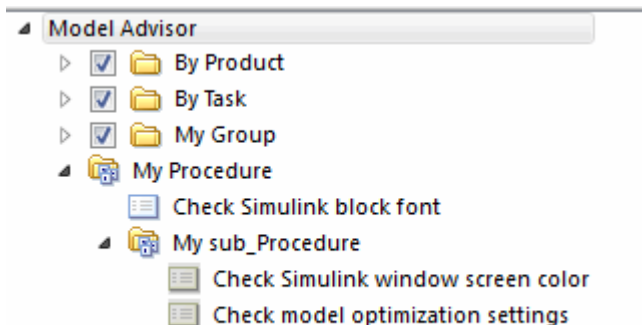
MAT5 = ModelAdvisor.Task('com.mathworks.sample.TaskSample5');
MAT5.DisplayName='Check Simulink window screen color';
MAT5.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT5);
```

```
MAT6 = ModelAdvisor.Task('com.mathworks.sample.TaskSample6');
MAT6.DisplayName='Check model optimization settings';
MAT6.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT6);
```

- The `ModelAdvisor.Procedure.addTask` method adds task MAT4 to My Procedure and tasks MAT5 and MAT6 to My sub_Procedure. The `ModelAdvisor.Procedure.addProcedure` method adds My sub_Procedure to My Procedure:

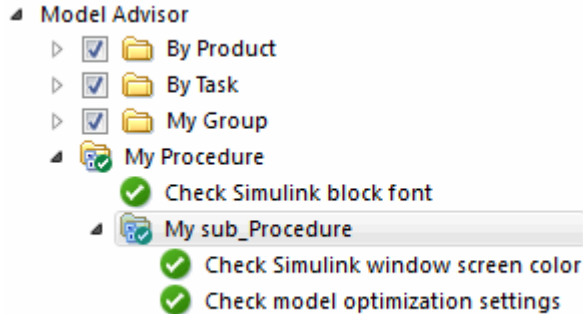
```
% Add tasks to procedures:
% Add Task4 to MAP
MAP.addTask(MAT4);
% Now Add Task5 and Task6 to MAP_sub
MAP_sub.addTask(MAT5);
MAP_sub.addTask(MAT6);
% Include the Sub-Procedure in the Procedure
MAP.addProcedure(MAP_sub);
```

- 4 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 5 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens.
- 6 In the left pane, expand **My Procedure > My sub_Procedure**. The Check Simulink block font check is in the My Procedure folder. My sub_Procedure contains Check Simulink window screen color and Check model optimization settings.



- 7 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run All**. The Model Advisor Check Simulink block font check fails. The Model Advisor does not check the remaining two checks in the My sub_Procedure folder. Running the checks in the My sub_Procedure folder depends on passing the Check Simulink block font check.

- 8 In the **Action** section of the Model Advisor dialog box, click **Fix block fonts**.
- 9 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run All**. The Check Simulink block font check passes. The Model Advisor runs the Check Simulink window screen color check. This check fails and the Model Advisor stops checking.
- 10 In the **Action** section of the Model Advisor dialog box, click **Fix window screen color**.
- 11 In the left pane of the Model Advisor, select My sub_Procedure. In the right pane of the Model Advisor, click **Run All**. The Check Simulink window screen color check passes. The Model Advisor runs the Check model optimization settings check. This check warns.
- 12 In the **Action** section of the Model Advisor dialog box, click **Fix model optimization settings**.
- 13 In the left pane of the Model Advisor, select Check model optimization settings. In the right pane of the Model Advisor, click **Run This Task**. The Check model optimization settings check passes.



See Also

ModelAdvisor.Procedure | ModelAdvisor.Procedure.addProcedure
 | ModelAdvisor.Procedure.addTask | ModelAdvisor.Root.register |
 ModelAdvisor.Task.setCheck

More About

- “Create Procedures” on page 30-2

- “Define Custom Checks” on page 28-33

Deploy Custom Configurations

Overview of Deploying Custom Configurations

About Deploying Custom Configurations

When you create a custom configuration, often you *deploy* the custom configuration to your development group. Deploying the custom configuration allows your development group to review models using the same checks.

After you create a custom configuration, you can use it in the Model Advisor, or deploy the configuration to your users. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

Deploying Custom Configurations Workflow

When you deploy custom configurations, you:

- 1** Optionally author custom checks, as described in “Create Model Advisor Checks”.
- 2** Organize checks and folders to create custom configurations, as described in “Create Custom Configurations” on page 29-2.
- 3** Deploy the custom configuration to your users, as described in “How to Deploy Custom Configurations” on page 31-3.

How to Deploy Custom Configurations

To deploy a custom configuration:

- 1 Determine which files to distribute. You might need to distribute more than one file.

If You...	Using the...	Distribute...
Created custom checks	Customization file	<ul style="list-style-type: none"> • <code>sl_customization.m</code> • Files containing check and action callback functions (if separate)
Organized checks and folders	Model Advisor Configuration Editor	Configuration MAT file
	Customization file	<code>sl_customization.m</code>

- 2 Distribute the files and tell the user to include these files on the MATLAB path.
- 3 Instruct the user to load the custom configuration.

Related Examples

- “Automatically Load and Set the Default Configuration” on page 31-5
- “Manually Load and Set the Default Configuration” on page 31-4

Manually Load and Set the Default Configuration

When you use the Model Advisor, you can load any configuration. Once you load a configuration, you can set it so that the Model Advisor use that configuration every time you open the Model Advisor.

- 1 Open the Model Advisor.
- 2 Select **Settings > Load Configuration**.
- 3 In the Open dialog box, navigate to and select the configuration file that you want to edit.
- 4 Click **Open**.

Simulink reloads the Model Advisor using the new configuration.

- 5 Optionally, when the Model Advisor opens, set the current configuration as the default by selecting **File > Set Current Configuration as Default**.

Related Examples

- “Automatically Load and Set the Default Configuration” on page 31-5
- “Update the Environment to Include Your sl_customization File” on page 29-19

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 29-5
- “Register Checks and Process Callbacks” on page 28-29

Automatically Load and Set the Default Configuration

When you use the Model Advisor, you can automatically set the default configuration by modifying an `sl_customization.m` file.

- 1 Place a configuration MAT file on your MATLAB path.
- 2 Modify your `sl_customization.m` file by adding the function:

```
function [checkCellArray taskCellArray] = ModelAdvisorProcessFunction ...  
    (stage, system, checkCellArray, taskCellArray)  
    switch stage  
        case 'configure'  
            ModelAdvisor.setConfiguration('qeAPIConfigFilePath.mat');  
        end
```

In the function, replace `qeAPIConfigFilePath.mat` with the name of the configuration MAT file in step 1.

- 3 The `sl_customization.m` file is loaded every time you start the Model Advisor, using `qeAPIConfigFilePath.mat` as the default configuration.

Tip You can restore the MathWorks default configuration by selecting **File > Restore Default Configuration**.

Related Examples

- “Manually Load and Set the Default Configuration” on page 31-4
- “Update the Environment to Include Your `sl_customization` File” on page 29-19

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 29-5
- “Register Checks and Process Callbacks” on page 28-29

