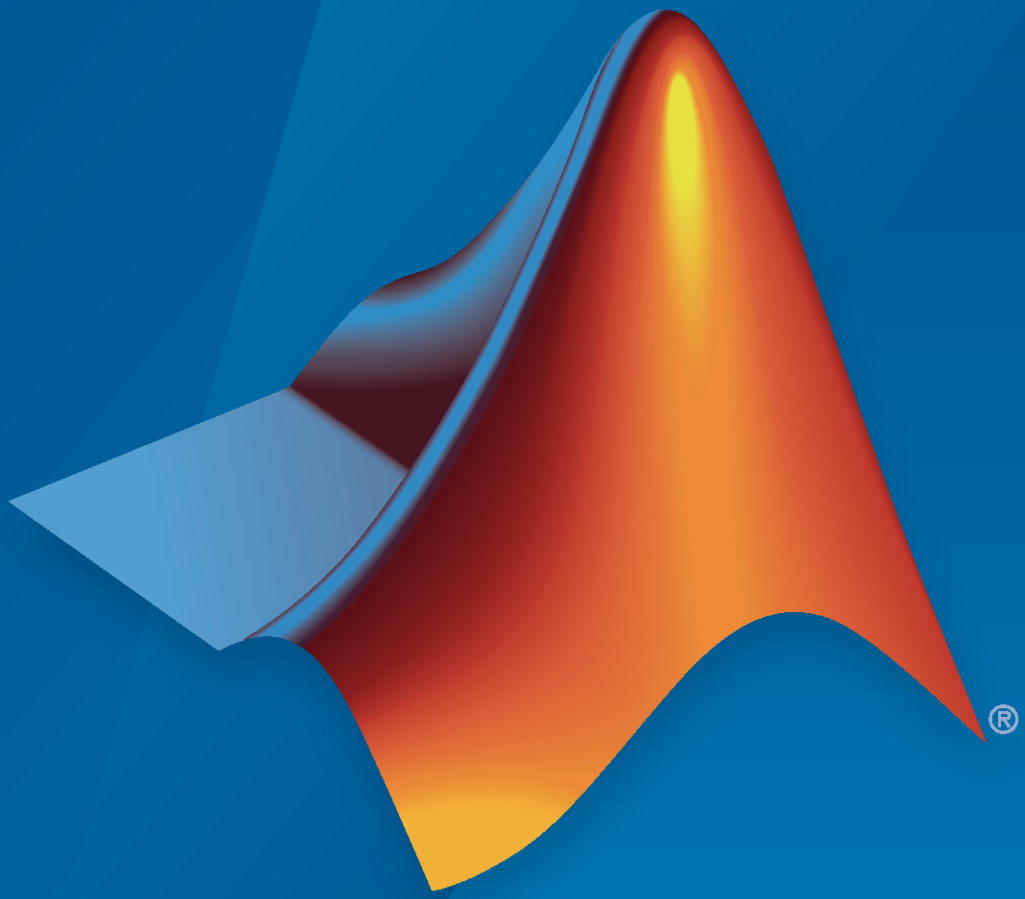
Simulink® Check™

User's Guide



MATLAB® & SIMULINK®

R2020a

MathWorks®

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Check™ User's Guide

© COPYRIGHT 2004–2020 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

September 2017	Online only	New for Version 4.0 (Release 2017b)
March 2018	Online only	Revised for Version 4.1 (Release 2018a)
September 2018	Online only	Revised for Version 4.2 (Release 2018b)
March 2019	Online only	Revised for Version 4.3 (Release 2019a)
September 2019	Online only	Revised for Version 4.4 (Release 2019b)
March 2020	Online only	Revised for Version 4.5 (Release 2020a)

Contents

1 Get Started

Simulink Check Product Description **1-2**
Key Features **1-2**

Assess and Verify Model Quality **1-3**

Detect and Fix Model Advisor Check Violations **1-4**
Detect and Fix Model Advisor Check Violations While You Edit **1-5**
Detect and Fix Model Advisor Check Violations Interactively **1-6**

Collect Model Metric Data by Using the Metrics Dashboard **1-8**
Analyze Metric Data **1-8**
Explore Metric Data **1-9**
Refactor Model Based on Metric Data **1-11**

Detect and Fix Compliance Issues **1-12**
Explore Compliance Results in the Dashboard **1-12**
Update Model to Fix Compliance Issues **1-13**
Rerun Model Metrics **1-14**

Refactor Models to Improve Component Reuse **1-16**
Identify and Replace Clones with Links to Library Blocks **1-16**
Explore Other Options **1-19**

Simplify Model for Targeted Analysis of Complex Models using Model Slicer Tool **1-20**

2 Verification and Validation

Test Model Against Requirements and Report Results **2-2**
Requirements – Test Traceability Overview **2-2**
Display the Requirements **2-2**
Link Requirements to Tests **2-3**
Run the Test **2-4**
Report the Results **2-5**

Analyze a Model for Standards Compliance and Design Errors **2-7**
Standards and Analysis Overview **2-7**
Check Model for Style Guideline Violations and Design Errors **2-7**

Perform Functional Testing and Analyze Test Coverage **2-9**
Incrementally Increase Test Coverage Using Test Case Generation **2-9**

Analyze Code and Test Software-in-the-Loop **2-12**
Code Analysis and Testing Software-in-the-Loop Overview **2-12**
Analyze Code for Defects, Metrics, and MISRA C:2012 . . . . . **2-12**

3 Checking Systems Interactively

Check Model Compliance by Using the Model Advisor **3-2**
Model Advisor Overview **3-2**
Run Model Advisor Checks and Review Results **3-4**
Check Your Model by Using Edit Time Checks **3-6**
View and Customize the Edit-Time Checks in a Model Advisor Configuration **3-8**

Exclude Blocks From the Model Advisor Check Analysis **3-10**
Model Advisor Exclusion Overview **3-10**
Save Model Advisor Exclusions in a Model File **3-11**
Save Model Advisor Exclusions in Exclusion File **3-11**
Create Model Advisor Exclusions **3-11**
Review Model Advisor Exclusions **3-12**
Manage Exclusions **3-13**
Exclude Blocks from Edit Time Checking **3-14**
Limit Model Checks by Excluding Gain and Outport Blocks **3-15**

Generate Model Advisor Reports **3-19**
Generate Results Report When Executing Model Advisor Checks **3-19**
Generate Results Report After Executing Model Advisor Checks **3-19**
Modify Template for Model Advisor Check Results Report **3-19**

Transform Model to Variant System **3-22**
Example Model **3-22**
Perform Variant Transform on Example Model **3-23**
Model Transformation Limitations **3-25**

Enable Component Reuse by Using Clone Detection **3-26**
Exact Clones and Similar Clones **3-26**
Identify Exact and Similar Clones **3-26**
Replace Clones **3-30**
Identifying and Replacing Clones in Model Libraries **3-31**
Check the Equivalency of the Model **3-31**

Improve Model Readability by Eliminating Local Data Store Blocks . . . **3-33**
Example Model **3-33**
Replace Data Store Blocks **3-34**
Limitations **3-37**

Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks **3-38**
Example Model **3-38**

Merge Prelookup Operation 3-39
Conditions and Limitations 3-41

Model Checks for DO-178C/DO-331 Standard Compliance 3-43
Model Checks for High Integrity Systems Modeling 3-44

Model Checks for DO-254 Standard Compliance 3-50
Model Checks for High Integrity Systems Modeling 3-50
HDL Code Advisor Checks 3-53

Model Checks for MAB and JMAAB Compliance 3-56
Accessing the MAB and JMAAB Model Advisor Checks 3-56
Modeling Guidelines and Model Advisor Checks for MAB and JMAAB . . . 3-56

Model Checks for High Integrity Systems Modeling 3-66
High Integrity Systems Modeling Checks 3-44

Model Checks for IEC 61508, IEC 62304, ISO 26262, and EN 50128 Standard Compliance 3-72
Model Checks for High Integrity Systems Modeling 3-73

Model Checks for MISRA C:2012 Compliance 3-79

Model Checks for Secure Coding (CERT C, CWE, and ISO/IEC TS 17961 Standards) 3-80

Model Checks for Requirements Links 3-81

4 Check Systems Programmatically

Checking Systems Programmatically 4-2

Create a Function for Checking Multiple Systems 4-3

Create a Function for Checking Multiple Systems in Parallel 4-5

Archive and View Results 4-7
Archive Results 4-7
View Results in Command Window 4-7
View Results in Model Advisor Command-Line Summary Report 4-7
View Results in Model Advisor GUI 4-8
View Model Advisor Report 4-8

Archive and View Model Advisor Run Results 4-10

5 Model Metrics

Collect and Explore Metric Data by Using the Metrics Dashboard **5-2**
Metrics Dashboard Widgets **5-3**
Size **5-4**
Modeling Guideline Compliance **5-5**
Architecture **5-6**
Metric Thresholds **5-7**
Dashboard Limitations **5-8**

Collect Model Metrics Using the Model Advisor **5-9**

Create a Custom Model Metric for Nonvirtual Block Count **5-11**

Collect Model Metrics Programmatically **5-15**

Model Metric Data Aggregation **5-18**
How Model Metric Aggregation Works **5-18**
Access Aggregated Metric Data **5-19**

Identify Modeling Clones with the Metrics Dashboard **5-22**

Collect Compliance Data and Explore Results in the Model Advisor . . . **5-24**

Collect Metric Data Programmatically and View Data Through the Metrics Dashboard **5-29**

Fix Metric Threshold Violations in a Continuous Integration Systems Workflow **5-33**
Project Setup **5-33**
GitLab Setup **5-35**
Jenkins Setup **5-35**
Continuous Integration Workflow **5-36**

Customize Metrics Dashboard Layout and Functionality **5-39**

Compare Model Complexity and Code Complexity Metrics **5-49**
Metric Threshold Values **5-49**
Comparing Code and Model Complexity Metric Results **5-49**

6 Create Model Advisor Checks

Overview of the Customization File for Custom Checks **6-2**

Common Utilities for Creating Checks **6-4**

Create Pass/Fail and Informational Model Advisor Checks **6-5**
Create an sl_customization Function **6-5**

Create the Check Definition Function for a Pass/Fail Check with No Fix Action 6-5
Create the Check Definition Function for an Informational Check 6-6
Run the Custom Checks in the Model Advisor 6-8

Create a Pass/Fail Model Advisor Check with Fix Action 6-9
Create the sl_customization File 6-9
Create the Check Definition File 6-9
Run the Check 6-13

Create Model Advisor Check for Model Configuration Parameters 6-16
Create a Data File for a Configuration Parameter Check 6-16
Create Check for Diagnostics Pane Model Configuration Parameters . . . 6-18
Data File for Configuration Parameter Check 6-20

Define Model Advisor Checks for Supported or Unsupported Blocks and Parameters 6-27
Example 6-27
Create Block Parameter Constraints 6-27
Create Model Advisor Checks from Constraints 6-29

Define Startup and Post-Execution Actions Using Process Callback Functions 6-32
Process Callback Function Arguments 6-32
Process Callback Function 6-32
Tips for Using the Process Callback Function in a sl_customization File . 6-33

Defining Custom Model Advisor Checks Workflow 6-34
Create the sl_customization File and Function 6-34
Register Custom Checks 6-35
Create a Check Definition Function 6-35

Define the Compile Option for Custom Model Advisor Checks 6-39
Checks for Models That Are Not Compiled by the Model Advisor 6-39
Checks That Require the Model to be Compiled and Simulated by the Model Advisor 6-39
Checks That Evaluate Code Generation Readiness of the Model 6-40
Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model 6-42

Exclude Blocks From Custom Checks 6-47

7 Model Advisor Customization

Customize the Configuration of the Model Advisor Overview 7-2

Use the Model Advisor Configuration Editor to Customize the Model Advisor 7-3
Overview of the Model Advisor Configuration Editor 7-3
Open the Model Advisor Configuration Editor 7-4
Specify a Default Configuration File 7-5

Customize the Model Advisor Configuration 7-5
Suppress Warning Message for Missing Checks 7-7
Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration 7-7

Programmatically Customize Tasks and Folders for the Model Advisor 7-15
Customization File Overview 7-15
Register Tasks and Folders 7-15
Define Custom Tasks 7-16
Define Custom Folders 7-17

Programmatically Create Procedural-Based Configurations 7-19
Create Procedural-Based Configurations 7-19

Update the Environment to Include Your Custom Configuration 7-22

Load and Verify a Custom Configuration 7-23

Deploy Custom Configurations 7-24

Create and Deploy a Model Advisor Custom Configuration 7-25

8 Model Slicer

Highlight Functional Dependencies 8-2

Highlight Dependencies for Multiple Instance Reference Models 8-8

Refine Highlighted Model 8-12
Define a Simulation Time Window 8-12
Exclude Blocks 8-16
Exclude Inputs of a Switch Block 8-19

Refine Dead Logic for Dependency Analysis 8-22
Analyze the Dead Logic 8-22

Create a Simplified Standalone Model 8-28

Highlight Active Time Intervals by Using Activity-Based Time Slicing . 8-29
Highlighting the Active Time Intervals of a Stateflow State or Transition 8-29
Activity-Based Time Slicing Limitations and Considerations 8-35
Stateflow State and Transition Activity 8-35

Simplify a Standalone Model by Inlining Content 8-36

Workflow for Dependency Analysis 8-38
Dependency Analysis Workflow 8-38
Dependency Analysis Objectives 8-38

Configure Model Highlight and Sliced Models **8-40**
Model Slice Manager **8-40**
Model Slicer Options **8-40**
Storage Options **8-40**
Refresh Highlighting Automatically **8-41**
Sliced Model Options **8-41**
Trivial Subsystems **8-41**
Inline Content Options **8-42**

Model Slicer Considerations and Limitations **8-43**
Model Compilation **8-43**
Model Highlighting and Model Editing **8-43**
Standalone Sliced Model Generation **8-43**
Sliced Model Considerations **8-43**
Port Attribute Considerations **8-44**
Simulation Time Window Considerations **8-45**
Simulation-based Sliced Model Simplifications **8-45**
Starting Points Not Supported **8-46**
Model Slicer Support Limitations for Simulink Software Features **8-46**
Model Slicer Support Limitations for Simulation Stepper **8-46**
Model Slicer Support Limitations for Simulink Blocks **8-46**
Model Slicer Support Limitations for Stateflow **8-47**

Using Model Slicer with Stateflow **8-49**
Model Slicer Highlighting Behavior for Stateflow Elements **8-49**
Using Model Slicer with Stateflow State Transition Tables **8-49**
Support Limitations for Using Model Slicer with Stateflow **8-49**

Isolating Dependencies of an Actuator Subsystem **8-51**
Choose Starting Points and Direction **8-51**
View Precedents and Generate Model Slice **8-52**

Isolate Model Components for Functional Testing **8-55**
Isolate Subsystems for Functional Testing **8-55**
Isolate Referenced Model for Functional Testing **8-57**

Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results **8-63**

Simplification of Variant Systems **8-65**
Use the Variant Reducer to Simplify Variant Systems **8-65**
Use Model Slicer to Simplify Variant Systems **8-65**

Programmatically Resolve Unexpected Behavior in a Model with Model Slicer **8-66**

Programmatically Resolve Unexpected Behavior in a Model with Model Slicer **8-88**
Prerequisites **8-88**
Find the Area of the Model Responsible for Unexpected Behavior **8-88**
Isolate the Area of the Model Responsible for Unexpected Behavior **8-90**
Investigate the Sliced Model and Debug the Source Model **8-94**
Clean Up **8-97**

Refine Highlighted Model Slice by Using Model Slicer Data Inspector . **8-99**
Investigate Highlighted Model Slice by Using Model Slicer Data Inspector **8-99**

Debug Slice Simulation by Using Fast Restart Mode **8-106**
Simulate and Debug a Test Case in a Model Slice **8-106**

Isolate Referenced Model for Functional Testing **8-113**

Analyze the Dead Logic **8-117**

Investigate Highlighted Model Slice by Using Model Slicer Data Inspector **8-122**

Get Started

- “Simulink Check Product Description” on page 1-2
- “Assess and Verify Model Quality” on page 1-3
- “Detect and Fix Model Advisor Check Violations” on page 1-4
- “Collect Model Metric Data by Using the Metrics Dashboard” on page 1-8
- “Detect and Fix Compliance Issues” on page 1-12
- “Refactor Models to Improve Component Reuse” on page 1-16
- “Simplify Model for Targeted Analysis of Complex Models using Model Slicer Tool” on page 1-20

Simulink Check Product Description

Verify compliance with style guidelines and modeling standards

Simulink Check provides industry-recognized checks and metrics that identify standard and guideline violations during development. Supported high-integrity software development standards include DO-178, ISO 26262, IEC 61508, IEC 62304, and MathWorks Advisory Board (MAB) Style Guidelines. Edit-time checks identify compliance issues as you edit. You can create custom checks to comply with your own standards or guidelines.

Simulink Check provides metrics such as size and complexity that you can use to evaluate your model's architecture and compliance to standards. A consolidated metrics dashboard lets you assess design status and quality. Automatic model refactoring lets you replace duplicate design elements, reduce design complexity, and identify reusable content. The Model Slicer tool isolates problematic behavior in a model and generates a simplified model for debugging.

Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178).

Key Features

- Edit-time checking to identify model guideline violations
- Compliance checking for MAB style guidelines and high-integrity system design guidelines (DO-178, ISO 26262, IEC 61508, IEC 62304)
- Compliance checking for secure coding standards (CERT C, CWE, ISO/IEC TS 17961)
- Custom check authoring with Model Advisor Configuration Editor
- Metrics for computing model size, complexity, and readability
- Dashboard providing consolidated view of metrics and project status
- Model refactoring with clone detection and model transformations

Assess and Verify Model Quality

With the Simulink Check product, you can use industry-recognized checks and metrics that identify standard and guideline violations. Supported high-integrity software development standards include the DO-178, ISO 26262, IEC 61508, IEC 62304, and MathWorks Advisory Board (MAB) style guidelines. Use edit-time checking, to identify compliance issues as you develop your model. And, when you are done editing, to assess whether your model complies with size, architecture, and compliance requirements, run the Metrics Dashboard. The Metrics Dashboard contains widgets that visualize the metric data. To obtain detailed results, drill in to the data.

From the Metrics Dashboard, you can fix compliance issues by launching the Model Advisor. To determine whether you can automatically refactor a model to increase component reuse, launch the Clone Detector.

Functions and classes are available for customizing the Metrics Dashboard and Model Advisor. For example, you can write your custom checks and use the Model Advisor Configuration editor to create a custom configuration. Use the Metrics Dashboard functions and classes to configure the compliance metric widgets to point to the custom configuration.

In this tutorial, you will learn to:

1. Address Model Advisor compliance issues.
2. Run the Metrics Dashboard to obtain and analyze metric data.
3. Address MAB and High Integrity check violations from the Metrics Dashboard.
4. Refactor a model to improve component reuse by launching the Clone Detector app from the Metrics Dashboard.

At the end of the tutorial, there are links to topics that provide more information.

To start the tutorial, see “Detect and Fix Model Advisor Check Violations” on page 1-4.

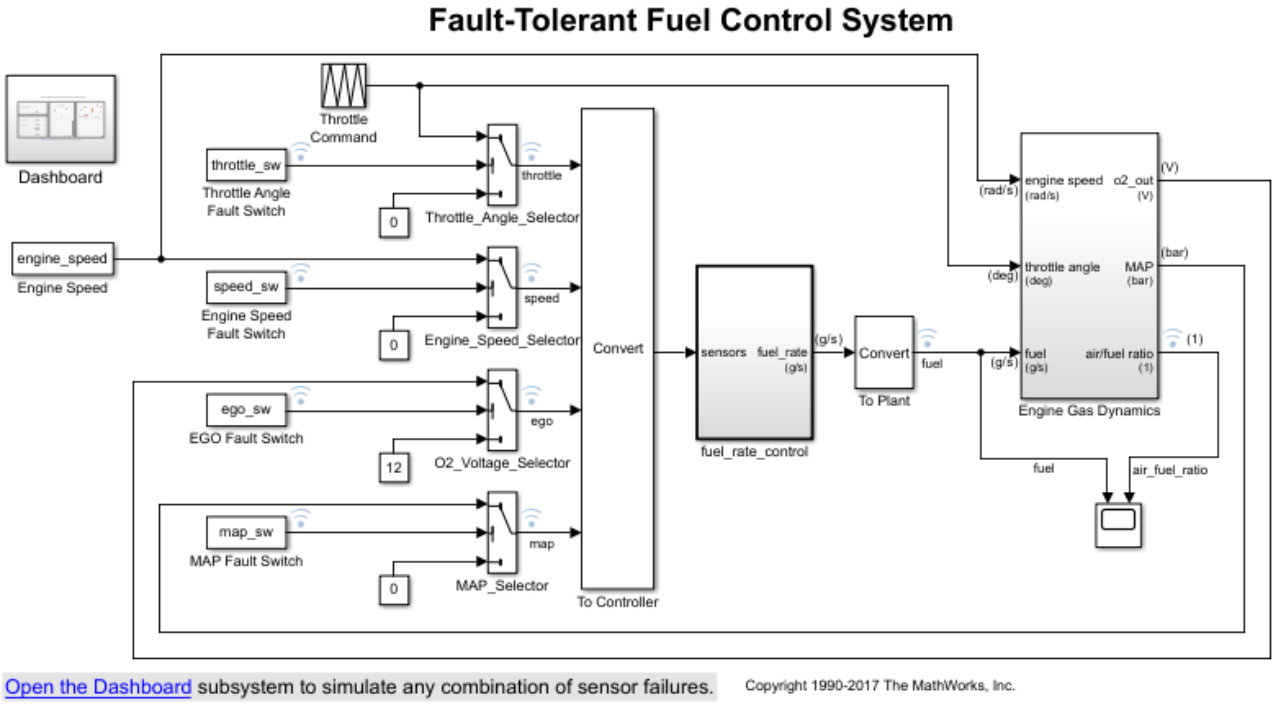
Detect and Fix Model Advisor Check Violations

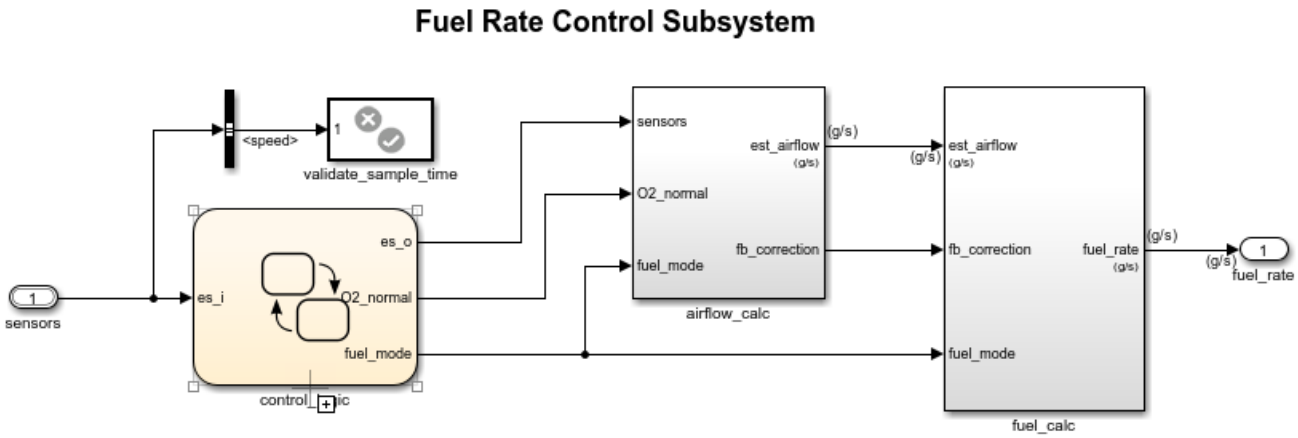
The Model Advisor checks your model or subsystem for modeling conditions and configuration settings that cause inaccurate or inefficient simulation and inefficient generated code and code that is unsuitable for safety-critical applications. The Model Advisor checks can help you verify compliance with industry standards and guidelines. By using the Model Advisor, you can implement consistent modeling guidelines across projects and development teams.

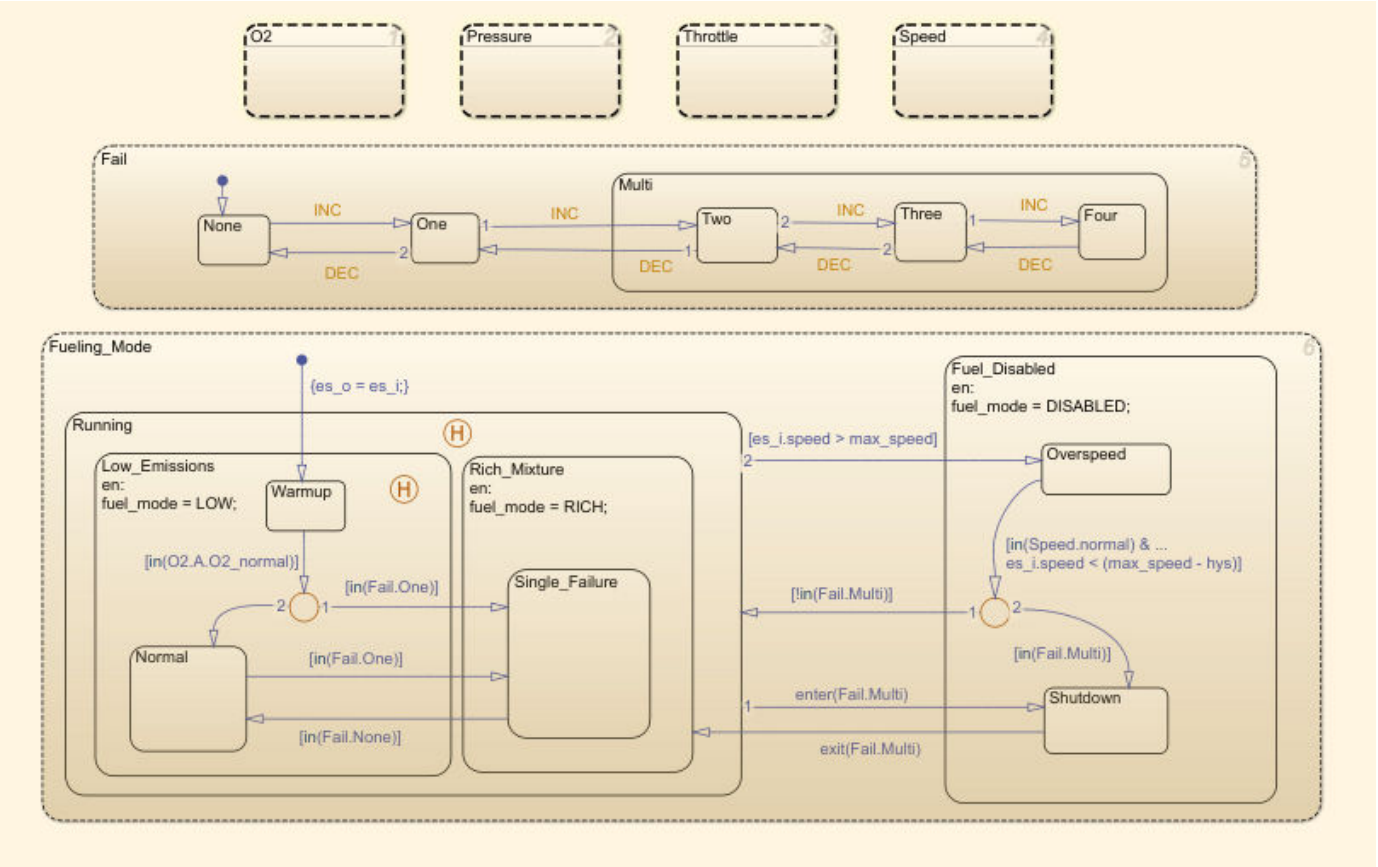
A subset of Model Advisor checks support edit-time checking. With edit-time checking, you can check for model conditions while you develop a model. Highlighted blocks in the model editor window alert you to issues in the model.

This tutorial uses the example model `sldemo_fuelsys`. This model is an air-fuel ratio control system designed with Simulink and Stateflow.

The figures show portions of the `sldemo_fuelsys` model. The top-level model is a closed-loop system that consists of a plant (Engine Gas Dynamics) and a controller (the Fuel Rate Control subsystem). The plant allows engineers to validate the controller through simulation early in the design cycle. The control logic is a Stateflow chart that specifies the different modes of operation.







Detect and Fix Model Advisor Check Violations While You Edit

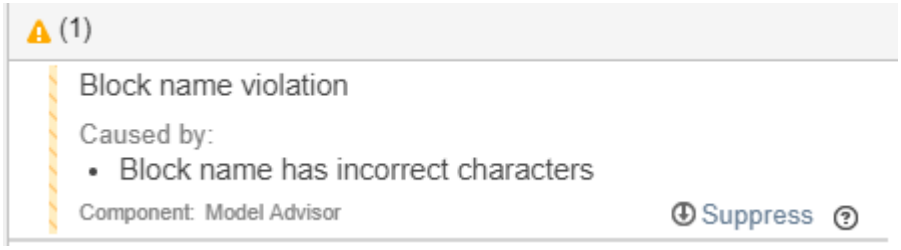
1. Set your current folder to a writeable directory.
2. Open the model `sldemo_fuelsys` by typing this command:

   ```
   open_system('sldemo_fuelsys')
   ```

3 To use edit-time checking, on the **Modeling** tab, select **Model Advisor > Edit-Time Checks**.

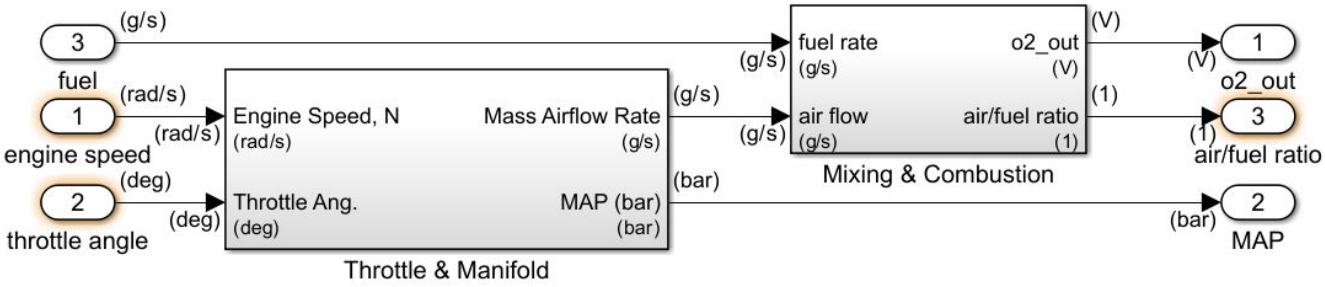
The highlighted blocks and subsystems indicate compliance issues.

4 Pause over a highlighted block and click the warning icon. A dialog box provides a description of the warning. For detailed documentation on the check that detected the issue, click the question mark. These blocks contain edit-time warnings because of incorrect block names.



To exclude a block from a selected check, you can click **Suppress**.

5 Open the `Engine Gas Dynamics` subsystem by double-clicking it. Pause over the `air/fuel ratio` output port and click the warning icon.



This output port returns warnings because its name violates two checks: “Check character usage in block names” and “Check port block names”.

6 Address the warnings by replacing the `/` symbol and the space in the block name with underscores. The block is no longer highlighted.

7 Address the warnings for the other highlighted blocks in the `Engine Gas Dynamics` subsystem.

Detect and Fix Model Advisor Check Violations Interactively

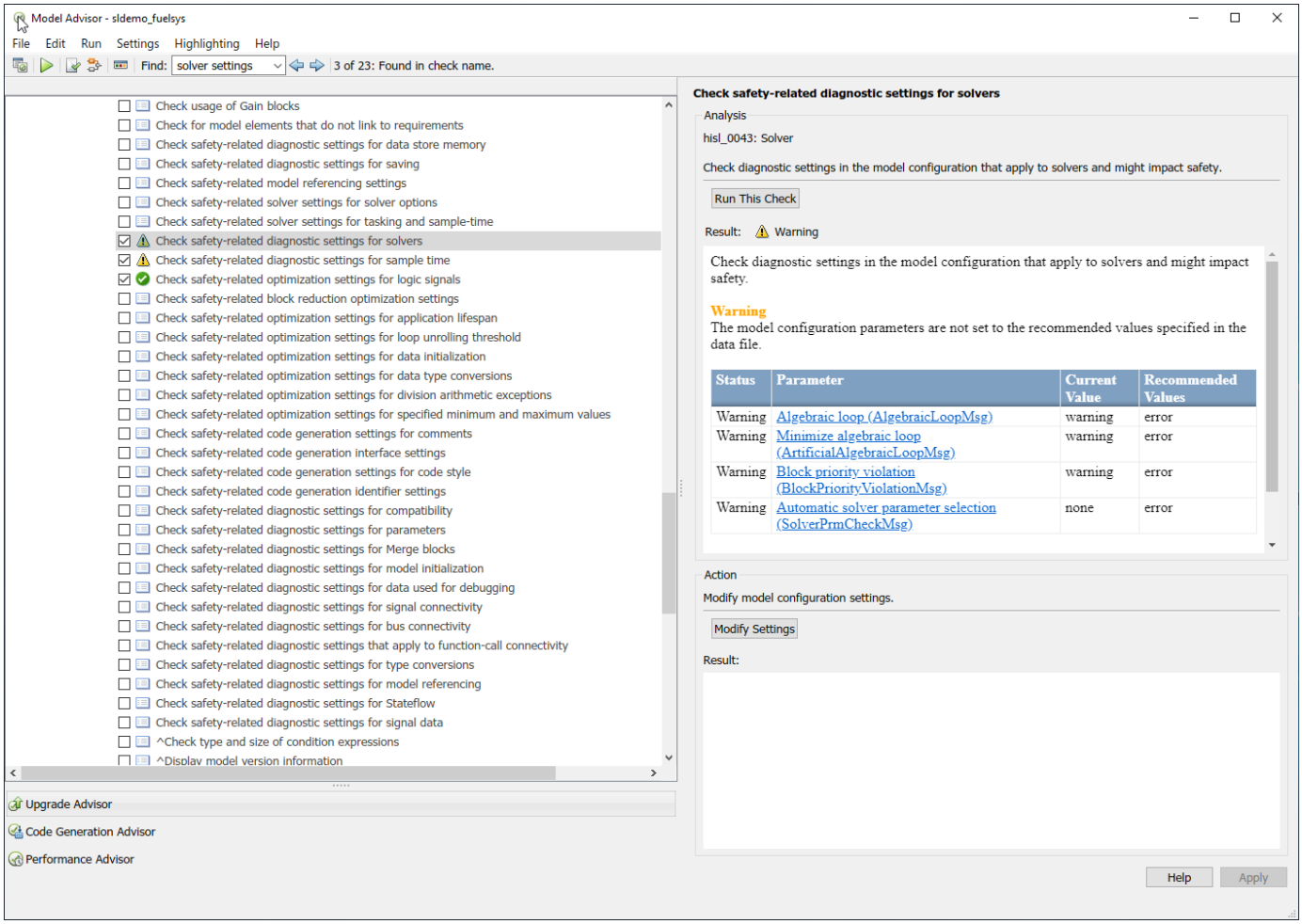
1 On the **Modeling** tab, select **Model Advisor**.

2 Select the top-level model `sldemo_fuelsys` from the System Hierarchy and click **OK**.

3 In the left pane, in the **By Product > Simulink Check > Model Standards > DO-178C/DO-331** folder, select:

- **Check safety-related diagnostic settings for solvers**
- **Check safety-related diagnostic settings for sample time**
- **Check safety-related optimization settings for logic signals**

4 Right-click **DO-178C/DO-331 Checks** node, and then select **Run Selected Checks**.



5. To review the configuration parameters that are not set to the recommended values, click **Check safety-related optimization settings for solvers**.
6. To update the parameters to the recommended values, in the right pane, click **Modify Settings**.

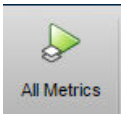
   The Model Advisor updates the parameters to the recommended values and details the result.
7. Repeat step 6 for the **Check safety-related diagnostic settings for sample time** check.
8. To verify that your model now passes, rerun the checks.
9. To generate a results report of the Simulink Check checks, select the **DO-178C/DO-331 Checks** node, and then, in the right pane click **Generate Report**.
10. Close the Model Advisor.

Next, collect metric data on the model and fix other compliance issues by using the Metrics Dashboard.

Collect Model Metric Data by Using the Metrics Dashboard

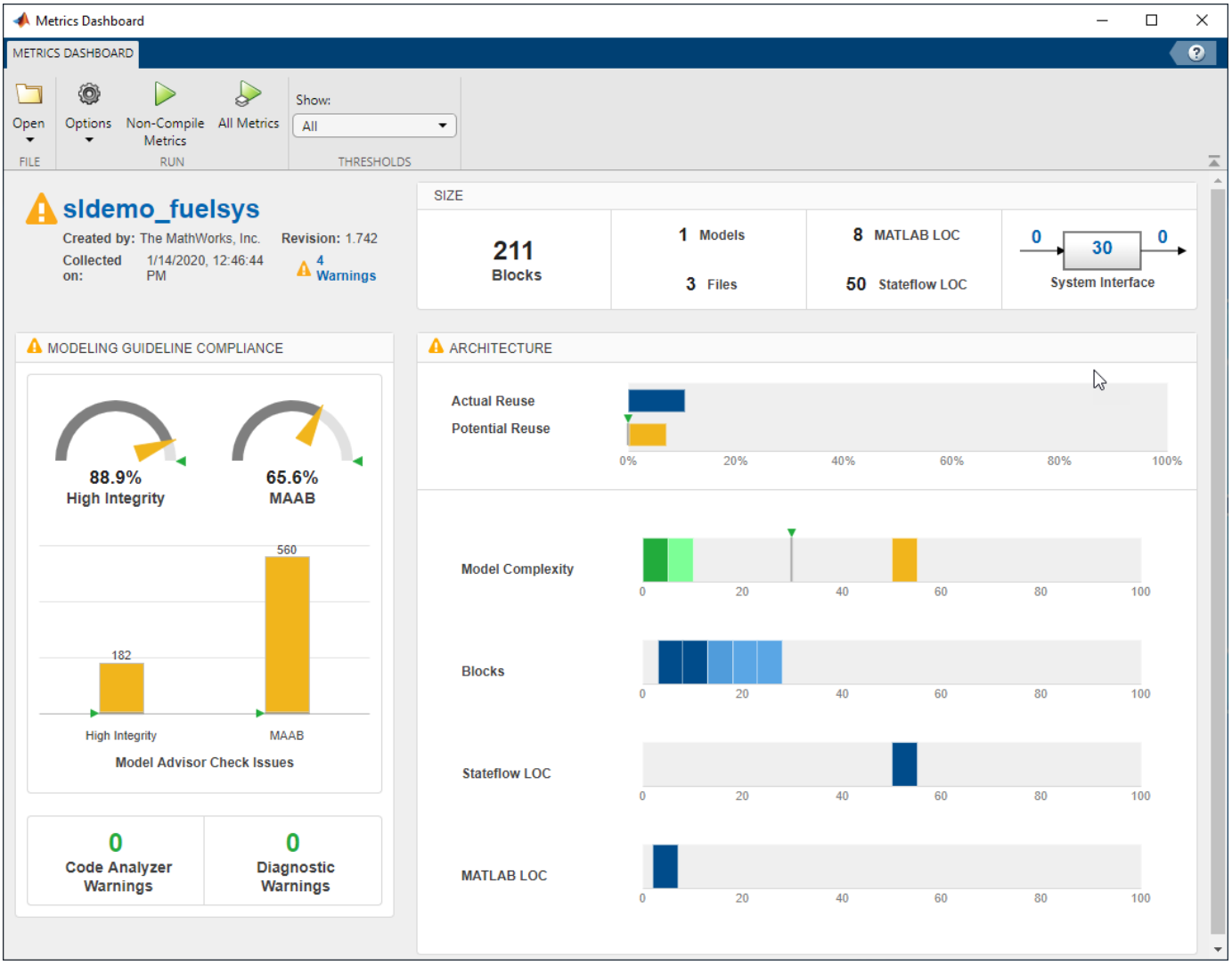
To collect model metric data and assess the design status and quality of your model, use the Metrics Dashboard. The Metrics Dashboard provides a view into the size, architecture, and guideline compliance of your model.

1. Return to the top level of the `sldemo_fuelsys` model.
2. On the **Apps** tab, open the Metrics Dashboard by clicking **Metrics Dashboard**.
3. To collect metric data for this model, click the **All Metrics** icon.



Analyze Metric Data

The Metrics Dashboard contains widgets that visualize metric data in these categories: size, modeling guideline compliance, and architecture. By default, some widgets contain metric threshold values. These values specify whether your metric data is compliant (which appears green in the widget) or produces a warning (which appears yellow in the widget). Metrics that do not have threshold values appear blue. Functions and classes are available for specifying noncompliant ranges and for changing the threshold values.

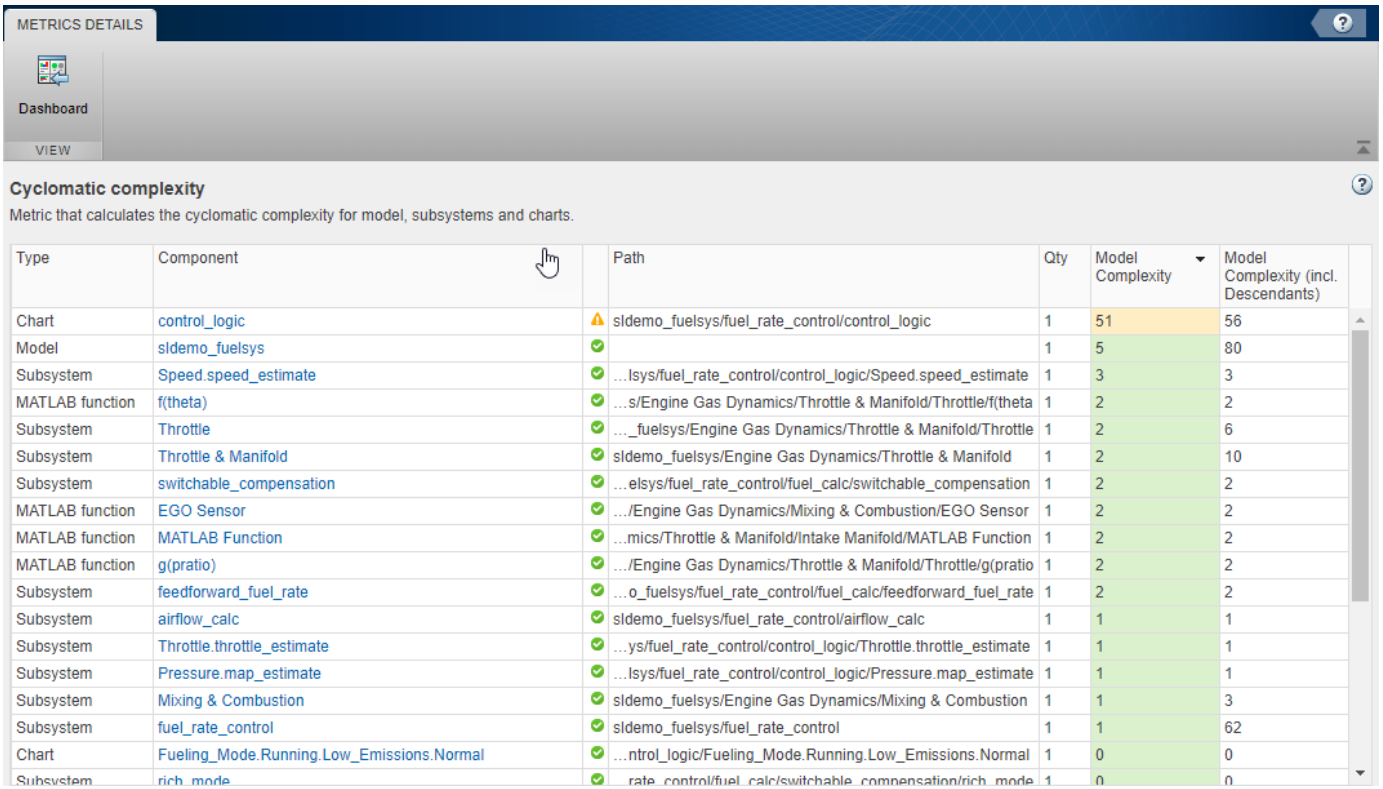


In the **Architecture** section of the dashboard, locate the **Model Complexity** widget. To view tooltips, pause over each vertical bar. This widget is a visual representation of the distribution of complexity across the components in the model hierarchy. For each complexity range, a colored bar indicates the number of components that fall within that range. Darker green colors indicate more components. In this case, several components have a cyclomatic complexity value in the lowest range, while just one component has a higher complexity. This component has a cyclomatic complexity above 30. Components with cyclomatic complexity above 30 return warnings. For more information, see “Cyclomatic complexity metric”

Explore Metric Data

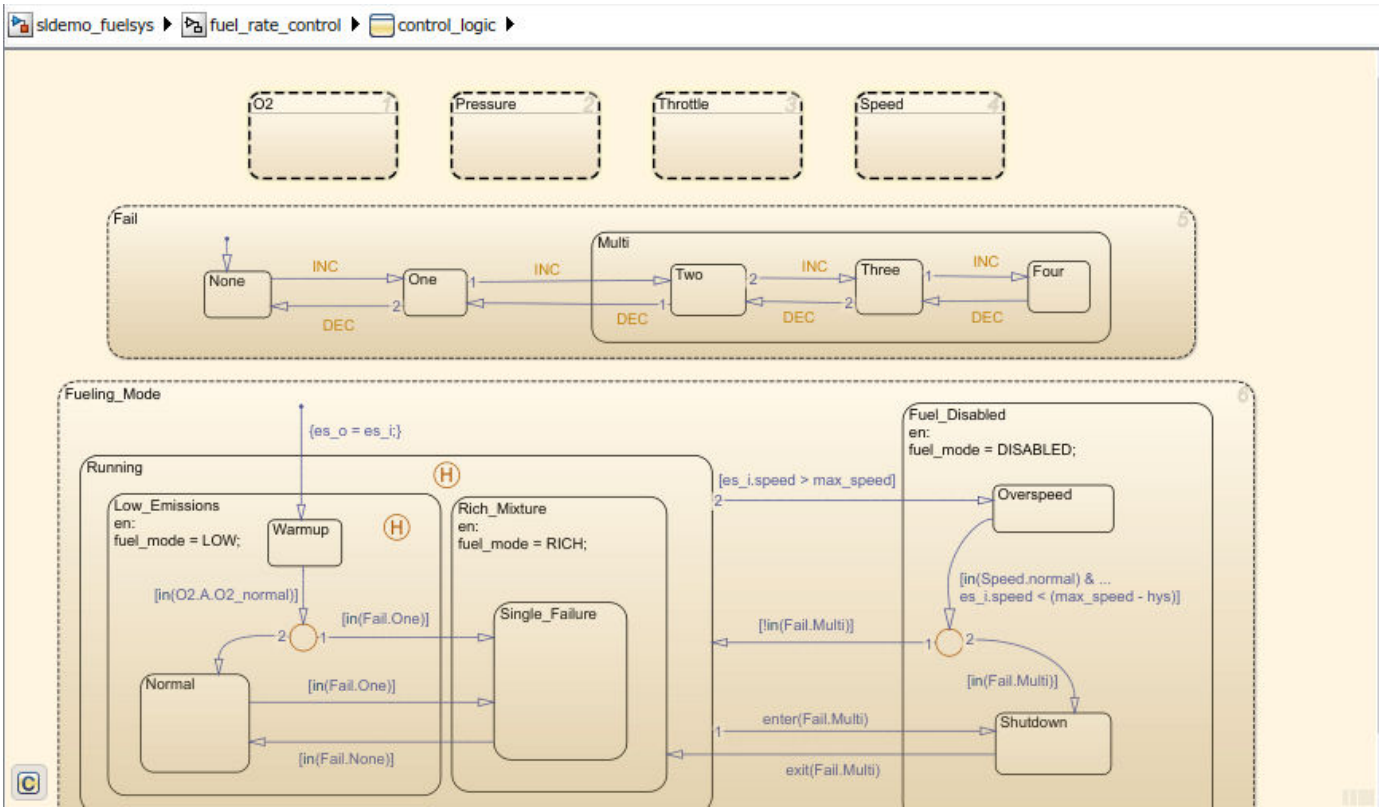
To explore metric data in more detail, click an individual metric widget. For your selected metric, a table displays the value, aggregated value, and measures (if applicable) at the model component level. From the table, the dashboard provides traceability and hyperlinks to the data source so that you can get detailed results.

To analyze the model complexity details at the model, subsystem, and chart level, click a bar in the **Model Complexity** widget. In this example, the `control_logic` chart has a cyclomatic complexity value of 51, which is yellow because it is in the warning range.



Type	Component		Path	Qty	Model Complexity	Model Complexity (incl. Descendants)
Chart	control_logic	⚠	sldemo_fuelsys/fuel_rate_control/control_logic	1	51	56
Model	sldemo_fuelsys	✓		1	5	80
Subsystem	Speed.speed_estimate	✓	...lsys/fuel_rate_control/control_logic/Speed.speed_estimate	1	3	3
MATLAB function	f(theta)	✓	...s/Engine Gas Dynamics/Throttle & Manifold/Throttle/f(theta	1	2	2
Subsystem	Throttle	✓	..._fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	2	6
Subsystem	Throttle & Manifold	✓	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	2	10
Subsystem	switchable_compensation	✓	...elsys/fuel_rate_control/fuel_calc/switchable_compensation	1	2	2
MATLAB function	EGO Sensor	✓	.../Engine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	2	2
MATLAB function	MATLAB Function	✓	...mics/Throttle & Manifold/Intake Manifold/MATLAB Function	1	2	2
MATLAB function	g(pratio)	✓	.../Engine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio	1	2	2
Subsystem	feedforward_fuel_rate	✓	...o_fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate	1	2	2
Subsystem	airflow_calc	✓	sldemo_fuelsys/fuel_rate_control/airflow_calc	1	1	1
Subsystem	Throttle.throttle_estimate	✓	...ys/fuel_rate_control/control_logic/Throttle.throttle_estimate	1	1	1
Subsystem	Pressure.map_estimate	✓	...lsys/fuel_rate_control/control_logic/Pressure.map_estimate	1	1	1
Subsystem	Mixing & Combustion	✓	sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	1	3
Subsystem	fuel_rate_control	✓	sldemo_fuelsys/fuel_rate_control	1	1	62
Chart	Fueling_Mode.Running.Low_Emissions.Normal	✓	...ntrol_logic/Fueling_Mode.Running.Low_Emissions.Normal	1	0	0
Subsystem	rich_mode	✓	...rate_control/fuel_calc/switchable_compensation/rich_mode	1	0	0

To see this component in the model, click the `control_logic` hyperlink.



Refactor Model Based on Metric Data

Once you have used the dashboard to determine which components you must modify to meet quality standards, you can refactor your model. For example, you could refactor the `control_logic` chart by moving the logic into atomic subcharts to reduce the complexity for that component.

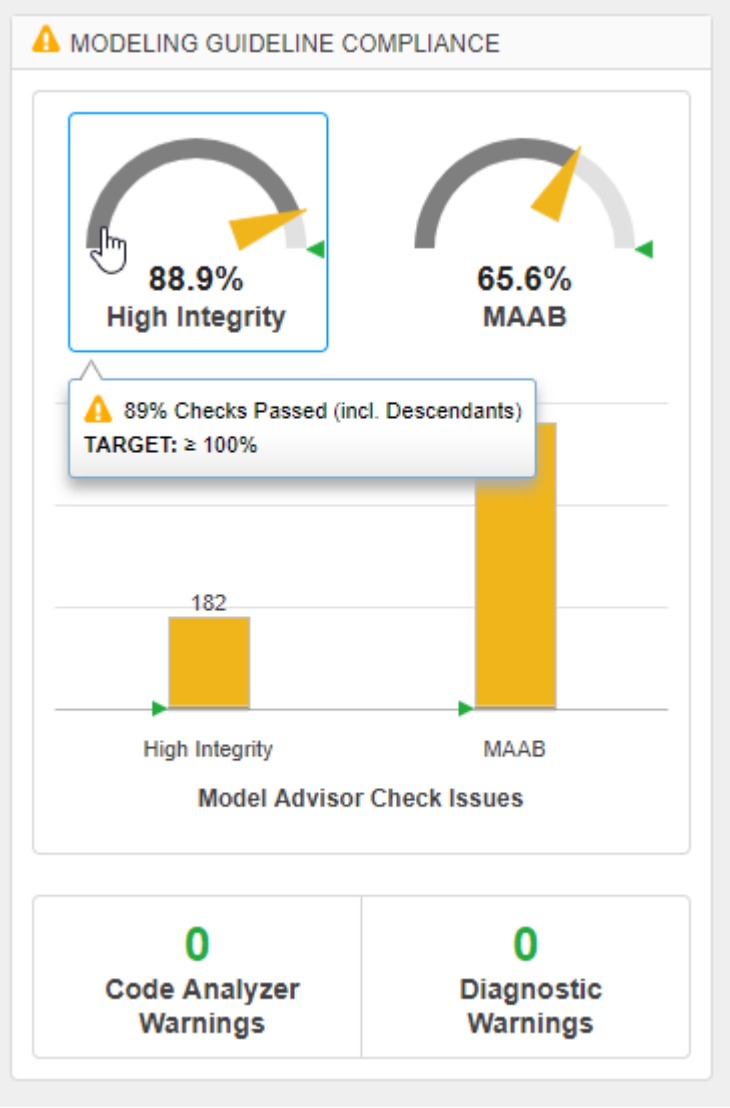
Next, you will use the **Modeling Guideline Compliance** widgets to fix issues associated with high-integrity Model Advisor checks.

Detect and Fix Compliance Issues

When you collect metric data, the Metrics Dashboard runs the MAB and high-integrity Model Advisor checks in the background. The **Modeling Guideline Compliance** section of the dashboard provides the percentages of checks that pass and a count of check warnings and errors. You can interactively investigate these fixes by toggling between the dashboard and your model.

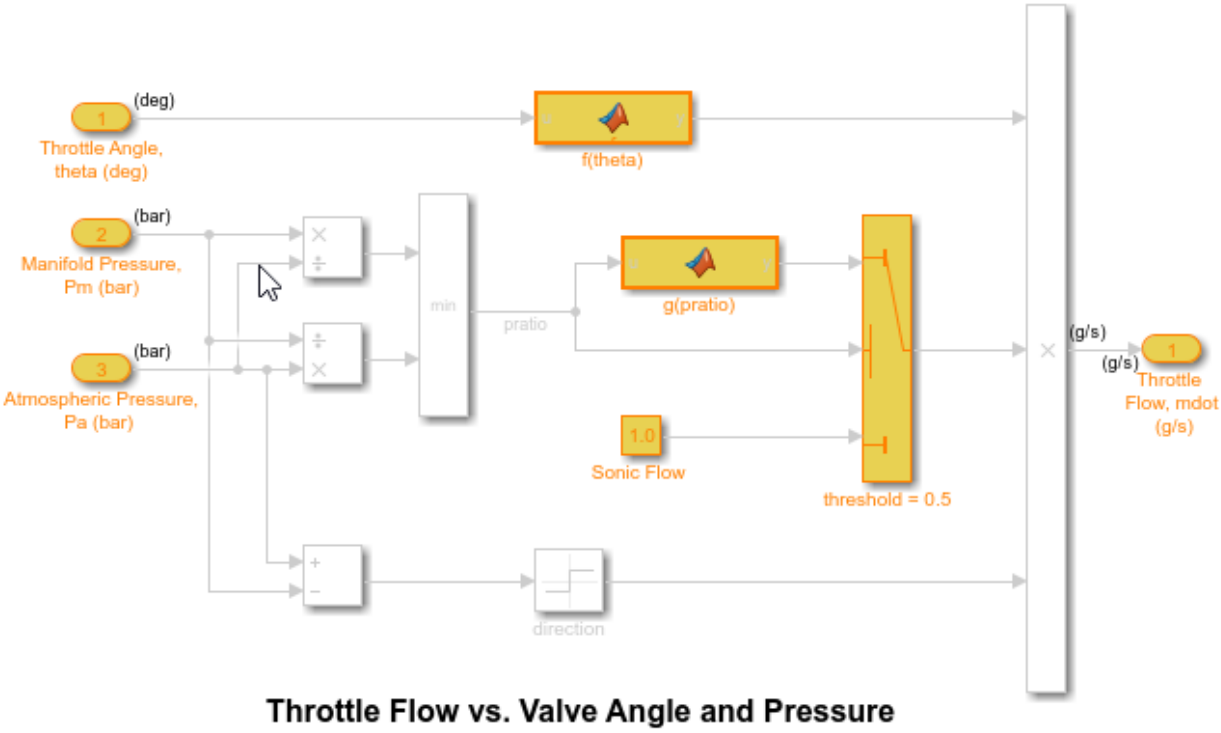
Explore Compliance Results in the Dashboard

1 In the Metrics Dashboard, locate the **Modeling Guideline Compliance** section. This section displays the percentage of high-integrity and MAB compliance checks that pass on all systems. The bar charts show the number of issues reported by the checks in each check group.



2 To see a table that details the number of compliance issues by component, click on the **High Integrity** bar chart. For more information on this metric, see “Model Advisor Check Issues for High-Integrity Systems”.

3 From the table, click the `Throttle` component hyperlink. The `Throttle` component opens in the model editor. The model editor highlights blocks in the component that have compliance issues. The Model Advisor Highlighting dialog box lists checks that do not highlight results.



4 In the Metrics Dashboard, return to the main dashboard page by clicking the **Dashboard** icon.

5 Click the **High Integrity** percentage gauge.

The **Grid** view enables you to identify patterns in results. The grid contains a row for each model component and a column for each check. To see check and component names, hover over a table element.

6 To see the status for each compliance check, click the **Table** view.

7 Expand the `sldemo_fuelsys` node.

8 To explore check results in more detail, click the **Check safety-related diagnostic settings for model referencing** hyperlink.

9 In the Model Advisor Highlighting dialog box, click **Check safety-related diagnostic settings for model referencing** hyperlink.

A Model Advisor report opens. The report lists current model configuration settings and their recommended values.

Update Model to Fix Compliance Issues

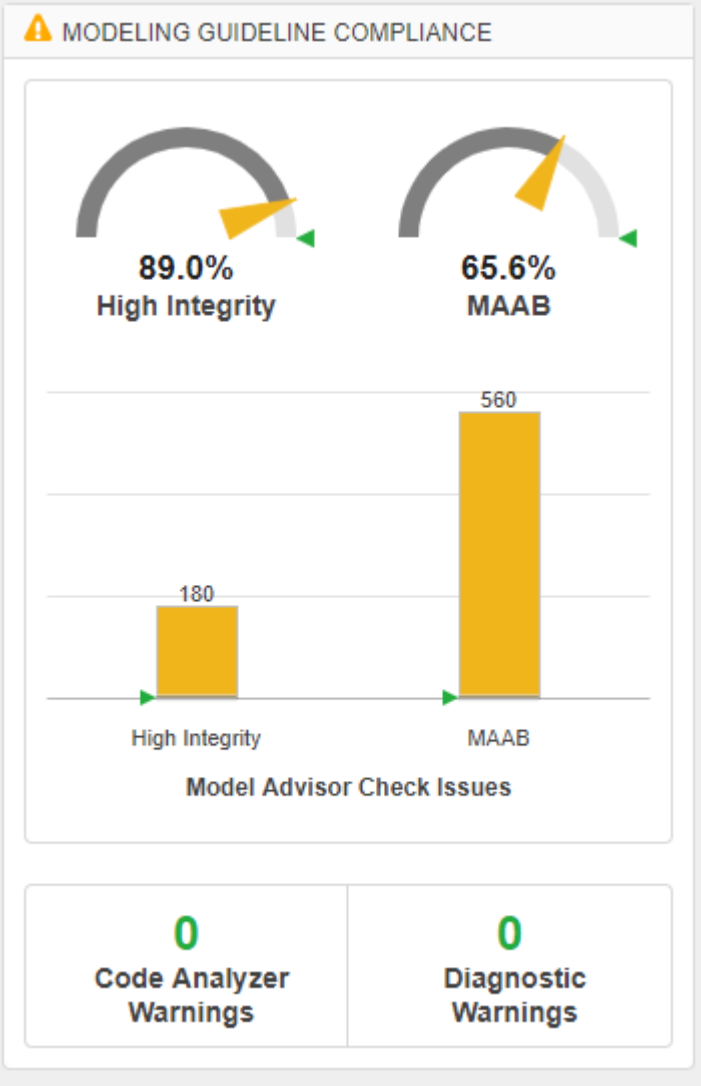
1 To change a parameter's current value to the recommended value, click a parameter. The Configuration Parameters dialog box opens.

2. Change the parameter setting to what the **Recommended Value** column in the report indicates.
3. Click **Apply** and close the dialog box.
4. Close the Model Advisor report.
5. In the Model Advisor Highlighting dialog box, click the **Check safety-related diagnostic settings from compatibility** check.

   The Model Advisor report opens.
6. For this check, repeat steps 1 through 4.
7. Close the Model Advisor Highlighting dialog box and return to the Metrics Dashboard table

Rerun Model Metrics

1. In the Metrics Dashboard table, return to the main dashboard page, by clicking **Dashboard**.
2. To rerun the model metrics, click **All Metrics**.
3. Confirm that the number of **High Integrity** check issues has reduced and the compliance percentage has increased.



Next, use the **Actual Reuse** and **Potential Reuse** widgets to investigate and replace clones across a model hierarchy.

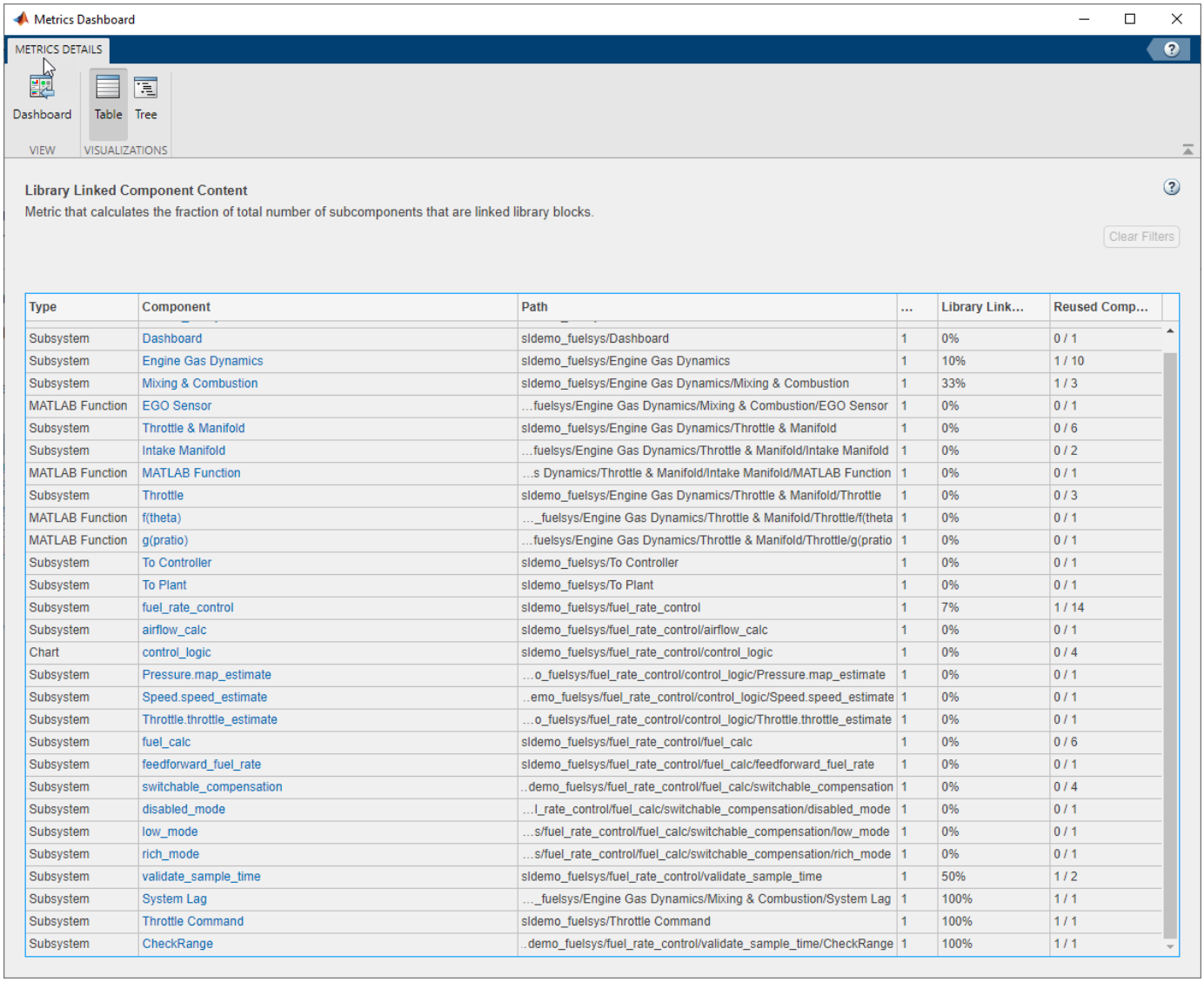
Refactor Models to Improve Component Reuse

You can use the Metrics Dashboard to identify clones across a model hierarchy. Clones are identical MATLAB Function blocks, identical Stateflow charts, and subsystems that have identical block types and connections. Clones can have different parameter settings and values. To replace clones with links to library blocks, you can open the Clone Detector app from the Metrics Dashboard.

Use the Clone Detector app to refactor a model, improve model componentization and readability, and reuse components within a model. In this example, you launch the Clone Detector from the Metrics Dashboard. However, you can also open it by opening the **Apps** tab and clicking **Clone Detector**.

Identify and Replace Clones with Links to Library Blocks

1. In the **Architecture** section, the blue bar in the Actual Reuse widget indicates the fraction of total number of subcomponents that are linked library blocks. Pause over the **Actual Reuse** widget to see more information. For this model, 10% of the total number of subcomponents are linked library blocks.
2. To see more details, click the blue bar. `System Lag`, `Throttle Command`, and `CheckRange` are linked library blocks.



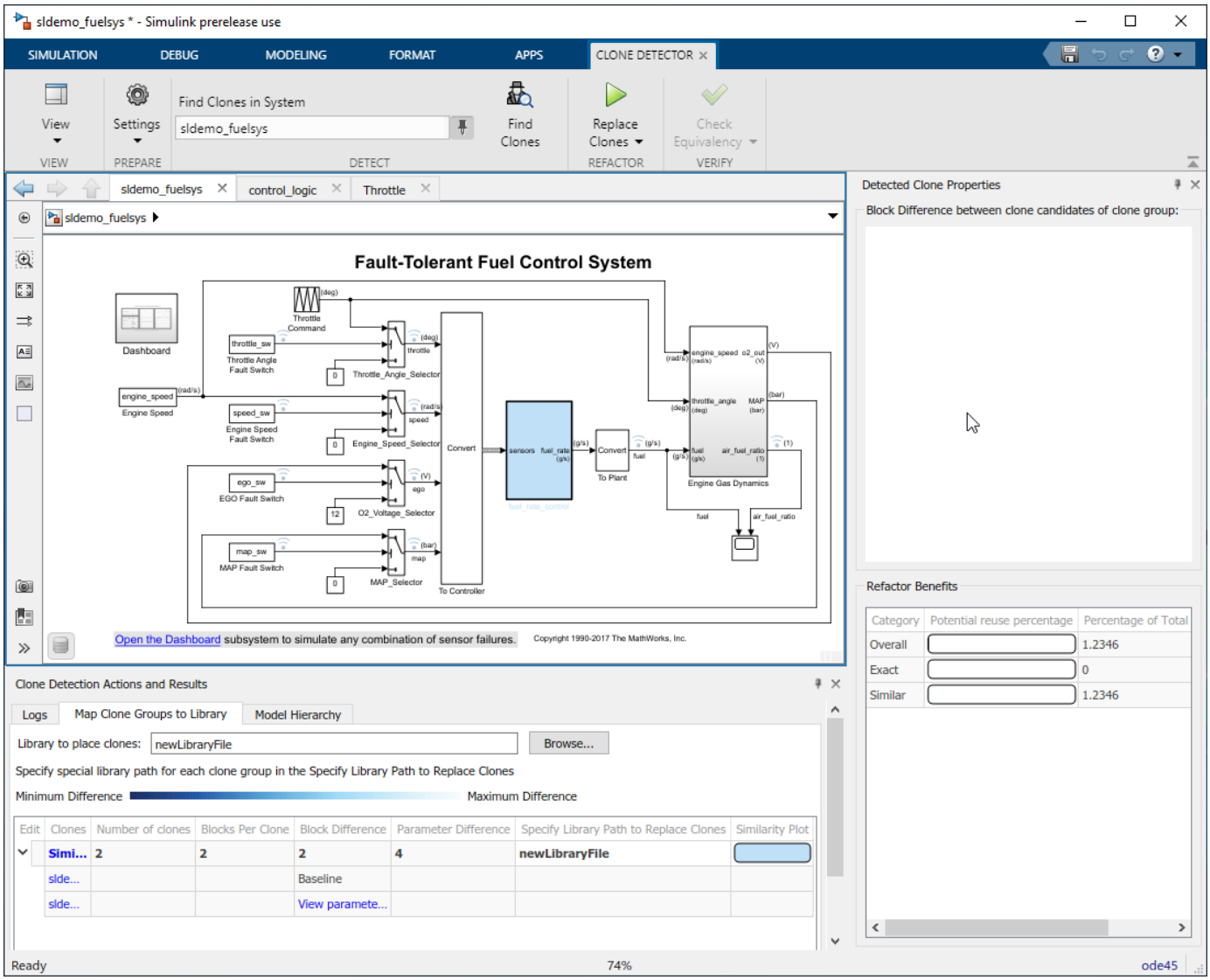
Type	Component	Path	...	Library Link...	Reused Comp...
Subsystem	Dashboard	sldemo_fuelsys/Dashboard	1	0%	0 / 1
Subsystem	Engine Gas Dynamics	sldemo_fuelsys/Engine Gas Dynamics	1	10%	1 / 10
Subsystem	Mixing & Combustion	sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	33%	1 / 3
MATLAB Function	EGO Sensor	...fuelsys/Engine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	0%	0 / 1
Subsystem	Throttle & Manifold	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	0%	0 / 6
Subsystem	Intake Manifold	...fuelsys/Engine Gas Dynamics/Throttle & Manifold/Intake Manifold	1	0%	0 / 2
MATLAB Function	MATLAB Function	...s Dynamics/Throttle & Manifold/Intake Manifold/MATLAB Function	1	0%	0 / 1
Subsystem	Throttle	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	0%	0 / 3
MATLAB Function	f(theta)	..._fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle/f(theta	1	0%	0 / 1
MATLAB Function	g(pratio)	...fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio	1	0%	0 / 1
Subsystem	To Controller	sldemo_fuelsys/To Controller	1	0%	0 / 1
Subsystem	To Plant	sldemo_fuelsys/To Plant	1	0%	0 / 1
Subsystem	fuel_rate_control	sldemo_fuelsys/fuel_rate_control	1	7%	1 / 14
Subsystem	airflow_calc	sldemo_fuelsys/fuel_rate_control/airflow_calc	1	0%	0 / 1
Chart	control_logic	sldemo_fuelsys/fuel_rate_control/control_logic	1	0%	0 / 4
Subsystem	Pressure.map_estimate	...o_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate	1	0%	0 / 1
Subsystem	Speed.speed_estimate	..emo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate	1	0%	0 / 1
Subsystem	Throttle.throttle_estimate	...o_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate	1	0%	0 / 1
Subsystem	fuel_calc	sldemo_fuelsys/fuel_rate_control/fuel_calc	1	0%	0 / 6
Subsystem	feedforward_fuel_rate	sldemo_fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate	1	0%	0 / 1
Subsystem	switchable_compensation	..demo_fuelsys/fuel_rate_control/fuel_calc/switchable_compensation	1	0%	0 / 4
Subsystem	disabled_mode	...l_rate_control/fuel_calc/switchable_compensation/disabled_mode	1	0%	0 / 1
Subsystem	low_mode	...s/fuel_rate_control/fuel_calc/switchable_compensation/low_mode	1	0%	0 / 1
Subsystem	rich_mode	...s/fuel_rate_control/fuel_calc/switchable_compensation/rich_mode	1	0%	0 / 1
Subsystem	validate_sample_time	sldemo_fuelsys/fuel_rate_control/validate_sample_time	1	50%	1 / 2
Subsystem	System Lag	..._fuelsys/Engine Gas Dynamics/Mixing & Combustion/System Lag	1	100%	1 / 1
Subsystem	Throttle Command	sldemo_fuelsys/Throttle Command	1	100%	1 / 1
Subsystem	CheckRange	..demo_fuelsys/fuel_rate_control/validate_sample_time/CheckRange	1	100%	1 / 1

3. Return to the main dashboard page.
4. In the **Architecture** section, the **Potential Reuse** bar indicates that the model contains clones. Pause over **Potential Reuse**. For this model, 7% of the subcomponents are clones.
5. To see more details, click the yellow bar. `Pressure.map_estimate` and `Throttle.throttle_estimate` are clones of each other.
6. To determine whether these clones are candidates for replacement with linked library blocks, click **Open Conversion Tool**.

   The **Clone Detector** app opens as a new tab in the model.
7. In the **Clone Detection Actions and Results** pane, click the **Map Clone Groups to Library** tab.

   There is one clone group. The light blue shading indicates that these clones are similar clones and not exact clones. Similar clones have different parameter settings and values.
8. Expand the clone group.

This clone group consists of two subcharts.



9 To determine parameter differences, in the **Block Difference** column, click **View parameter difference**.

The subcharts in this clone group call Simulink functions that differ only by the value of the breakpoints parameters in the Lookup Table blocks inside of them.

10 In the **Map Clone Groups to Library** tab, for the **Library to place clones** parameter, use the **Browse** button to choose a library or specify a new library name. If you specify a new library name, the app creates the library.

11 Save the model to your working folder and, in the **Clone Detector** tab, click **Replace Clones**. The app replaces similar clones with links to masked library subsystems, if possible.

In the **Logs** tab, click the latest log.

The log contains a message indicating that the clones cannot be replaced with linked library blocks because the data in the Simulink Functions can not be promoted to subchart data.

12 Close the Metrics Dashboard and the model.

When the Clone Detector app refactors a model to replace clones with links to library blocks, the app creates a backup folder. The backup folder name has the prefix `m2m_<model name>`. If you have a Simulink Test™ license, you can verify the equivalency of the refactored and original models by clicking **Check Equivalency** in the **Clone Detector** tab.

Explore Other Options

This table contains a list of common tasks that you can address with Simulink Check.

Task	Reference
Simplify and debug complex models.	“Highlight Functional Dependencies” on page 8-2
Run Model Advisor checks for compliance with safety standards associated with High-Integrity System Modeling and MAB Control Algorithm Modeling guidelines.	“Check Model Compliance by Using the Model Advisor” on page 3-2
Write custom Model Advisor checks.	“Defining Custom Model Advisor Checks Workflow” on page 6-34
Create and deploy a custom Model Advisor configuration.	“Create and Deploy a Model Advisor Custom Configuration” on page 7-25 and “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3
Learn more about using the Metrics Dashboard to collect and view metric data for quality assessment.	“Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2
Configure compliance metrics, add metric thresholds, and customize the Metrics Dashboard layout.	“Customize Metrics Dashboard Layout and Functionality” on page 5-39
Use the Model Transformer tool and the Clone Detector app to refactor a model to improve model componentization and readability and enable reuse.	“Transform Model to Variant System” on page 3-22 and “Enable Component Reuse by Using Clone Detection” on page 3-26
Learn more about how using Simulink products to test models and code, check for design errors, check against standards, measure coverage, and validate the system.	“Verification and Validation”

Simplify Model for Targeted Analysis of Complex Models using Model Slicer Tool

You can simplify simulation, debugging, and formal analysis of large, complex models by focusing on areas of interest in your model. After highlighting a portion of your model using the Model Slicer, you can generate a simplified standalone model. The simplified model contains the blocks and dependency paths in the highlighted portion. Apply changes to the simplified standalone model based on simulation, debugging, and formal analysis, and then apply these changes back to the original model.

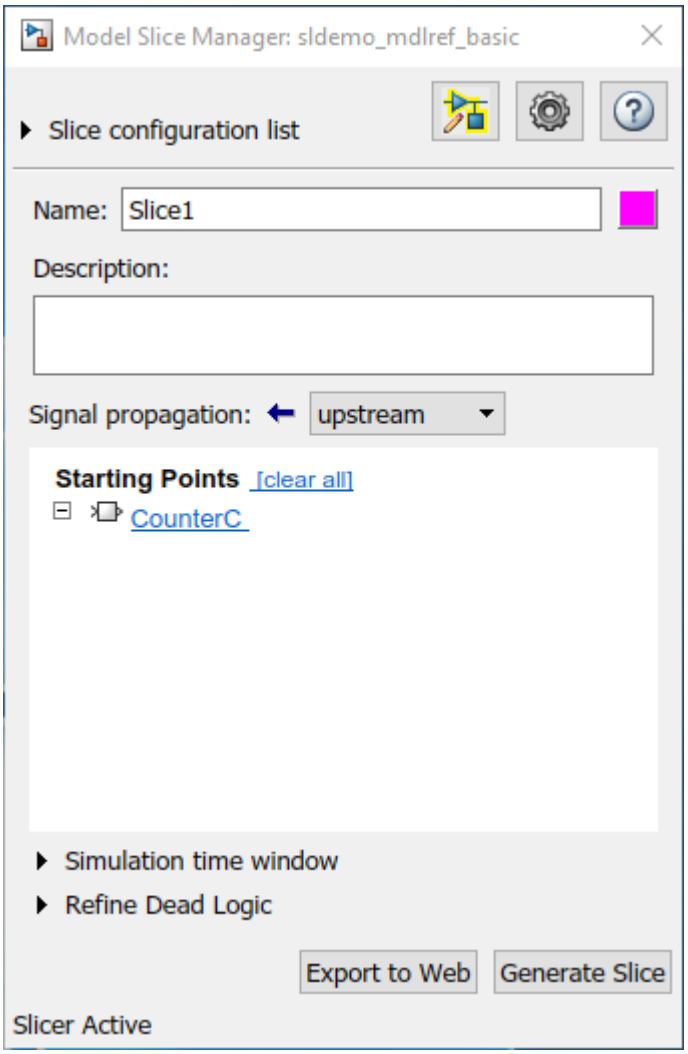
1. The example model `sldemo_mdlref_basic` contains three instances of the model `sldemo_mdlref_counter`. To open the model, at the MATLAB command prompt, enter:

   ```
   open_system('sldemo_mdlref_basic')
   ```

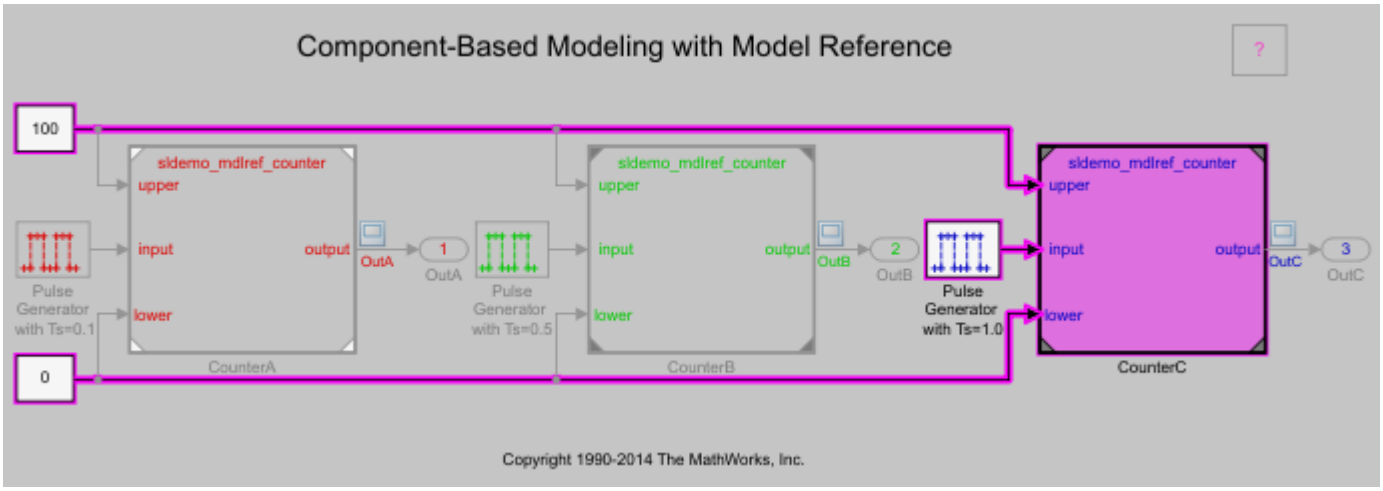
2. To open the Model Slicer Manager, on the Model Verification, Validation, and Test section of the **Apps** tab, click **Model Slicer**.
3. In the Model Slice Manager, click the arrow to expand the **Slicer configuration list**.
4. Set the slice properties:

   - **Name**: `Slice1`
   - **Color**: (magenta)
   - **Signal Propagation**: `upstream`

   Model Slicer can also highlight the constructs downstream of or bidirectionally from a block in your model, depending on which direction you want to trace the signal propagation.
5. Add `CounterC` as a starting point. In the model, right-click `CounterC` and select **Model Slicer > Add as Starting Point**.

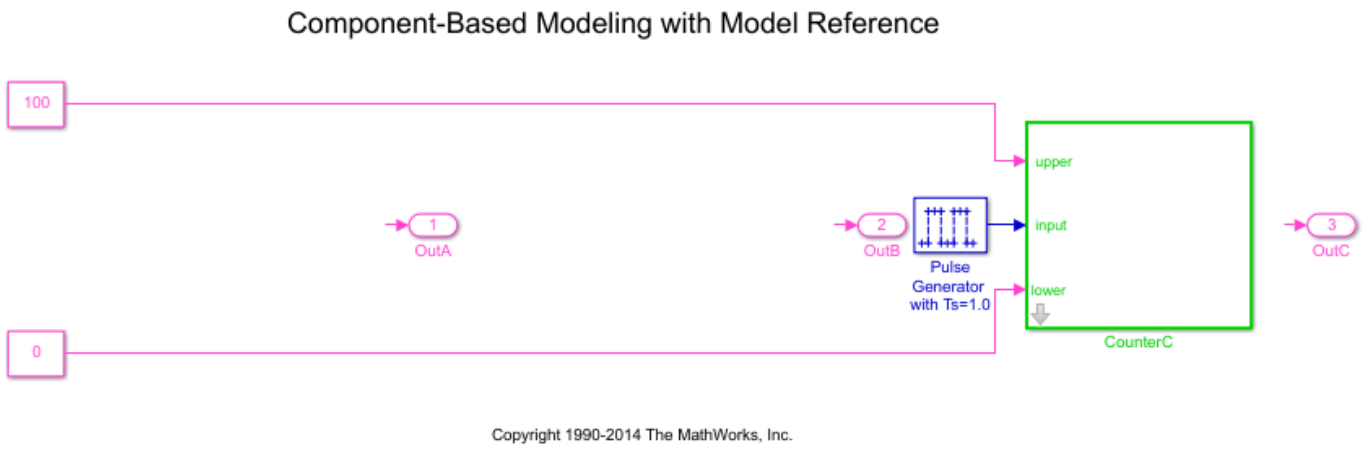


The Model Slicer now highlights the upstream constructs that affect `CounterC`.



6 In the Model Slice Manager, click **Generate slice**.

7 In the **Select File to Write** dialog box, select the save location and enter a model name. The simplified standalone model contains the highlighted model items.



8 To remove highlighting from the model, close the Model Slice Manager.

You can now analyze the simplified standalone model and apply any changes to the source model.

See Also

More About

- “Programmatically Resolve Unexpected Behavior in a Model with Model Slicer” on page 8-88
- “Model Slicer Considerations and Limitations” on page 8-43
- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12

Verification and Validation

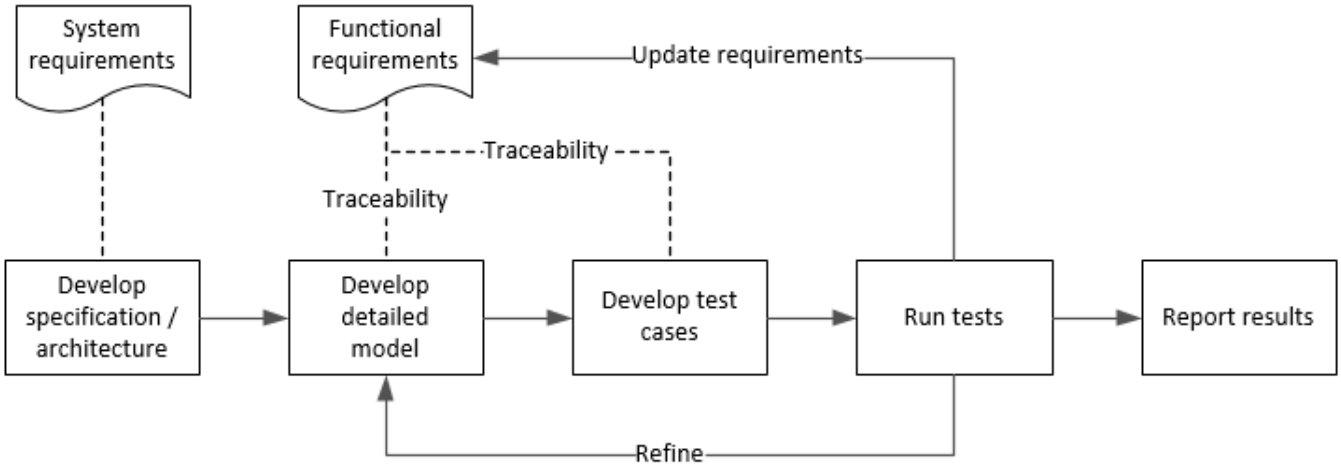
- “Test Model Against Requirements and Report Results” on page 2-2
- “Analyze a Model for Standards Compliance and Design Errors” on page 2-7
- “Perform Functional Testing and Analyze Test Coverage” on page 2-9
- “Analyze Code and Test Software-in-the-Loop” on page 2-12

Test Model Against Requirements and Report Results

Requirements – Test Traceability Overview

Traceability between requirements and test cases helps you interpret test results and see the extent to which your requirements are verified. You can link a requirement to elements that help verify it, such as test cases in the Test Manager, `verify` statements in a Test Sequence block, or Model Verification blocks in a model. When you run tests, a pass/fail summary appears in your requirements set.

This example demonstrates a common requirements-based testing workflow for a cruise control model. You start with a requirements set, a model, and a test case. You add traceability between the tests and the safety requirements. You run the test, summarize the verification status, and report the results.



In this example, you conduct a simple test of two requirements in the set:

- That the cruise control system transitions to disengaged from engaged when a braking event has occurred
- That the cruise control system transitions to disengaged from engaged when the current vehicle speed is outside the range of 20 mph to 90 mph.

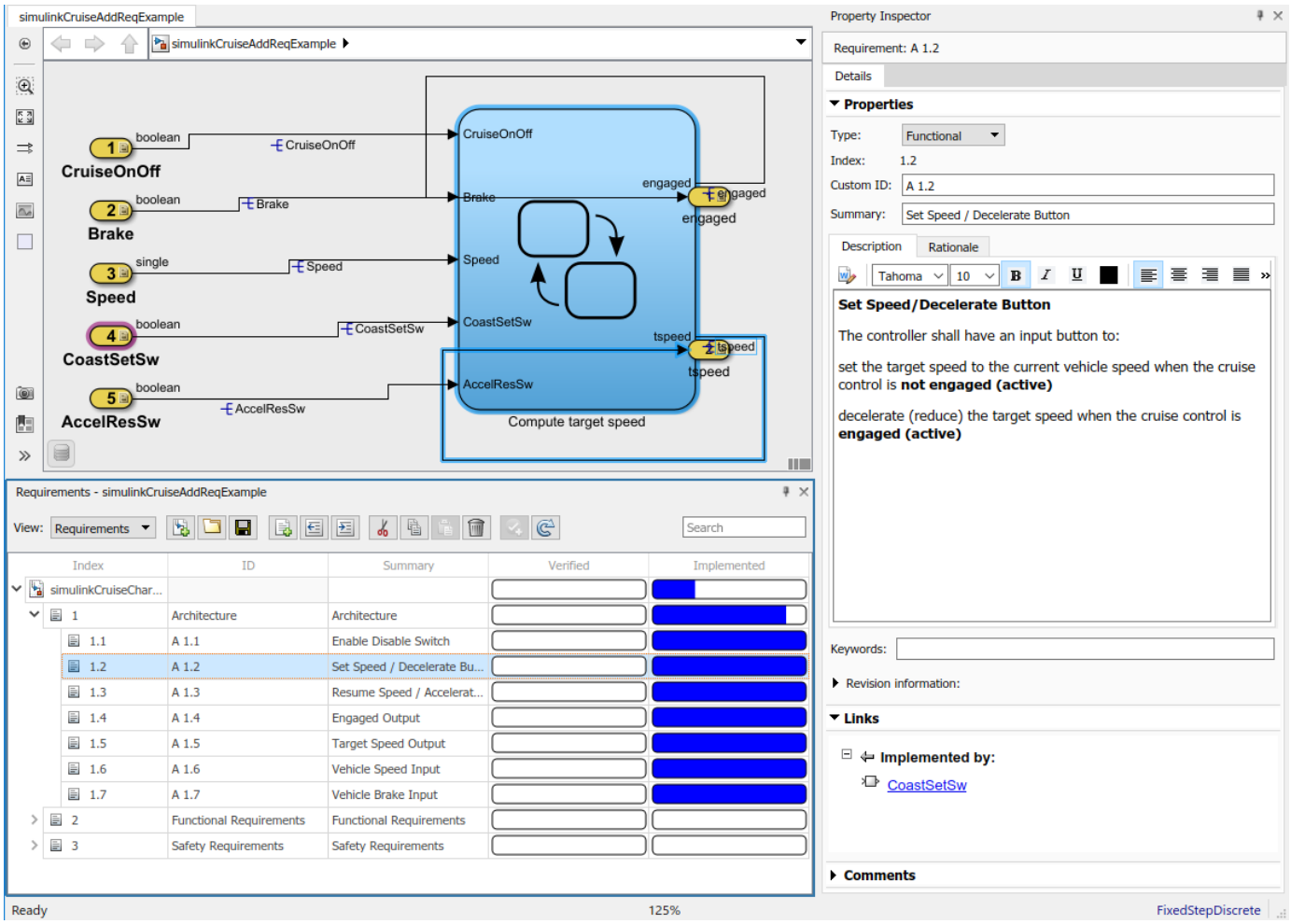
Display the Requirements

1. Create a copy of the project in a working folder. The project contains data, documents, models, and tests. Enter:

   ```
   path = fullfile(matlabroot,'toolbox','shared','examples',...
   'verification','src','cruise')
   run(fullfile(path,'slVerificationCruiseStart'))
   ```

2. In the project `models` folder, open the `simulinkCruiseAddReqExample.slx` model.
3. Display the requirements. Click the icon in the lower-right corner of the model canvas, and select **Requirements**. The requirements appear below the model canvas.

4 Expand the requirements information to include verification and implementation status. Right-click a requirement and select **Verification Status** and **Implementation Status**.



5 In the Project window, open the Simulink Test file `slReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

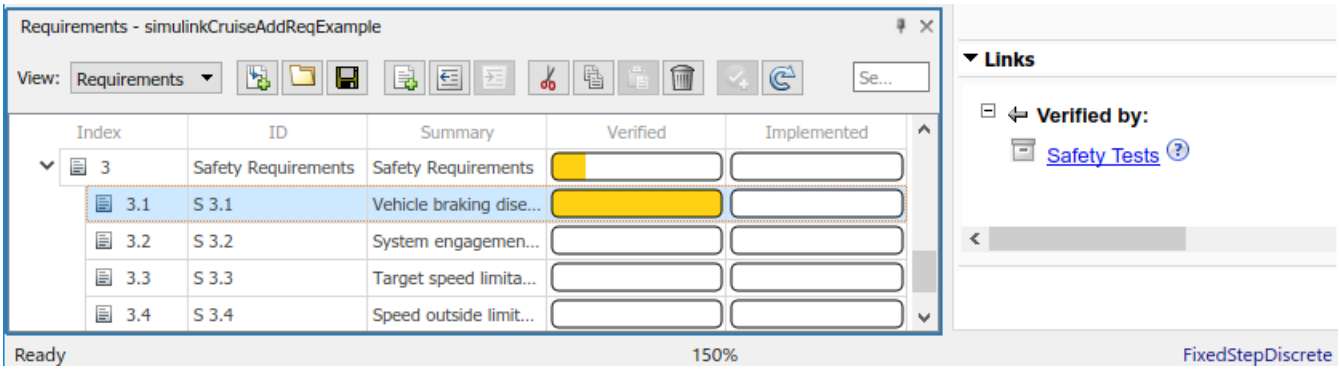
Link Requirements to Tests

Link the requirements to the test case.

1 In the Project window, open the Simulink Test file `slReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager. Explore the test suite and select `Safety Tests`.

Return to the model. Right-click on requirement `S 3.1` and select **Link from Selected Test Case**.

A link to the `Safety Tests` test case is added to **Verified by**. The yellow bars in the **Verified** column indicate that the requirements are not verified.



2 Also add a link for item `S 3.4`.

Run the Test

The test case uses a test harness `SafetyTest_Harness1`. In the test harness, a test sequence sets the input conditions and checks the model behavior:

- The `BrakeTest` sequence engages the cruise control, then applies the brake. It includes the `verify` statement

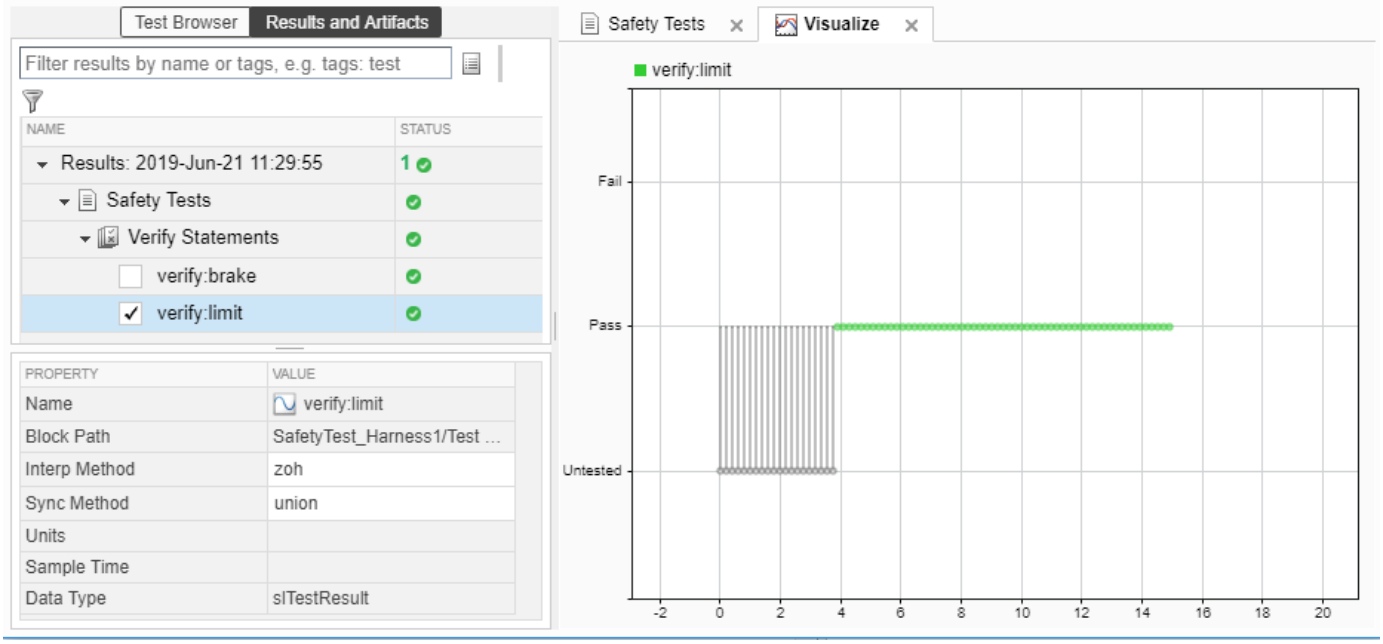
```
verify(engaged == false,...
    'verify:brake',...
    'system must disengage when brake applied')
```

- The `LimitTest` sequence engages the cruise control, then ramps up the vehicle speed until it exceeds the upper limit. It includes the `verify` statement.

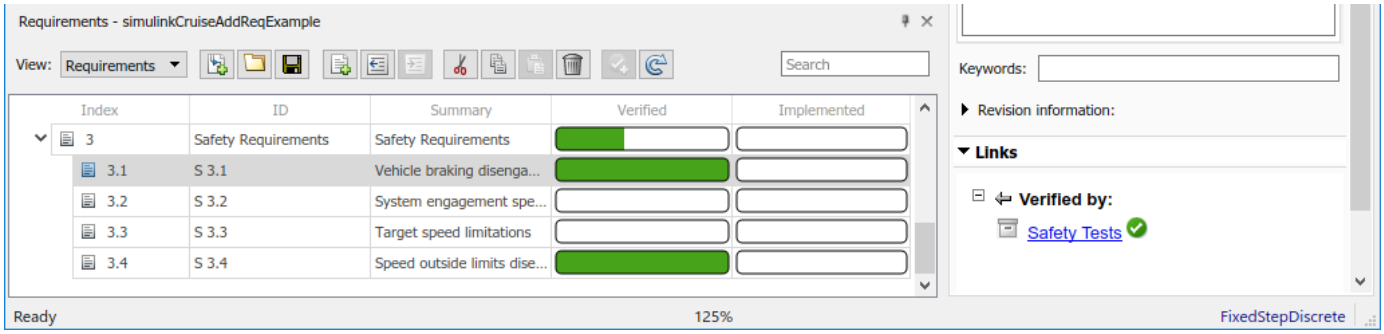
```
verify(engaged == false,...
    'verify:limit',...
    'system must disengage when limit exceeded')
```

1 Return to the Test Manager. To run the test case, click **Run**.

2 When the test finishes, review the results. The Test Manager shows that both assessments pass and the plot provides the detailed results of each `verify` statement.



3 Return to the model and refresh the Requirements. The green bar in the **Verified** column indicates that the requirement has been successfully verified.



Report the Results

1 Create a report using a custom Microsoft Word template.

- **a** From the Test Manager results, right-click the test case name. Select **Create Report**.
- **b** 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.
- **c** Enter a file name and select a location for the report.
- **d** For the **Template File**, select the `ReportTemplate.dotx` file in the **documents** project folder.
- **e** Click **Create**.

2 Review the report.

   a The **Test Case Requirements** section specifies the associated requirements

   b The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

See Also

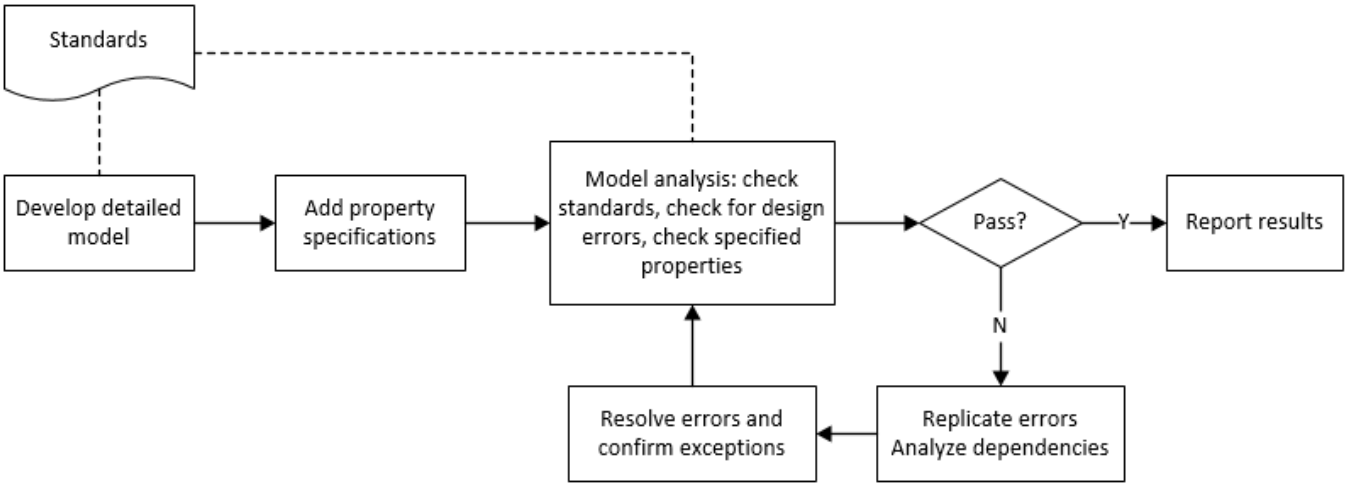
Related Examples

- “Link to Requirements” (Simulink Test)
- “Validate Requirements Links in a Model” (Simulink Requirements)
- “Customize Requirements Traceability Report for Model” (Simulink Requirements)

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 MAB 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® Advisory Board (MAB) 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 MAB Style Guideline Violations

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

1. Create a copy of the project in a working folder. On the command line, enter

   ```
   path = fullfile(matlabroot,'toolbox','shared','examples',...
   'verification','src','cruise')
   run(fullfile(path,'slVerificationCruiseStart'))
   ```

2. Open the model. On the command line, enter

   ```
   open_system simulinkCruiseErrorAndStandardsExample
   ```

3. In the **Modeling** tab, select **Model Advisor**.
4. Click OK to choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
5. Check your model for MAB style guideline violations using Simulink Check.

a In the left pane, in the **By Product > Simulink Check > Modeling Standards > MAB Checks** folder, select:

- **Check Indexing Mode**
- **Check model diagnostic parameters**

b Right-click on the **MAB Checks** node and select `Run Selected Checks`.

c Click **Check model diagnostic parameters** to review the configuration parameter settings that violate MAB 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 Check checks, select the **MAB 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**. All the checks in the folder are selected.

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**.

See Also

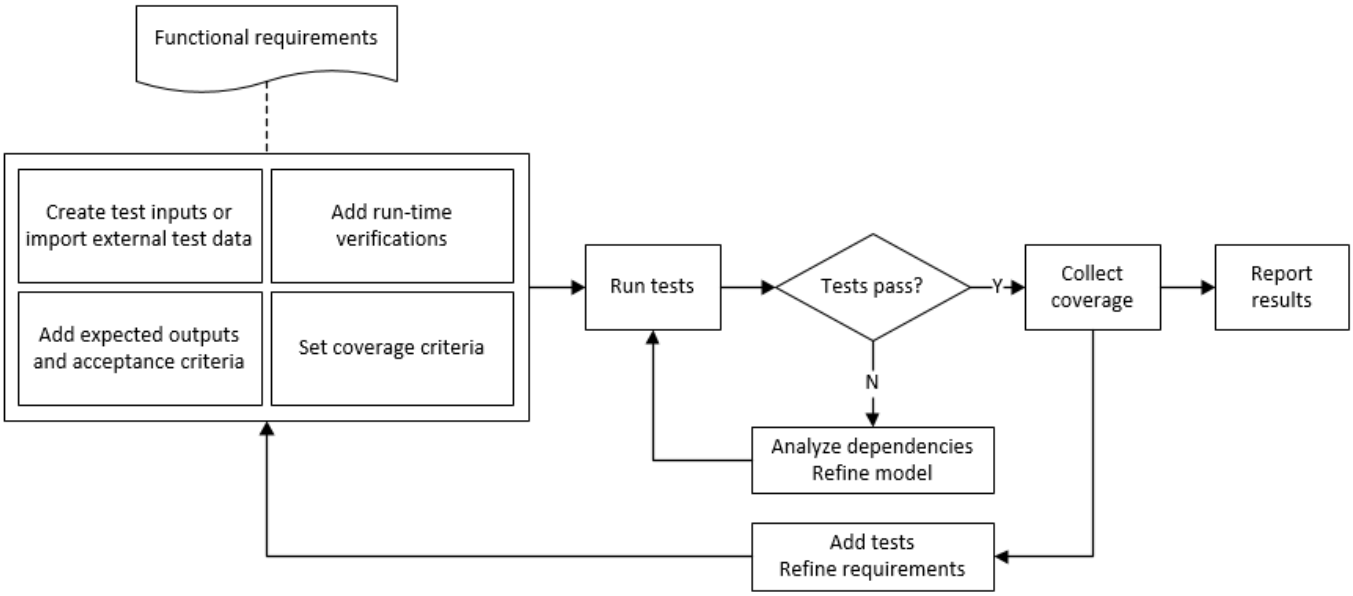
Related Examples

- “Check Model Compliance by Using the Model Advisor” on page 3-2
- “Collect Model Metrics Using the Model Advisor” on page 5-9
- “Run a Design Error Detection Analysis” (Simulink Design Verifier)
- “Prove Properties in a Model” (Simulink Design Verifier)

Perform Functional Testing and Analyze Test Coverage

Functional testing begins 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 iteratively. 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 Coverage, 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:

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'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” (Simulink Test) and open the Simulink Test Manager. At the command line, enter:

```
sltest.testmanager.load('slReqTests.mldatx')
sltest.testmanager.view
```

4 Open the test sequence block. The sequence tests that the system disengages when the:

- Brake pedal is pressed
- Speed exceeds a limit

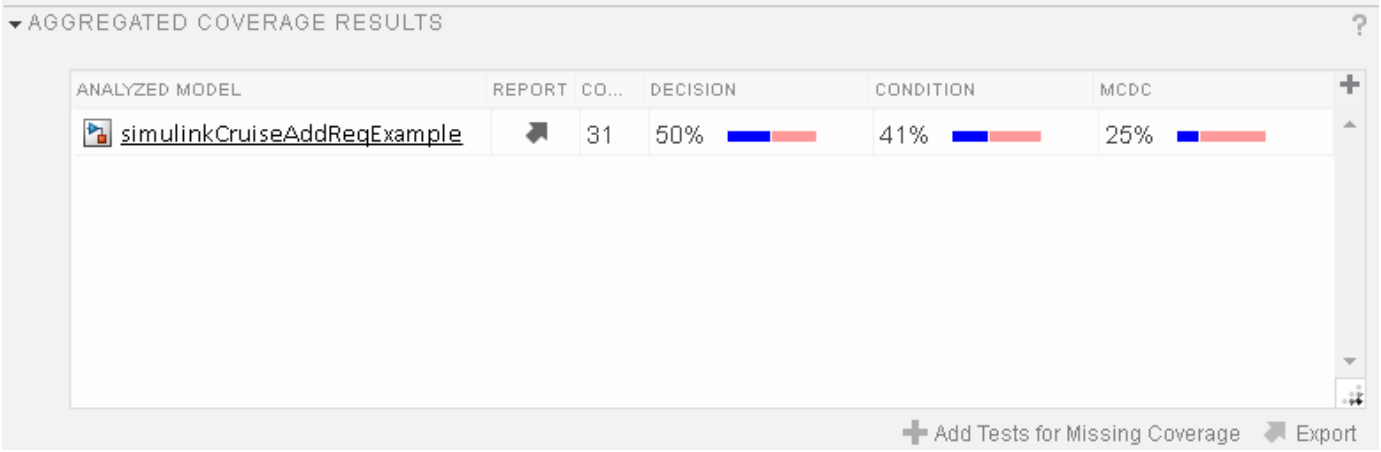
Some test sequence steps are linked to requirements document `simulinkCruiseChartReqs.docx`.

Measure Model Coverage

1 In the Simulink Test Manager, click the `slReqTests` test file.
2 To enable coverage collection for the test file, in the right page under **Coverage Settings**:

- Select **Record coverage for referenced models**
- Use **Coverage filter filename** to specify a coverage filter to use for the coverage analysis. The default setting honors the model configuration parameter settings. Leaving the field empty attaches no coverage filter.
- Select **Decision**, **Condition**, and **MCDC**.

3 To run the tests, on the Test Manager toolstrip, click **Run**.
4 When the test finishes select the Results in the Test Manager. The aggregated coverage results show that the example model achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.



Generate Tests to Increase Model Coverage

1 Use Simulink Design Verifier to generate additional tests to increase model coverage. In **Results and Artifacts**, select the `slReqTests` test file and open the **Aggregated Coverage Results** section located in the right pane.
2 Right-click the test results and select **Add Tests for Missing Coverage**.
3 Under **Harness**, choose `Create a new harness`.
4 Click **OK** to add tests to the test suite using Simulink Design Verifier. The model being tested must either be on the MATLAB path or in the working folder.
5 On the Test Manager toolstrip, click **Run** to execute the updated test suite. The test results include coverage for the combined test case inputs, achieving increased model coverage.

See Also

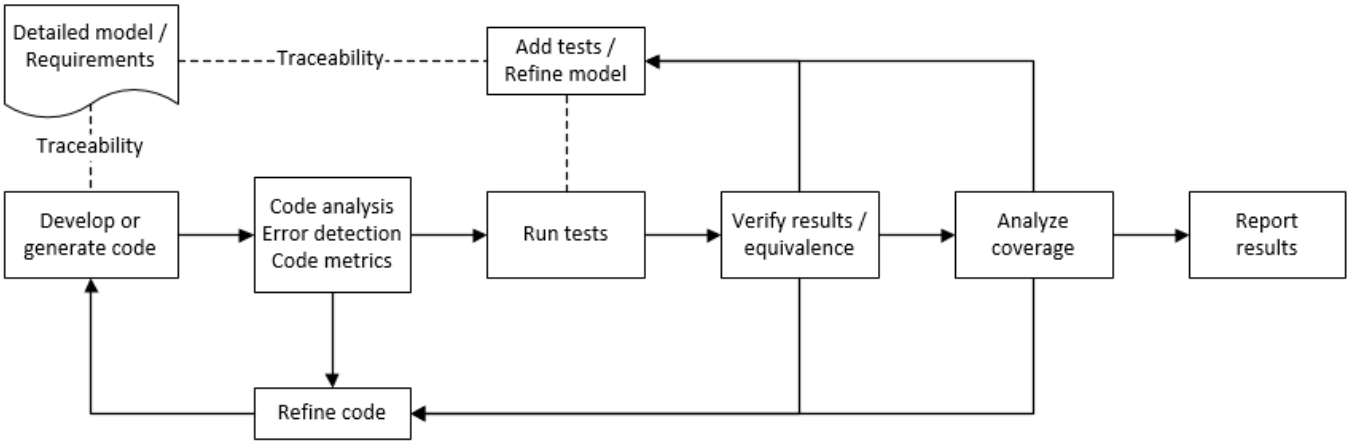
Related Examples

- “Link to Requirements” (Simulink Test)
- “Assess Model Simulation Using verify Statements” (Simulink Test)
- “Compare Model Output To Baseline Data” (Simulink Test)
- “Generate Test Cases for Model Decision Coverage” (Simulink Design Verifier)
- “Increase Test Coverage for a Model” (Simulink Test)

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.



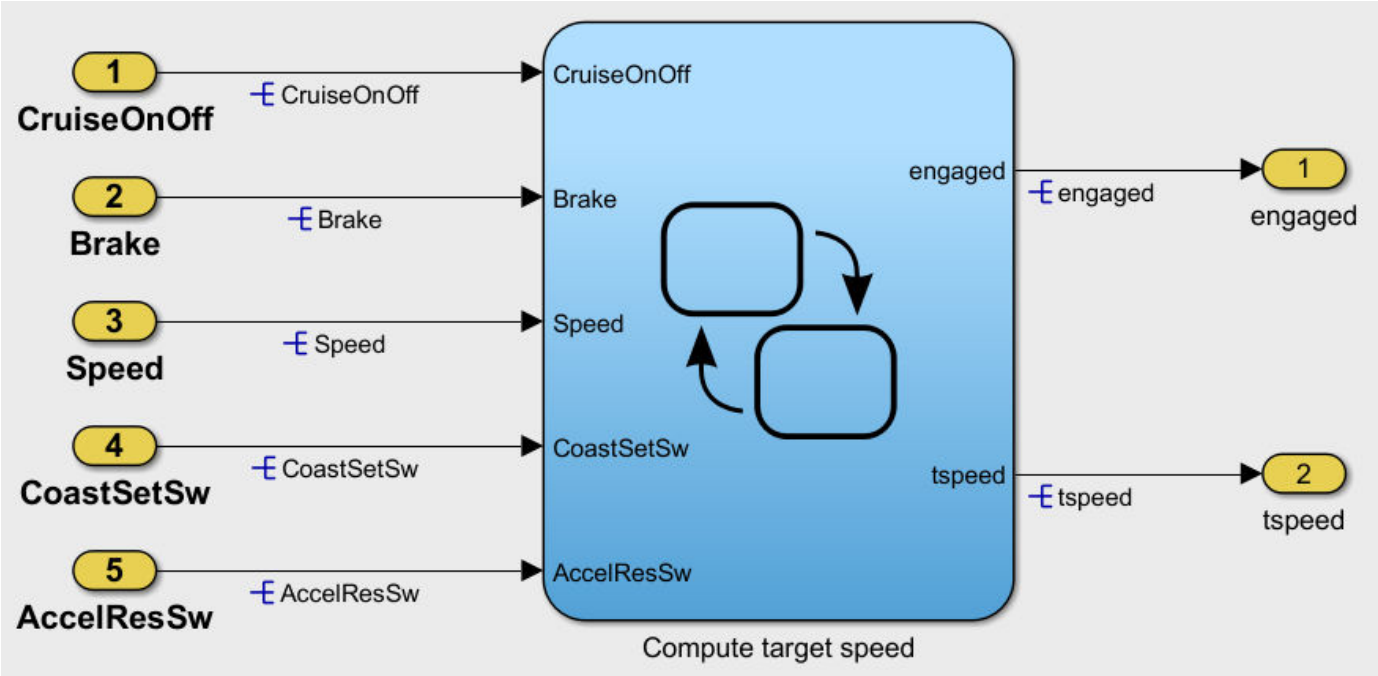
Analyze Code for Defects, Metrics, and MISRA C:2012

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 Advisor. 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 project.

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'slVerificationCruiseStart'))
```

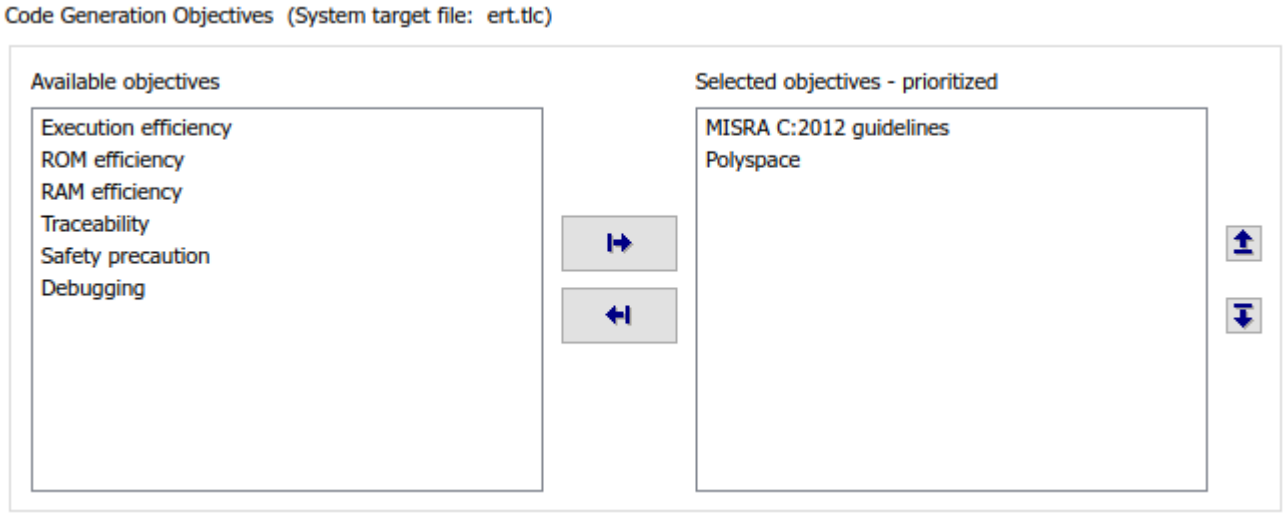
2. From the project, open the model `simulinkCruiseErrorAndStandardsExample`.



Run Code Generator Checks

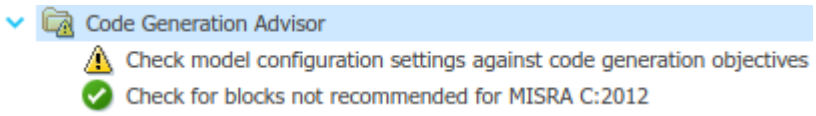
Before you generate code from your model, there are steps that you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows how to use the Code Generation Advisor to check your model before generating code.

1. Right-click Compute target speed and select **C/C++ Code** > **Code Generation Advisor**.
2. Select the Code Generation Advisor folder. In the right pane, move `Polyspace` to **Selected objectives - prioritized** . The `MISRA C:2012 guidelines` objective is already selected.



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 model, 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 did not pass. Accept the parameter changes by selecting **Modify Parameters**.
5. Rerun the check by selecting **Run This Check**.

Run Model Advisor Checks

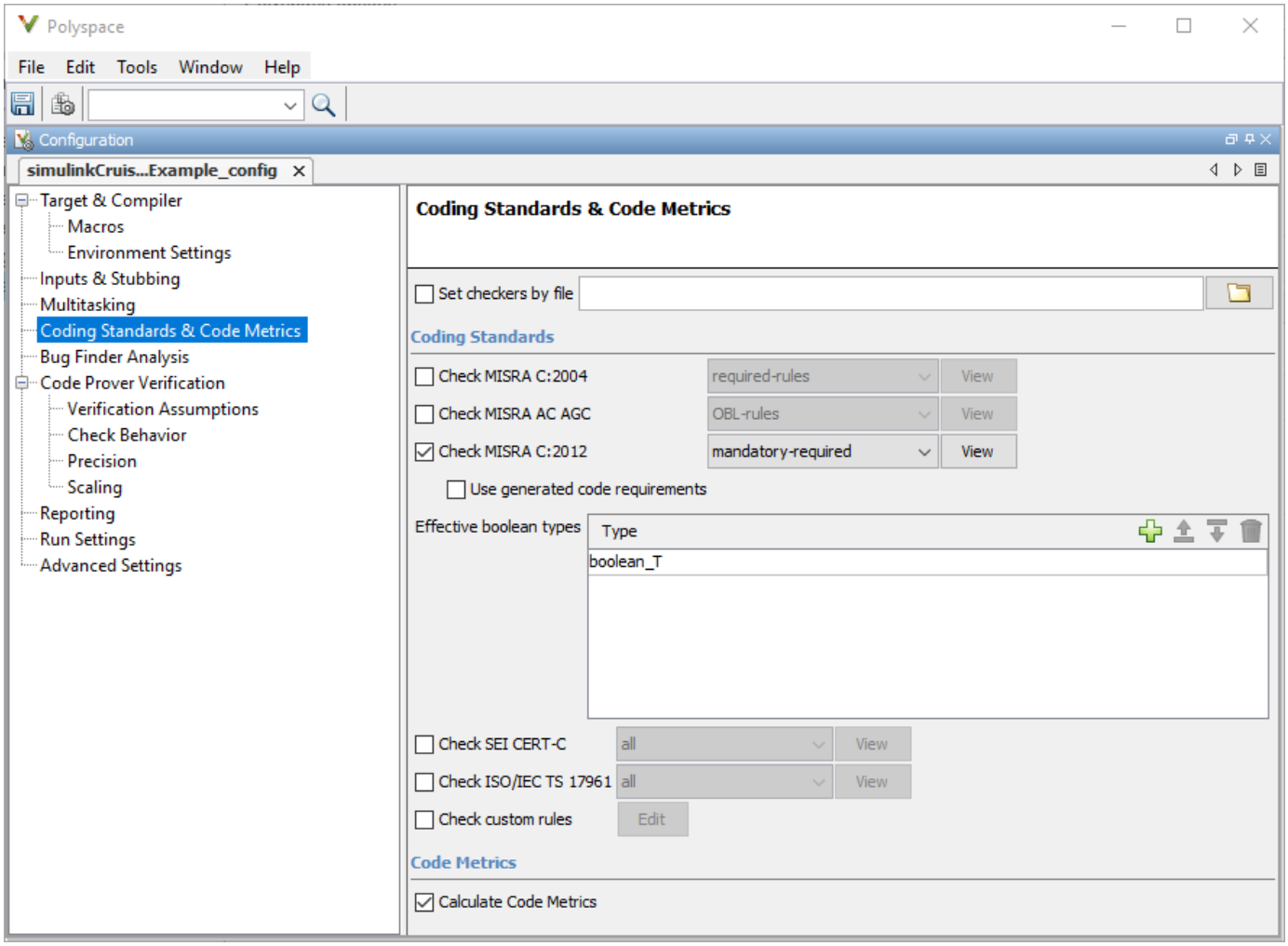
Before you generate code from your model, there are steps you can take to generate code that is more compliant with MISRA C and more compatible with Polyspace. This example shows you how to use the Model Advisor to check your model before generating code.

1. At the bottom of the Code Generation Advisor window, select **Model Advisor**.
2. Under the **By Task** folder, select the **Modeling Standards 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.

Generate and Analyze Code

After you have done the model compliance checking, you can generate the 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++ Code > 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**.
4. Click the **Configure** (Polyspace Bug Finder) button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.



5 On the same pane, select **Calculate Code Metrics**. This option turns on code metric calculations for your generated code.

6 Save and close the Polyspace configuration window.

7 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.

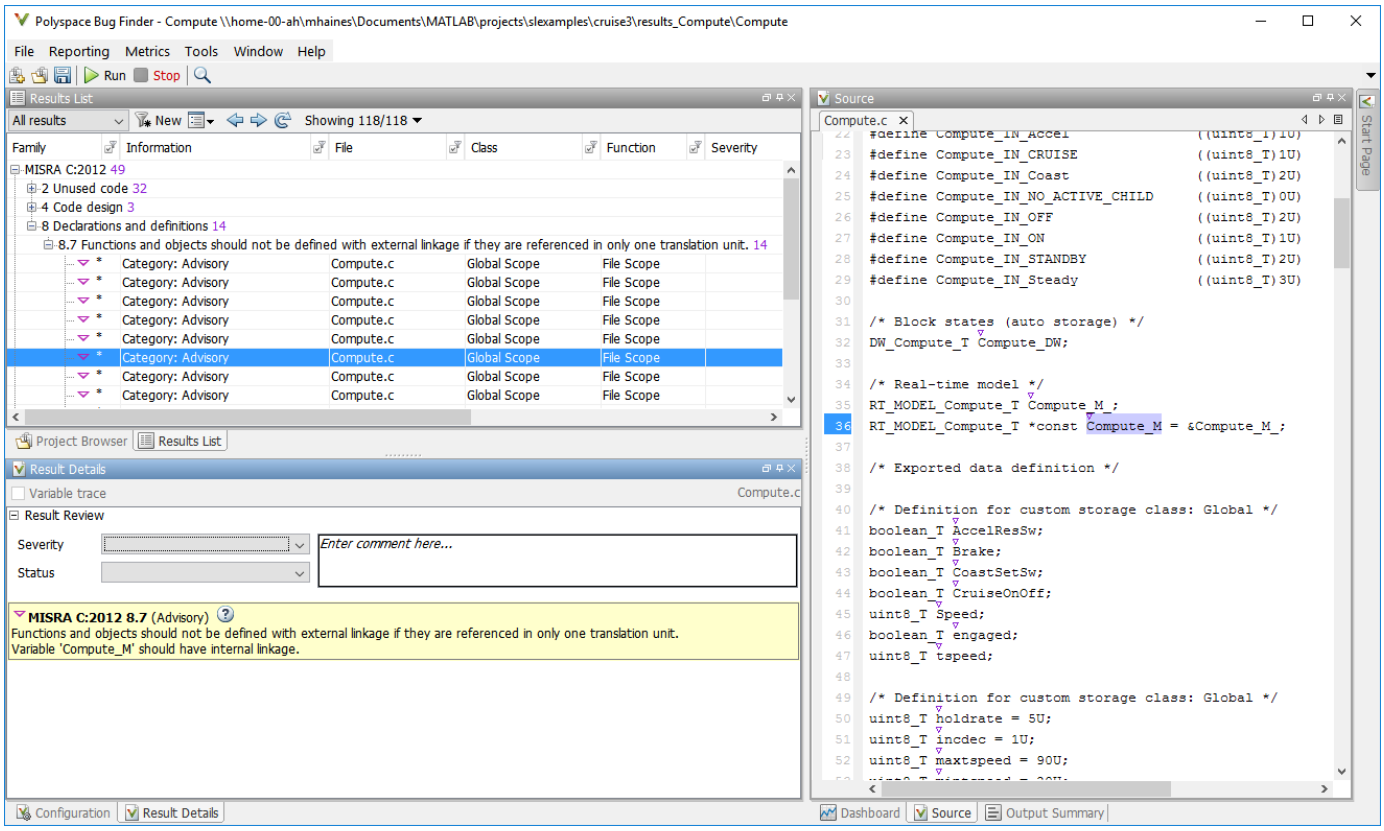
Review Results

After you run a Polyspace analysis of your generated code, the Polyspace environment shows you the results of the static code analysis.

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

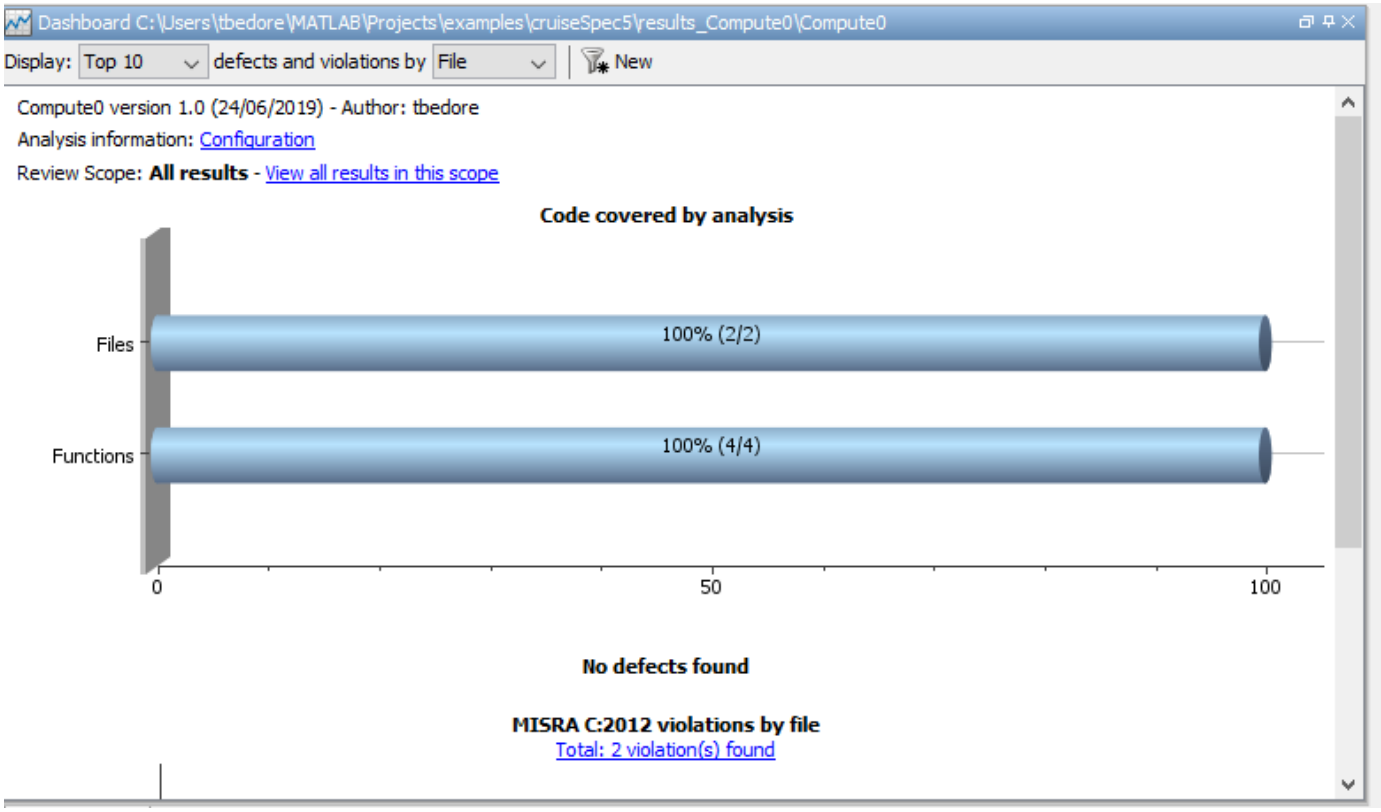
Rule 8.7 states that functions and objects should not be global if the function or object is local. As you click through the 8.7 violations, you can see that these results refer to variables that other components also use, such as `CruiseOnOff`. You can annotate your code or your model to justify

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** (Polyspace Bug Finder) option to `Project configuration`. This option allows 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 Standards & 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.

   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, only two 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**.

See Also

Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Bug Finder)
- “Test Two Simulations for Equivalence” (Simulink Test)
- “Export Test Results and Generate Test Results Reports” (Simulink Test)

Checking Systems Interactively

Check Model Compliance by Using the Model Advisor

Model Advisor Overview

The Model Advisor checks your model or subsystem for modeling conditions and configuration settings that cause inaccurate or inefficient simulation of the system that the model represents. The Model Advisor checks can help you verify compliance with industry standards and guidelines. By using the Model Advisor, you can implement consistent modeling guidelines across projects and development teams.

Upon completing the analysis of your model, the Model Advisor produces a report that lists the suboptimal conditions, settings, and modeling techniques and proposes solutions, when applicable.

You can use the Model Advisor to check your model in these ways:

- Interactively run Model Advisor checks
- Configure the Model Advisor to automatically run edit-time checks (requires Simulink Check)

These limitations apply when you use the Model Advisor to check your model. For limitations that apply to specific checks, see the Capabilities and Limitations section in the check documentation.

- If you rename a system, you must restart the Model Advisor to check that system.
- In systems that contain a variant subsystem, the Model Advisor checks the active subsystem. To check both the active and inactive subsystems, set the `Advisor.Application` property, `AnalyzeVariants`, to `true`.
- Model Advisor does not analyze commented blocks.
- Checks do not search in model blocks or subsystem blocks with the block parameter **Read/Write** set to `NoReadorWrite`. However, on a check-by-check basis, Model Advisor checks do search in library blocks and masked subsystems.
- Unless specified otherwise in the documentation for a check, the Model Advisor does not analyze the contents of a Model block. To run checks on referenced models, use instances of the `Advisor.Application` class (Simulink Check license required).

Note Software is inherently complex and may not be free of errors. Model Advisor checks might contain bugs. MathWorks reports known bugs brought to its attention on its Bug Report system at https://www.mathworks.com/support/bugreports/. The bug reports are an integral part of the documentation for each release. Examine 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 increases the likelihood that your model does not violate certain modeling standards or guidelines, their application cannot guarantee that the system being developed will be safe or error-free. It is ultimately your responsibility to verify, using multiple methods, that the system being developed provides its intended functionality and does not include unintended functionality.

Model Advisor Checks Documentation

The Model Advisor only displays the checks for your installed products. This table provides links to the product-specific check documentation. A product license may be required to review some of the documentation.

Product	Model Advisor Check Documentation
Simulink	“Simulink Checks” (Simulink)
Embedded Coder	“Embedded Coder Checks” (Embedded Coder)
AUTOSAR Blockset	“MathWorks AUTOSAR Blockset Checks” (AUTOSAR Blockset)
Simulink Coder™	“Simulink Coder Checks” (Simulink Coder)
HDL Coder™	“HDL Code Advisor Checks” (HDL Coder)
Simulink Code Inspector™	“Simulink Code Inspector Checks” (Simulink Code Inspector)
Simulink Check	“DO-178C/DO-331 Checks” “IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks” “Model Checks for DO-254 Standard Compliance” on page 3-50 “High Integrity System Modeling Checks” “Model Advisor Checks for MAB and JMAAB Compliance” “MISRA C:2012 Checks” “Secure Coding Checks for CERT C, CWE, and ISO/IEC TS 17961 Standards” “Model Metrics” “Clone Detection Checks”
Simulink Design Verifier	“Simulink Design Verifier Checks” (Simulink Design Verifier)
Simulink Requirements	“Requirements Consistency Checks” (Simulink Requirements)
Simscape™	Documentation is available only in the Model Advisor. To review the documentation for the check, in the Model Advisor, right-click on the check title and select **What's This?**
Simulink Control Design™	“Simulink Control Design Checks” (Simulink Control Design)

Product	Model Advisor Check Documentation
IEC Certification Kit	“IEC Certification Kit Checks for Bug Reports” (IEC Certification Kit) “High Integrity System Modeling Checks”
DO Qualification Kit	“DO Qualification Kit Checks for Bug Reports” (DO Qualification Kit) “High Integrity System Modeling Checks”

Run Model Advisor Checks and Review Results

You can use the Model Advisor to check your model interactively against modeling standards and guidelines. The following example uses the `sldemo_mdladv` model to demonstrate the execution of the Model Advisor checks using the Model Advisor.

1. Open the Model Advisor example model `sldemo_mdladv`.
2. To open the Model Advisor, in the Simulink editor, click the **Modeling** tab and select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Select the model or system that you want to review and click **OK**.
3. In the left pane of the Model Advisor, select the checks you want to run on your model:
   a. You can select the checks by using the **By Product** or **By Task** folders. If these folders are not displayed in the Model Advisor window, open **Settings** > **Preferences** and select:
      - **Show By Product Folder** — Displays checks available for each product
      - **Show By Task Folder** — Displays checks related to specific tasks
   b. You can search for and execute a specific check by enter the *Title* or *TitleID* of the check in the **Find:** field and click the **Find Next** button. The Model Advisor searches in check names, folder names, and analysis descriptions. You can use the **Source** tab to identify the *Title*, *TitleID*, and location of the MATLAB® source code for each check. To display the **Source** in the right pane of the Model Advisor, open **Settings** > **Preferences** and select **Show Source Tab**.
4. Click on the folder that contains the checks and, on the right pane of the Model Advisor, select:
   - **Show report after run** to automatically generate and display the report in HTML format
   - **Run Selected Checks** to execute the analysis.

   To run a single check, right-click the check in the folder and select **Run This Check**.
5. View the results on the Model Advisor User Interface. Common check status results include
   - `Pass` − Check did not identify issues.
   - `D-Pass` − Dependent on configuration parameter or successful execution of another check.
   - `Warn` − Check has identified issues.
   - `Fail` − Check fails to execute.
6. Fix the warnings or failures as desired. For more information, see “Address Model Check Results” (Simulink).

7 Use the **Exclusions** tab to review checks that were marked for exclusion from the analysis. To display the **Exclusions** tab in the right pane of the Model Advisor, open **Settings** > **Preferences** and select **Show Exclusion tab**.

8 View and save the report. For additional information, see "Save and View Model Advisor Check Reports" (Simulink).

Note If you did not select **Show report after run** when you executed the checks, you can generate a report of the results after the analysis is complete. See "Generate Model Advisor Reports" on page 3-19.

9 If desired, you can reset the status of the checks to the Not Run state. In the left pane, right-click on **Model Advisor** and select `Reset`. This action does not delete the results of the analysis from the Model Advisor.

Save Analysis Time by Running the Checks from a Previous Analysis

You can save time by consistently running the same set of checks on your model by using the Model Advisor dashboard. When you use the dashboard, the Model Advisor does not reload the checks before executing them, saving analysis time.

1 Open the Model Advisor example model `sldemo_mdladv`.

2 Select **Model Advisor** > **Model Advisor Dashboard**. A **System Selector — Model Advisor** dialog box opens. Select the model or system that you want to review and click **OK**.

3 The Model Advisor Dashboard window opens. From this dashboard, you can:

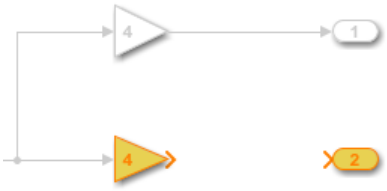
- Click the **Run checks** button to execute the same checks from the previous analysis
- Click the **Switch to standard view** button to open the Model Advisor and select different checks
- Click the **Enable Highlighting** button to view the highlighted results in the Simulink editor

4 Click the **Run checks** button to run the same checks on the model that were used in the previous analysis. If desired, click the **Enable Highlighting** button.

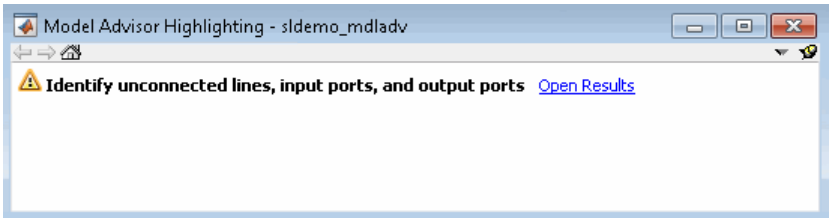
5 The Model Advisor execute the checks and updates the dashboard to reflect the results of the analysis, including the number of:

- Passed checks
- Failed checks
- Flagged checks
- Total checks

If you clicked the **Enable Highlighting** button, the flagged results are highlighted in the model.



The Model Advisor Highlighting information 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.



6 Click the **Open Report** button to open the entire report in HTML format. Alternatively, you can select the number link beside the results to filter the report results.

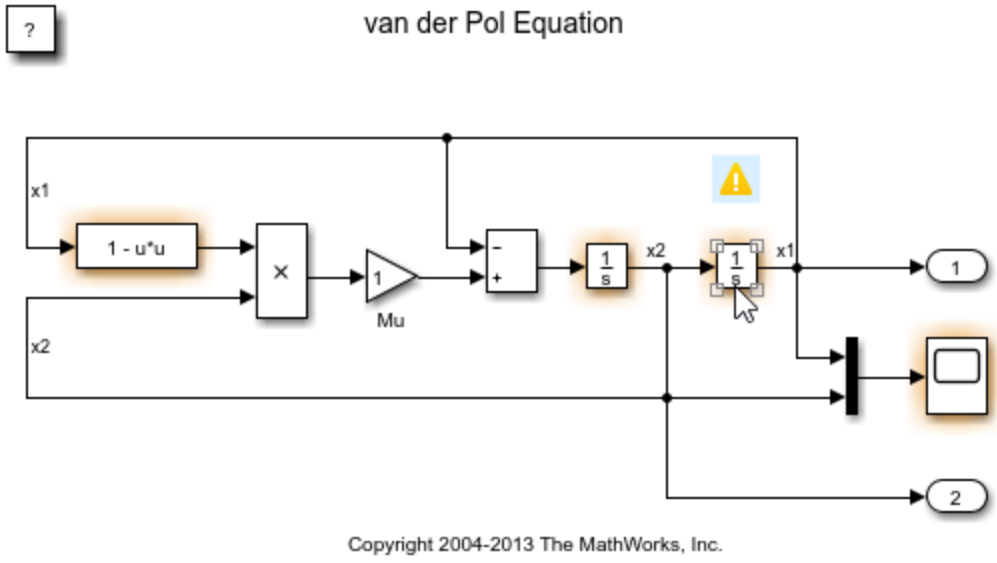
Check Your Model by Using Edit Time Checks

When you enable edit-time checks, the Model Advisor evaluates the model against a subset of Model Advisor checks. Highlighted blocks in the model editor window alert you to issues in your model. This enables you to identify modeling issues earlier in the model design process.

You can use one of these methods to enable edit-time checking of your model:

- In the **Debug** tab, select **Diagnostics > Edit-Time Checks**
- In the **Modeling** tab, select **Model Advisor > Edit-Time Checks**
- If you have an Embedded Coder or Simulink Coder license, you can use edit-time checks to evaluate your model for issues that are specific to code generation. To enable these checks, open the C Code app and select the **C/C++ Code Advisor > Edit-Time Checks**.

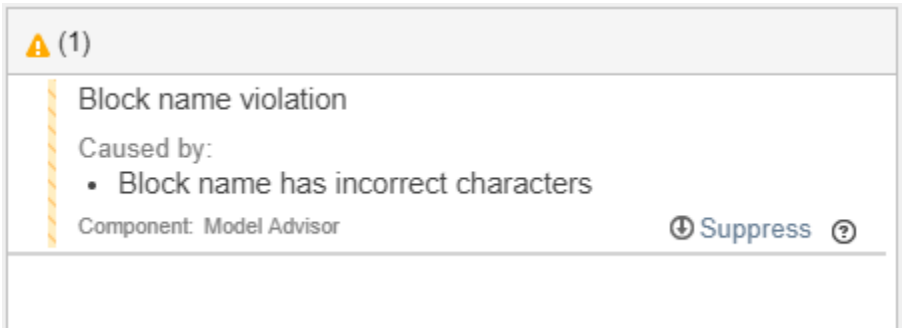
When edit-time checking is enabled, the Model Advisor highlights blocks in your model that violate Model Advisor checks.



To review a check violation, click the error or warning icon above the highlighted block. A diagnostics window opens, which provides information about the modeling issue that violates the Model Advisor check. When a block violates multiple checks, you can use the diagnostics window to review all issues.

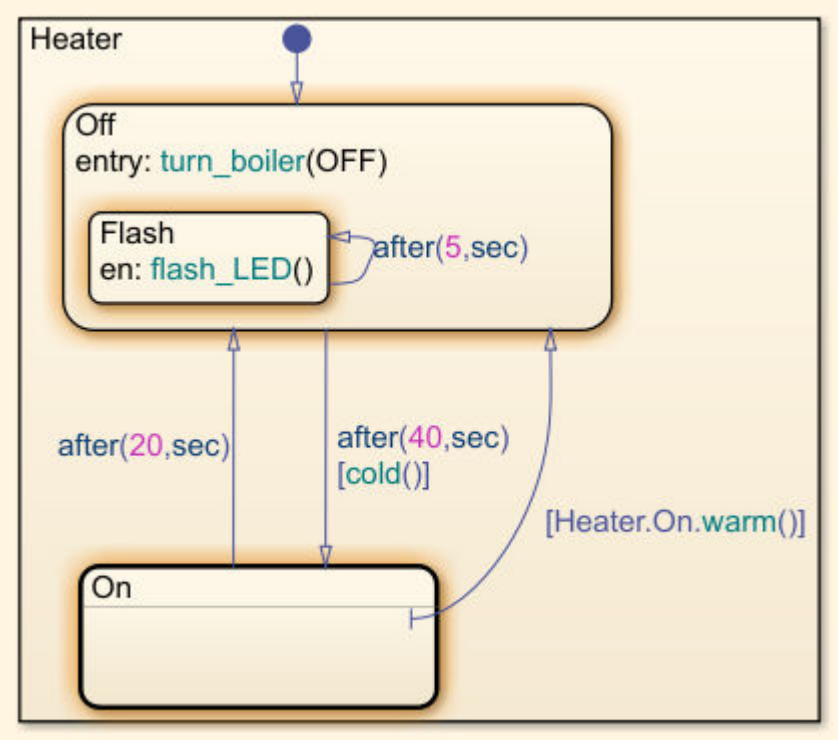
For each modelling issue, you can use the diagnostics window to:

- Review the cause and explore suggested options for fixing the issue, if any.
- Click the question mark to access detailed documentation about the violated Model Advisor check.
- Ignore the warning and add the block to the exclusion list for that check by clicking **Suppress**.



In this example, you use edit-time checking to verify the compliance of a Stateflow chart with the MAB guidelines while you edit.

1. Open a model that contains Stateflow charts. For example, at the command prompt, type: `open sf_boiler`.
2. To enable the edit-time checking, in the **Modeling** tab, select **Model Advisor** > **Edit-Time Checks** .
3. Open the **Bang-Bang Controller** chart by double-clicking it. The Model Advisor highlights multiple states. Place your cursor over the warning of the **Off** state to discover the issue.



4 Select the warning. The Model Advisor indicates that there must be a new line after **entry** action. In your model, place your cursor after the **en:** and press **Enter**. A new line is added and the warning is cleared.

View and Customize the Edit-Time Checks in a Model Advisor Configuration

The Model Advisor checks that are available for edit-time checking are specified by using a Model Advisor configuration file. You use the Model Advisor Configuration Editor to review and modify existing configuration files and create new configuration files.

To open a Model Advisor configuration file and review the Model Advisor checks that are enabled for use in edit-time checking:

1 In the Simulink editor, click the **Modeling** tab and select **Model Advisor > Customize Edit-Time Checks**.

2 The Model Advisor Configuration Editor opens. The file name for the configuration that is currently being used by the Model Advisor is displayed at the top of the window. Verify that you are evaluating the correct configuration file. To open a different configuration file, click **Open** and browse to the file you would like to review.

3 In the Model Advisor Configuration Editor, on the Model Advisor tab, select the `Edit time supported checks` option. The filtered list identifies the Model Advisor checks that are supported for edit-time checking.

Note When a check is included in multiple folders of your Model Advisor hierarchy, for edit-time checking, the Model Advisor prioritizes 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.

4 In the **Model Advisor** tab, check the box beside the checks that you want to include in the edit-time check analysis. Deselect the box beside the checks that you do not want analyzed. For additional information about using the Model Advisor Configuration Editor to create a custom Model Advisor configuration, including the customization of edit-time checks, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” on page 3-4
- “Address Model Check Results” (Simulink)
- “Generate Model Advisor Reports” on page 3-19
- “Save and View Model Advisor Check Reports” (Simulink)
- “Find Model Advisor Check IDs” (Simulink)
- “Archive and View Results” on page 4-7
- “Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration” on page 7-7

More About

- “Check Your Model Using the Model Advisor” (Simulink)
- “Exclude Blocks From the Model Advisor Check Analysis” on page 3-10
- “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3

Exclude Blocks From the Model Advisor Check Analysis

Model Advisor Exclusion Overview

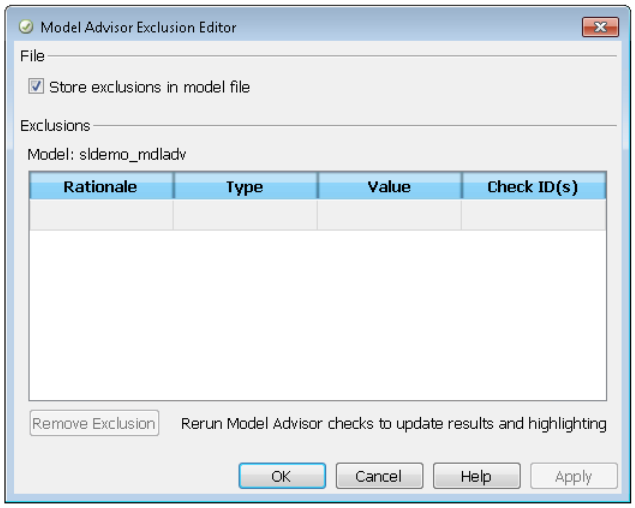
To save time during model development and verification, you can limit the scope of a Model Advisor analysis of your model. You can create a Model Advisor exclusion to exclude blocks in the model from selected checks. You can exclude all or selected checks from:

- 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.

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.	• **Exclude block only > Select Checks**. • In the Check Selector dialog box, select the checks. Click **OK**.
Exclude all blocks of this type from selected checks.	• **Exclude all blocks with type <block_type> > Select Checks**. • In the Check Selector dialog box, select the checks. Click **OK**.
Exclude the block from all failed checks. After a Model Advisor analysis, this option is available.	**Exclude block only > Only failed checks**
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**

To	Select Model Advisor >
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.
- In the **Modeling** tab, open **Model Advisor > 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 **Modeling** tab, select **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** ( ).

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.
- After you run the checks, in the left pane of the Model Advisor window, the checks that contain exclusion rules are highlighted in orange. The Model Advisor results include additional information about the exclusion.

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_mdladv`:

   ```
   mo = get_param('sldemo_mdladv','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.

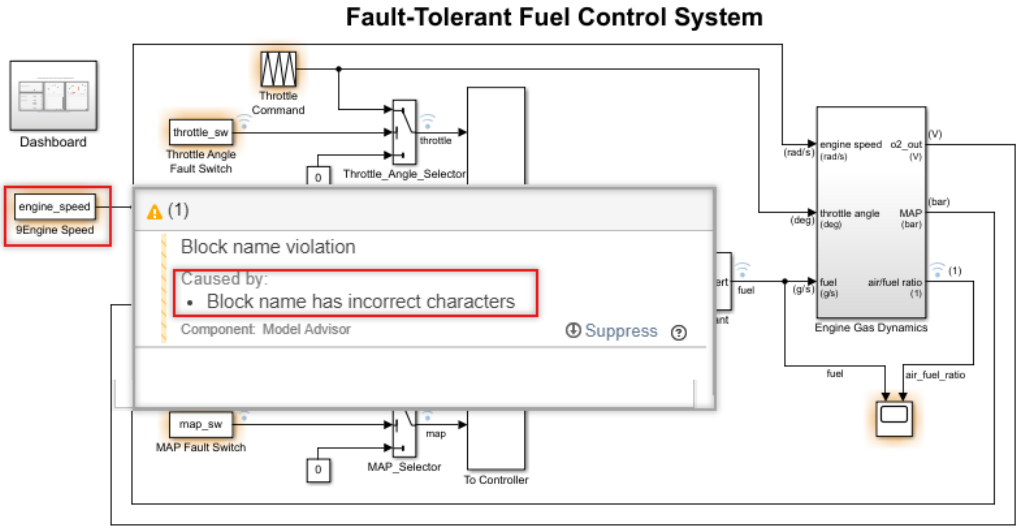
Exclude Blocks from Edit Time Checking

While editing a model, you can exclude blocks from Model Advisor analysis. Applicable Model Advisor exclusions specified through the Simulink editor are also applied during edit-time.

To exclude a block from Model Advisor analysis during edit-time:

1. From the command prompt, open `sldemo_fuelsys`.
2. Introduce a warning that is visible in edit-time checking. Add the number 9 to the beginning of the Engine Speed block name, which results in a violation of Model Advisor check “Check character usage in block names”.
3. In the **Modeling** tab, select **Edit-Time Checks**. Multiple blocks are flagged as Model Advisor check violations.

For instance, the Engine Speed block is highlighted and, when you click the warning icon above the block, you see that the cause of the violation is `Block name has incorrect characters`.



4 To exclude the Engine Speed block from Model Advisor analysis, either:

   a Right-click the block, select **Model Advisor > Exclude block only > Select checks**, and select the check.

   b Click the warning icon above the highlighted Engine Speed block. In the diagnostic window, select **Suppress**.

The block is excluded from Model Advisor analysis for that check and is not highlighted for check violation.

Note The list of exclusions is shared between all executions of the Model Advisor checks.

Limit Model Checks by Excluding Gain and Outport Blocks

This example shows how to exclude a Gain block and all Outport blocks from a Model Advisor check during a Model Advisor analysis. By excluding individual blocks from checks, you limit the scope of the analysis and might save time during model development and verification.

1 At the MATLAB command line, type `sldemo_mdladv`.

2 From the model window, in the **Modeling** tab, select **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, 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.

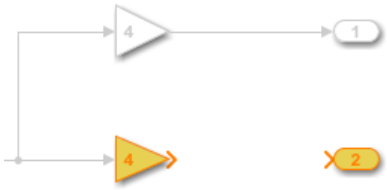
4 **Note** If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

5 Run the selected checks by clicking the **Run selected checks** button. After the Model Advisor runs the checks, an HTML report displays the check results in a browser window. The check **Identify unconnected lines, input ports, and output ports** triggers a warning.

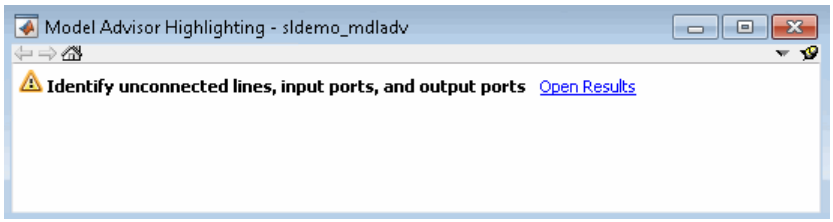
6 In the left pane of the Model Advisor window, select the check **By Product > Simulink > Identify unconnected lines, input ports, and output ports**.

7 In the Model Advisor window, click the **Enable highlighting** button ( ).

- The model window opens. The blocks causing the **Identify unconnected lines, input ports, and output ports** check warning are highlighted in yellow.

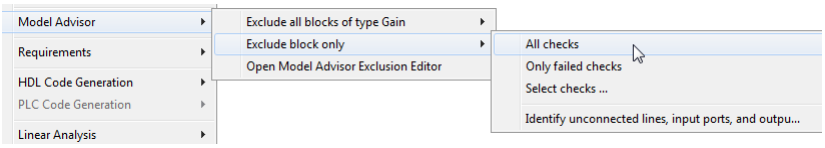


- The Model Advisor Highlighting window opens with a link to the Model Advisor window. In the Model Advisor window, you can find more information about the check results and how to fix the warning condition.

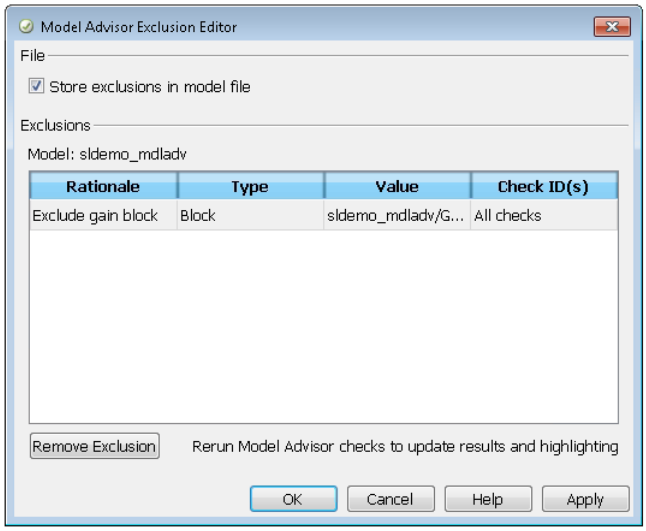


8 After reviewing the check results, exclude the Gain2 block from all Model Advisor checks:

a In the model window, right-click the Gain2 block and select **Model Advisor > Exclude block only > All checks** .



b In the Model Advisor Exclusion Editor dialog box, double-click in the first row of the **Rationale** field, and enter `Exclude gain block`.



c Click **OK** to store the exclusion in the model file.

9 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 Outport > 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.

10 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.

11 After the Model Advisor completes the analysis, you can view exclusion information for the Identify unconnected lines, input ports, and output ports check in the:

- HTML report:

Identify unconnected lines, input ports, and output ports

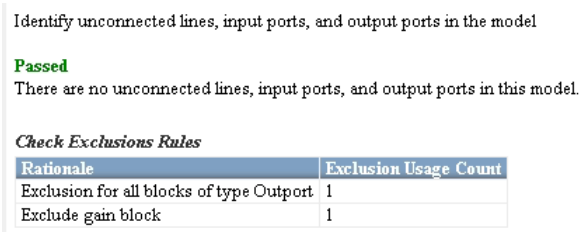
Identify unconnected lines, input ports, and output ports in the model

Passed
There are no unconnected lines, input ports, and output ports in this model.

Check Exclusions Rules

Rationale	Exclusion Usage Count
Exclusion for all blocks of type Outport	1
Exclude gain block	1

- Model Advisor window. In the left pane of the Model Advisor window, select **By Product > Simulink > Identify unconnected lines, input ports, and output ports**.


Identify unconnected lines, input ports, and output ports in the model

Passed

There are no unconnected lines, input ports, and output ports in this model.

Check Exclusions Rules

Rationale	Exclusion Usage Count
Exclusion for all blocks of type Outport	1
Exclude gain block	1

12 Close `sldemo_mdladv`.

See Also

Related Examples

- “Exclude Blocks From Custom Checks” on page 6-47
- “Run Model Advisor Checks and Review Results” on page 3-4
- “Address Model Check Results” (Simulink)
- “Generate Model Advisor Reports” on page 3-19
- “Save and View Model Advisor Check Reports” (Simulink)
- “Find Model Advisor Check IDs” (Simulink)
- “Archive and View Results” on page 4-7

More About

- “Check Your Model Using the Model Advisor” (Simulink)
- “Exclude Blocks From the Model Advisor Check Analysis” on page 3-10

Generate Model Advisor Reports

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 HTML, Adobe® PDF, and Microsoft Word .docx formats.

The beginning of the Model Advisor reports contain the:

- Model name
- Simulink version
- System
- Treat as Referenced Model
- Model version
- Current run

Generate Results Report When Executing Model Advisor Checks

Use these steps to generate a Model Advisor report when executing the checks. This report is in HTML format.

1. In the left pane of the Model Advisor, select the checks you want to run.
2. Click on the folder that contains the checks and, in the right pane of the Model Advisor, select **Show report after run**.
3. Click **Run Selected Checks**. When complete, the Model Advisor automatically opens an `HTML` version of the report.
4. To save, right-click on the report and select **Save As**

Generate Results Report After Executing Model Advisor Checks

To generate a Model Advisor report in Adobe PDF or Microsoft Word:

1. In the left pane of the Model Advisor, select the checks you want to run. Click on the folder that contains the checks and, in the right pane of the Model Advisor, select **Run Selected Checks**.
2. When complete, reselect the folder and click **Generate Report** in the right pane of the Model Advisor.
3. 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.
   - Use **File format** to select `HTML`, `PDF`, or `Word`.
4. Click **OK**. The Model Advisor generates the report and saves it to the designated location. If you selected **View report after generation**, the report opens automatically.

Modify Template for Model Advisor Check Results Report

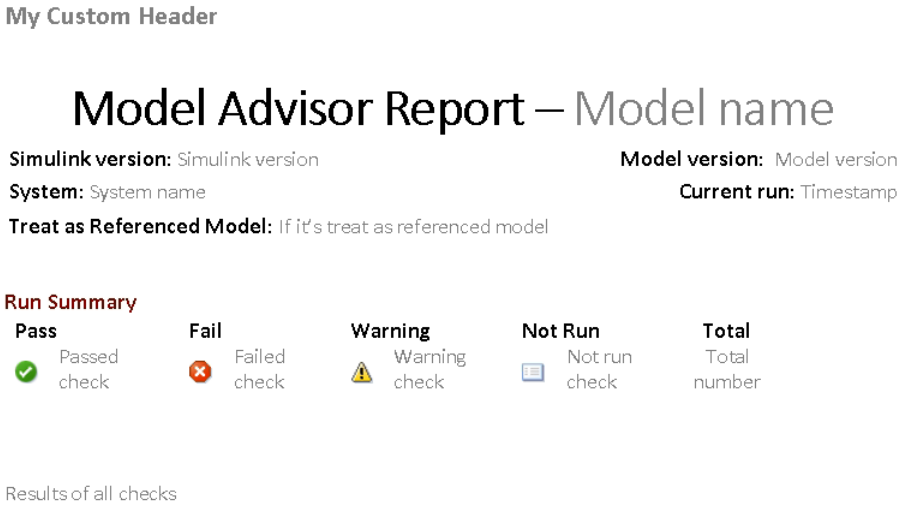
If you have a MATLAB Report Generator license, you can modify the default template that the Model Advisor uses to generate the report in PDF or Microsoft Word.

The default template contains fields 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 fields. When the Model Advisor generate the report, analysis information overrides the text that you enter in the template field.

Template Field	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.

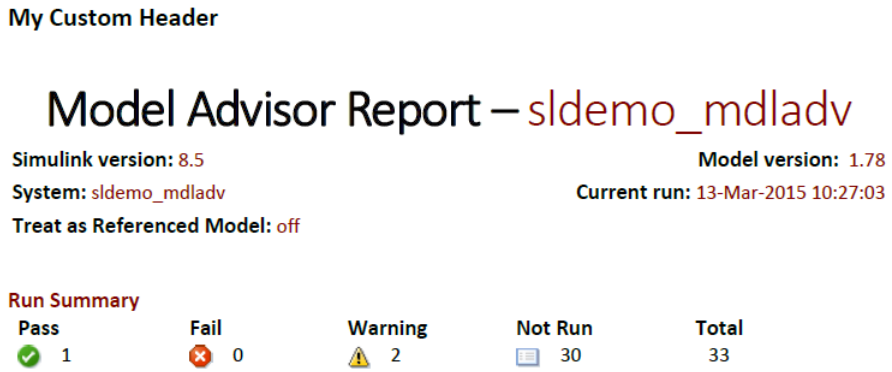


4. In the Model Advisor window Report pane, click **Generate Report**.

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 the custom header.



See Also

Related Examples

- “Save and View Model Advisor Check Reports” (Simulink)
- “Customize Microsoft Word Component Templates” (MATLAB Report Generator)
- “Run Model Advisor Checks and Review Results” on page 3-4

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, Variant Model 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:

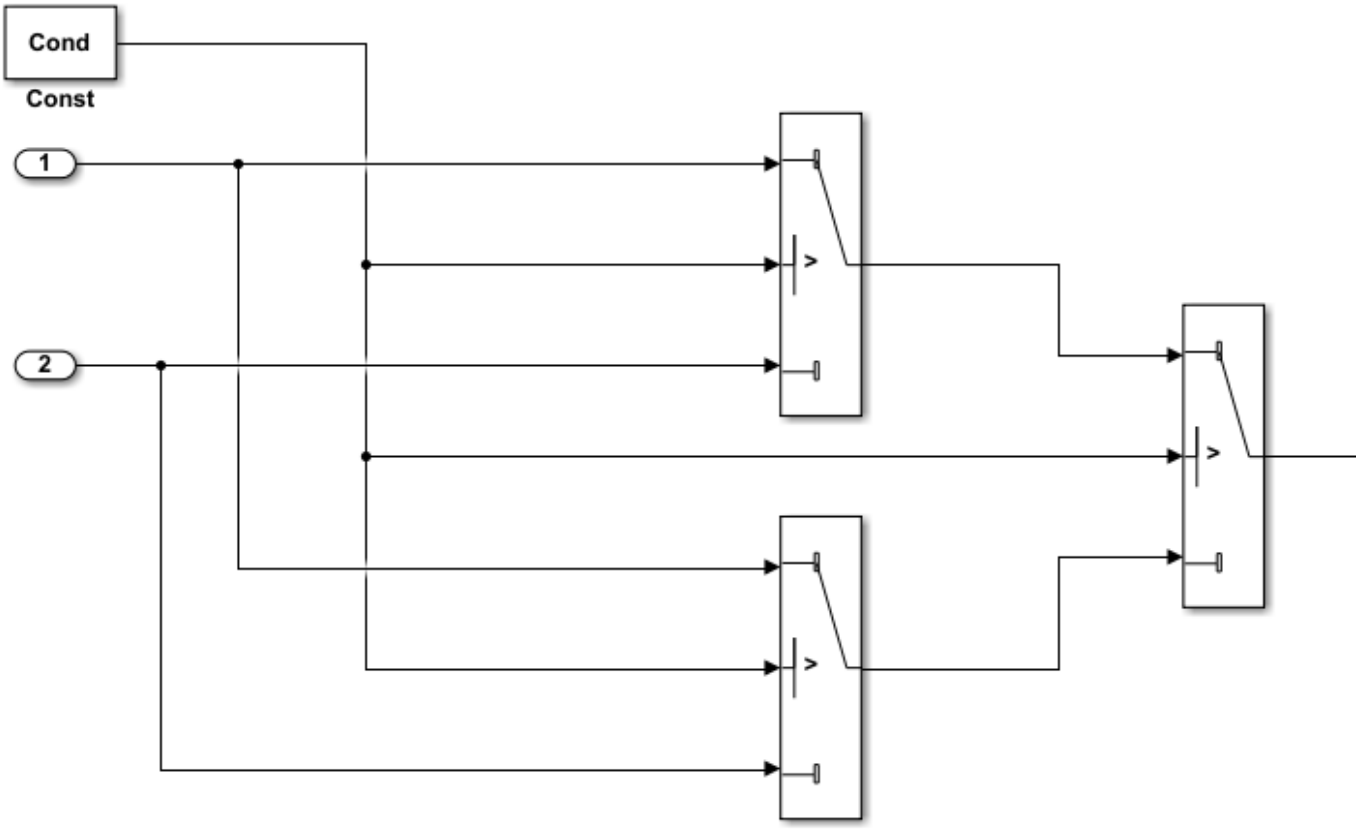
- If an If block connects to one or more If Action Subsystems and each one has one outport, replace this modeling pattern with a subsystem and a Variant Source block.
- If an If block connects to an If Action Subsystem that does not have an outport or has two or more outports, replace this modeling pattern with a Variant Subsystem block.
- If a Switch Case block connects to one or more Switch Case Action Subsystems and each one has one outport, 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 outport or has two or more outports, 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 `Simulink.Parameter` object of scalar type.
- Constant blocks in which the **Constant value** parameters are `Simulink.Parameter` objects of scalar type and some other combination of blocks that form a supported MATLAB expression. The MATLAB expressions in “Operators and Operands in Variant Condition Expressions” (Simulink) are supported except for bitwise operations.

Example Model

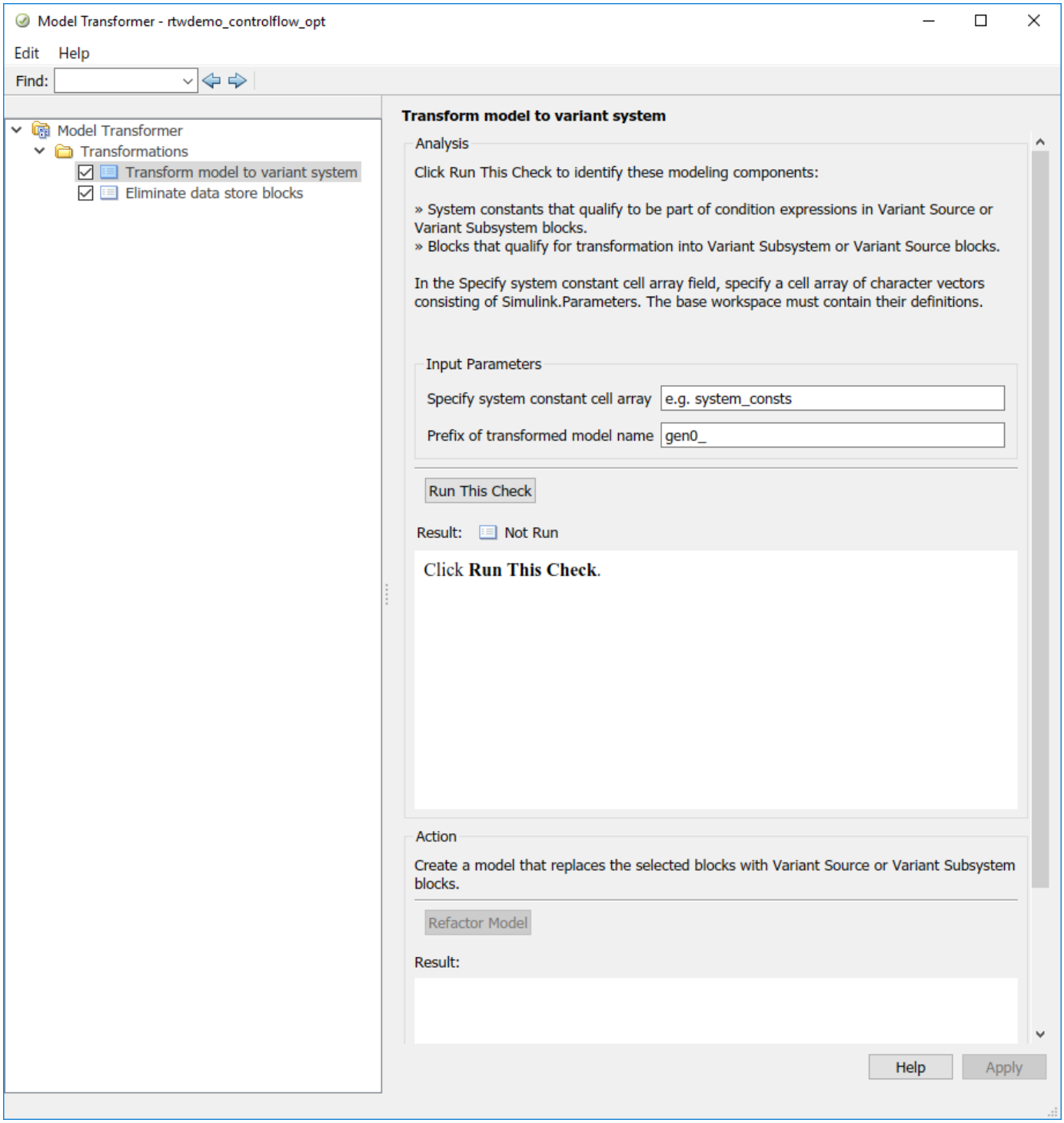
This example shows how to use the Model Transformer to transform a model into a variant system. The example uses the model `rtwdemo_controlflow_opt`. This model has three Switch blocks. The control input to these Switch blocks is the `Simulink.Parameter` `cond`. The Model Transformer dialog box and this example refer to `cond` as a system constant.



1. Open the model. In the Command Window, type `rtwdemo_controlflow_opt`.
2. 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.
3. Repeat step 2 for the Switch blocks `Switch1`, `Switch2`, and `Switch3`.
4. Save the model to your working folder.

Perform Variant Transform on Example Model

1. In the **Apps** tab, Open the Model Transformer by selecting **Model Transformer**. Or, in the Command Window, type:
   `mdltransformer('rtwdemo_controlflow_opt')`
2. Select the check “Transform the model to variant system”.



3 In the **Specify system constant cell array** field, you can specify a cell array of character vectors consisting of `Simulink.Parameters`. The base workspace must contain their definitions.

4 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`.

5. Select **Run This Check**. The Model Transformer lists system constants and blocks that qualify to be part of condition expressions in Variant Source or Variant Subsystem blocks. For the Model Transformer to list a system constant, it must be a `Simulink.Parameter` object of scalar type. For this example, `Cond` qualifies to part of a condition expression.
6. If you do not want one of the transformations to occur, you can clear the check box next to it.
7. Select **Refactor Model**. The Model Transformer provides a hyperlink to the transformed model and hyperlinks to the corresponding blocks in the original model and the transformed model.

   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`.
8. 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.
9. 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.

Model Transformation Limitations

The Model Transformer tool has these limitations:

- In order to run the Model Transformer on a model, you must be able to simulate the model.
- 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. This modeling pattern produces a warning and an excluded candidate message.
- The Model Transformer cannot perform a variant transformation for every modeling pattern. 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`.
- After you run one or more tasks, you cannot rerun the 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.

See Also

Related Examples

- “Variant Systems” (Simulink)

Enable Component Reuse by Using Clone Detection

Clones are modeling patterns that have identical block types and connections. The Clone Detector app identifies clones across referenced model boundaries. You can use the Clone Detector app to reuse components by creating library blocks from subsystem clones and replacing the clones with links to those library blocks. You can also use the tool to link to clones in an existing library.

Exact Clones and Similar Clones

There are two types of clones: exact clones and similar clones. Exact clones have identical block types, connections, and parameter values. Similar clones have identical block types and connections, but they can have different block parameter values. For example, the value of a Gain block can be different in similar clones but must be the same in exact clones.

Exact clones and similar clones can have these differences:

- Two clones can have a different sorted order.
- The length of signal lines and the location and size of blocks can be different if the block connections are the same.
- Blocks and signals can have different names.

To detect only exact clones, for each check in the Identify Modeling Clones tool, set the **Maximum Number of Unmatched Block Parameters** to `0`. Increasing this parameter value increases the number of similar clones that the tool can potentially detect.

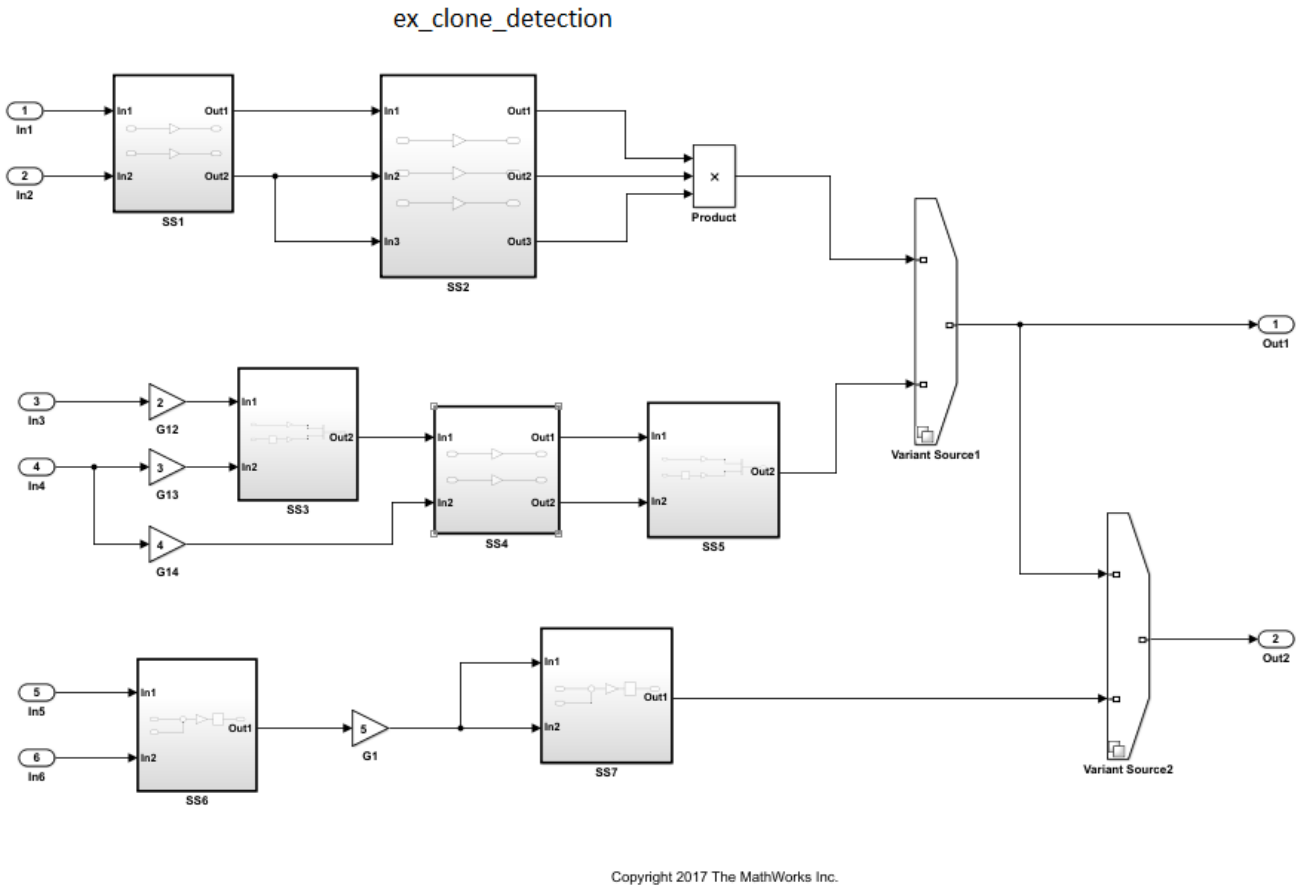
After you identify clones, you can replace them with links to library blocks. Similar clones link to masked library subsystems.

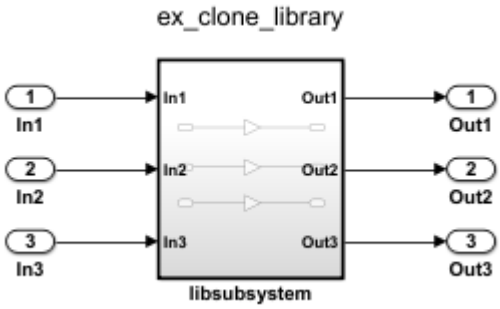
Identify Exact and Similar Clones

This example shows how to use the Clone Detector app to identify exact clones and similar clones, and then replace them with links to library blocks.

1 Open the model `ex_clone_detection`. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
ex_clone_detection
```





2 Save the model to your working folder. A model must be open to access the app.

3 On the **Apps** tab, click **Clone Detector**. Alternatively, on the MATLAB command line enter:

```
clonedetection("ex_clone_detection")
```

4 The app opens the Clone Detector tab. This example takes you through each section.

Set Up panes for Clone Detection

The app displays information on multiple panes. You can select three of the panes under the **View** menu. The panes are:

- **Help**. Select to access a help pane that contains an overview of the clone detection workflow.
- **Results**. Select to view the Clone Detection and Results pane.
- **Properties**. Select to view the Detected Clone Properties pane.

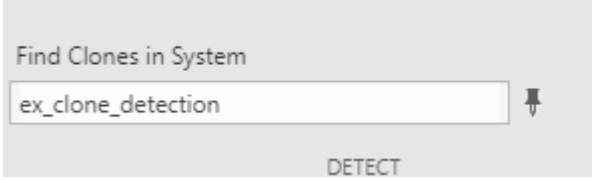
Set the Parameters for Clone Detection

You can set up the parameters for clone detection by using the **Settings** drop-down menu.

- Select **Ignore differences in > Signal Names** to identify and classify clones when the signal names are different.
- Select **Ignore differences in > Block Properties** to identify and classify clones when the block properties are different. For more information about block properties, see “Specify Block Properties” (Simulink).
- Click **Exclude Components** to access the **Exclude model references**, **Exclude library Links**, and **Exclude inactive and commented out regions** options. Enabling the **Exclude inactive and commented out regions** option identifies variable number clones because of Variant Source block in the model. For more information, see “Exclude subsystems and referenced models from clone detection”. Enabling the **Exclude model references** and **Exclude library Links** options will lead to identification of fewer clones, depending on the model.
- Click **Match Patterns with Libraries** and select an external library to look for clones. For more information, see “Identifying and Replacing Clones in Model Libraries” on page 3-31.
- The **Maximum number of unmatched block parameters** is `50` by default. This represents the number of parameters that can vary among subsystems and still be classified as similar clones. You may reduce this number to identify and classify fewer similar clones. Setting the value to zero, will identify only exact clones.

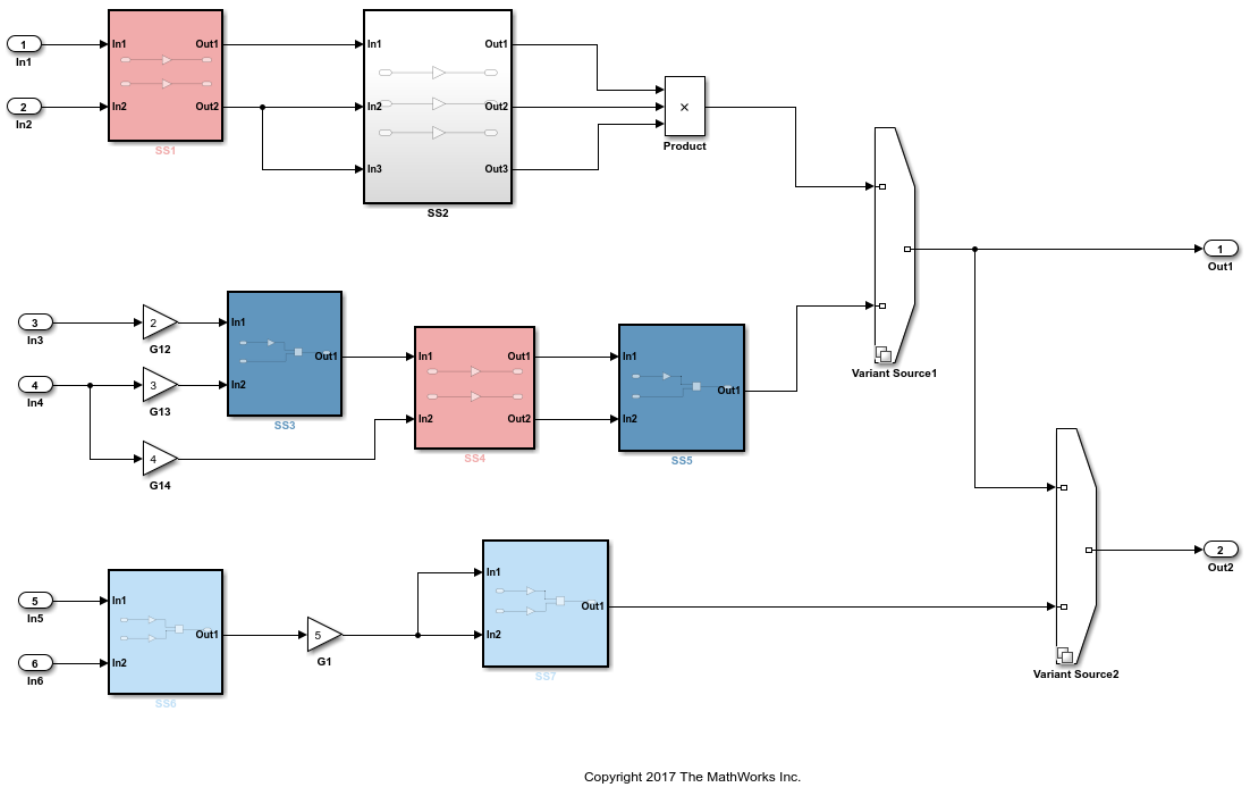
Identify Clones in the Model

1. In the **Detect** section, in the **Find Clones in System** tab, enter `ex_clone_detection` as the model to identify clones in. Toggle the pin to access other subsystems to identify clones in.



2. Click **Find Clones** to identify clones.

3 The color of the subsystems changes to reflect the similar and exact clones identified. The red highlighting represents exact clones and the different shades of blue highlighting represent similar clones.



Analyze the Clone Detection Results

After identifying clones, you can analyze the results of the clone detection and make changes to the model as necessary. To analyze the results:

1 In the **Clone Detection Action and Results** panel, on the **Logs** pane, click the hyperlink.

A new window opens the clone detection results with an integrated report on the identified clones, the types of clones, the parameters of detection, and the exclusions in the clone detection.

2 In the **Clone Detection Action and Results** pane, click the **Map Clone Groups to Library** tab.

A list of clone groups are displayed.

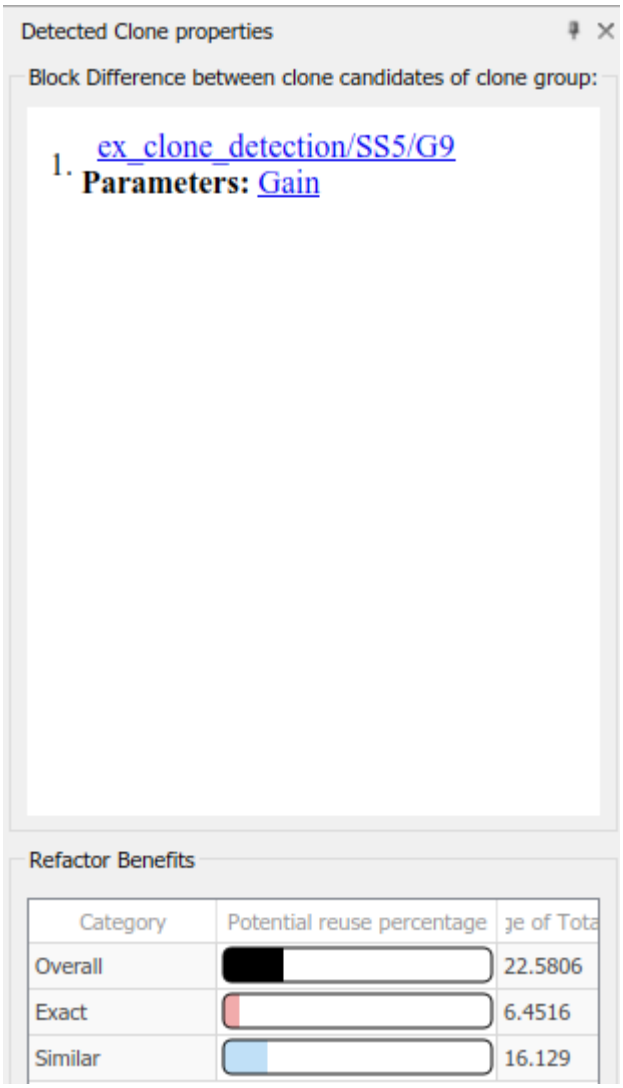
3 Click the > symbol next to `Exact Clone Group 1` to see all of the subsystems that are exact clones, the number of blocks, and the block differences. Repeat the same for `Similar Clone Group 1` and `Similar Clone Group 2`.

4 Click the **Model Hierarchy** tab. Click the hyperlinks to highlight the subsystems that are present in the model.

5 On the **Map Clone Groups to Library** tab, expand `Similar Clone Group 1` and click the **View Parameter Difference** hyperlink.

6. On the **Detected Clone Properties** panel, click the `ex_clone_detection/SS5/G9` hyperlink, which opens the gain block `G9` in the subsystem `SS5`, where you can access the parameter that are different from the baseline subsystem.
7. Change value of the gain parameter from `A` to `B` and click **Find Clones**. This will reclassify `Similar Clone Group 1` to `Exact Clone Group 2` because you resolved the difference in the subsystems and converted it into an exact clone.
8. Under the **Refactor Benefits** panel, you can consider the percentage of different types of clones present.

   In the **Clone Detection Actions and Results** pane, in the **Map Clone Groups to Library** tab, select the clones you would like to refactor. Select all the clone groups for refactoring to reduce `22.5806%` of the model reuse.



Replace Clones

1. You may use the default library name or change the name of the library file and its location on the **Map Clone Groups to Library** tab before replacing the clones.
2. Click **Replace Clones**.

The model is refactored and the clones are replaced with links to the `newLibraryFile` library file in your working directory.

3 You can restore the model to its original configuration with clones by clicking **Restore** button found in the clone detector log that was generated on the **Logs** tab of the **Clone Detector Actions and Results** pane.

Identifying and Replacing Clones in Model Libraries

1 Open the library `ex_clone_library`. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
ex_clone_library
```

2 Click **Settings > Match Patterns with Libraries** and select `ex_clone_library.slx`. Then click **Find Clones**.

Note Identifying and refactoring clones in external libraries must be done separately from the model. During model refactoring only exact clones within the libraries will be replaced library links.

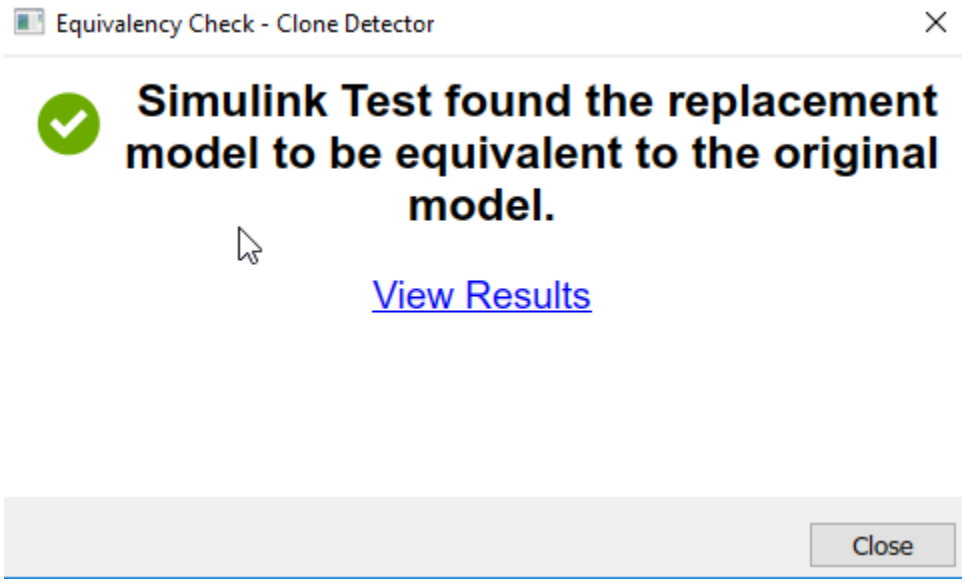
---

3 Click **Replace Clones**.

The model is refactored with the exact clones replaced.

Check the Equivalency of the Model

If you have a Simulink Test license, you can click **Check Equivalency**. A window opens and displays that the current model has been successfully refactored into an equivalent model.



See Also

Related Examples

- “Custom Libraries” (Simulink)
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)
- Clone Detector

Improve Model Readability by Eliminating Local Data Store Blocks

You can use the Model Transformer tool to improve model readability by replacing Data Store Memory, Data Store Read, and Data Store Write blocks with either a direct signal line, a Delay block, or a Merge block. For bus signals, the tool might also add Bus Creator or Bus Selector blocks as part of the replacement. Replacing these blocks improves model readability by making data dependency explicit. The Model Transformer creates a model with these replacements. The new model has the same functionality as the existing model.

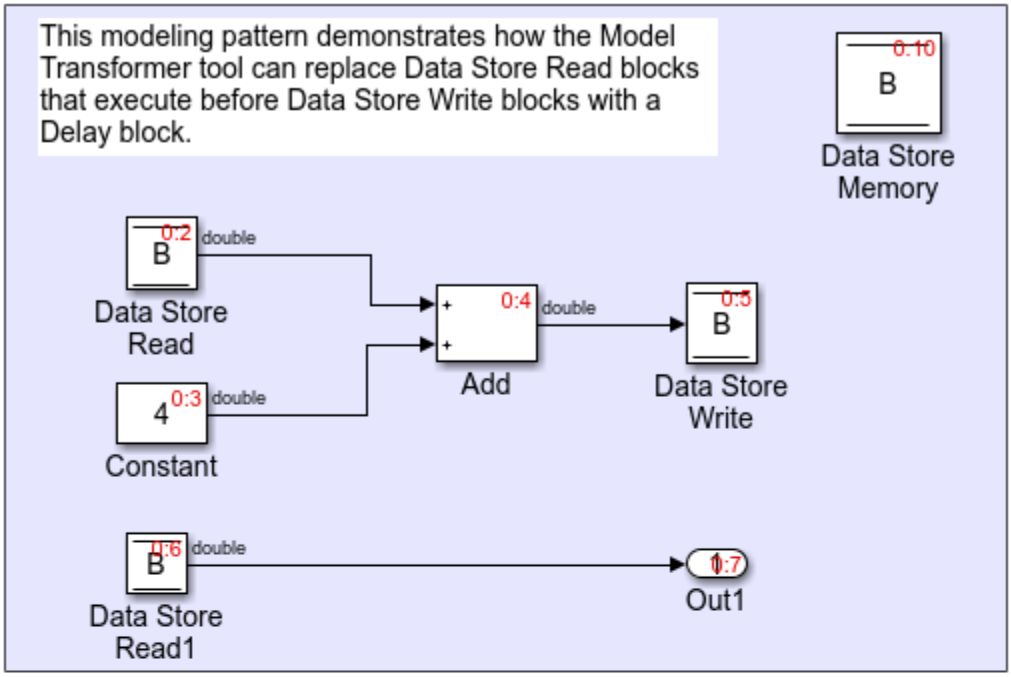
The Model Transformer can replace these data stores:

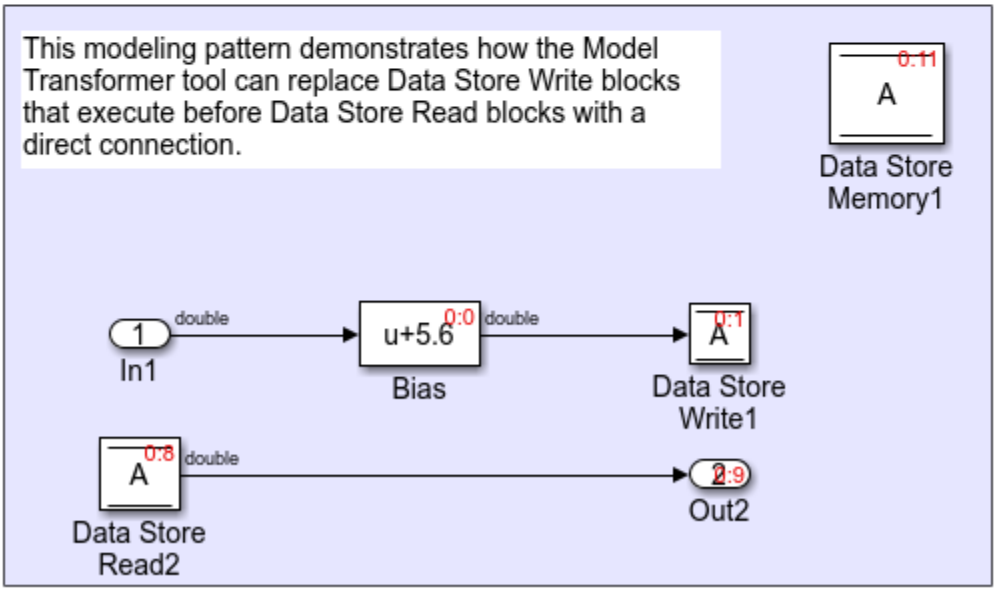
- For signals that are not buses, if a Data Store Read block executes before a Data Store Write block, the tool replaces these blocks with a Delay block.
- For signals that are not buses, if a Data Store Write block executes before a Data Store Read block, the tool replaces these blocks with a direct connection.
- For bus signals, if the write to bus elements executes before the read of the bus, the tool replaces the Data Store Read and Data Store Write blocks with a direct connection and a Bus Creator block.
- For bus signals, if the write to the bus executes before the read of bus elements, the tool replaces the Data Store Read and Data Store Write blocks with a direct connection and a Bus Selector block.
- For conditionally executed subsystems, the tool replaces the Data Store Read and Data Store Write blocks with a direct connection and a Merge block. For models in which a read/write pair crosses an If subsystem boundary and the Write block is inside the subsystem, the tool might also add an Else subsystem block.

The Model Transformer tool eliminates only local data stores that Data Store Memory blocks define. The tool does not eliminate global data stores. For the Data Store Memory block, on the **Signal Attributes** tab in the block parameters dialog box, you must clear the **Data store name must resolve to Simulink signal object** parameter.

Example Model

The model `ex_data_store_elimination` contains the two local data stores: `B` and `A`. For data store `B`, there are two Data Store Read blocks and one Data Store Write block. For data store `A`, there is one Data Store Write block and one Data Store Read block. The red numbers represent the sorted execution order.





Copyright 2017 The MathWorks, Inc.

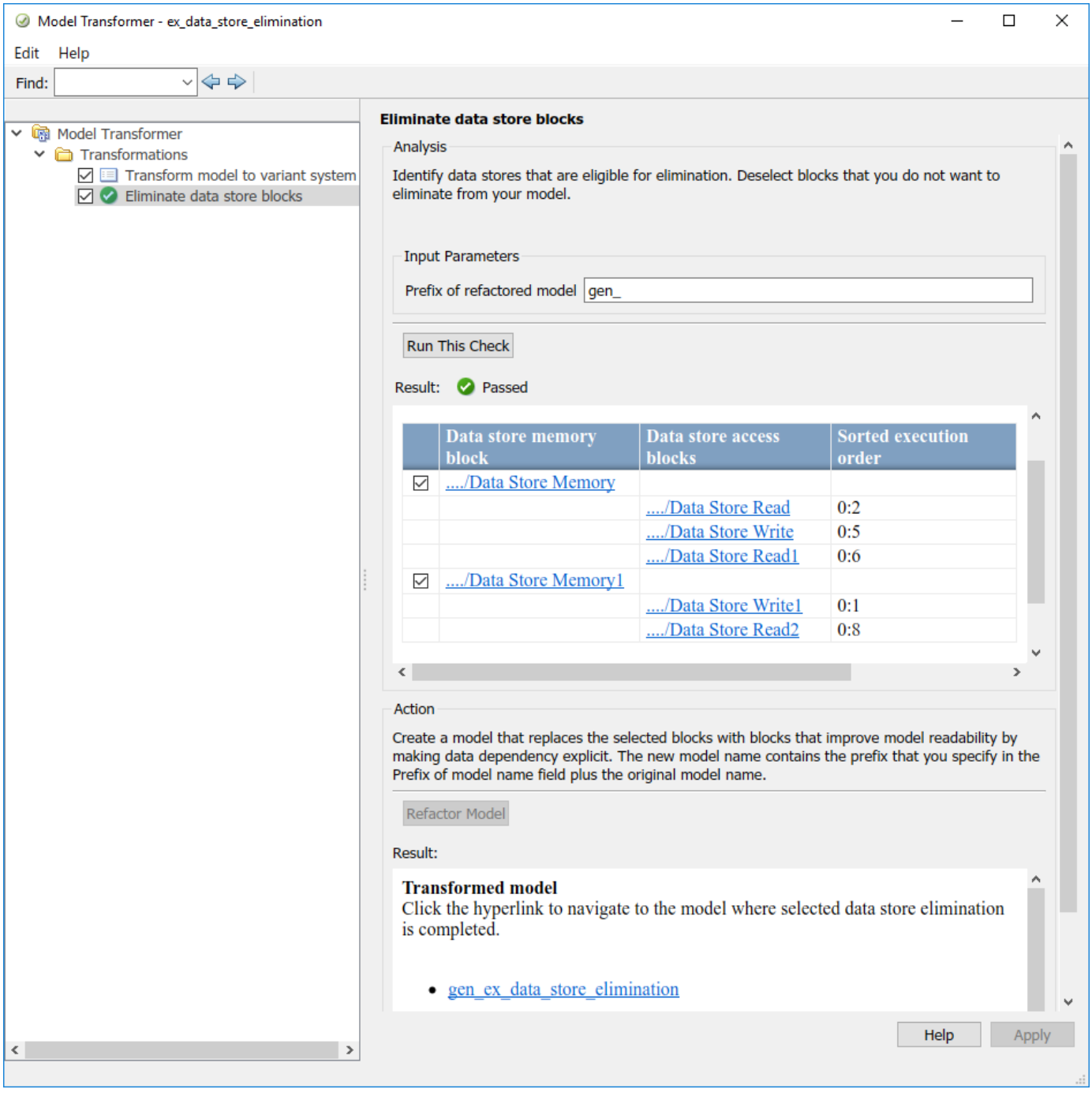
Replace Data Store Blocks

Identify data store blocks that qualify for replacement. Then, create a model that replaces these blocks with direct signal lines, Delay blocks, or Merge blocks.

1. Open the model `ex_data_store_elimination`. At the MATLAB command line, enter:

   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   ex_data_store_elimination
   ```

2. Save the model to your working folder.
3. In the Simulink Editor, from the **Analysis** menu, select **Refactor Model > Model Transformer**. To open the Model Transformer programmatically, at the MATLAB command prompt, type this command: `mdltransformer('ex_data_store_replacement')`.
4. In the **Transformations** folder, select the **Eliminate data store blocks** check.
5. In the **Prefix of refactored model** field, specify a prefix for the refactored model.
6. Click the **Run This Check** button. The top **Result** table contains hyperlinks to the Data Store Memory blocks and the corresponding Data Store Read and Data Store Write blocks that qualify for elimination.
7. Click the **Refactor Model** button. The bottom **Result** table contains a hyperlink to the new model. The tool creates an `m2m_ex_data_store_replacement` folder. This folder contains the `gen_ex_data_store_replacement.slx` model.



For local data store `A`, `gen_ex_bus_struct_in_code.slx` contains a Delay block in place of the Data Store Write block and a direct signal connection in place of the Data Store Read block. For local data store `B`, `gen_ex_bus_struct_in_code.slx` contains a direct signal connection from the Bias block to `Out2`.

Limitations

The Model Transformer does not replace Data Store Read and Write blocks that meet these conditions:

- They cross boundaries of conditionally executed subsystems such as Enabled, Triggered, or Function-Call subsystems and Stateflow Charts.
- They do not complete mutually exclusive branches of If-Action subsystems.
- They cross boundaries of variants.
- They have more than one input or output.
- They access part of an array.
- They execute at different rates.
- They are inside different instances of library subsystems and have a different relative execution order.

See Also

Related Examples

- “Refactor Models”
- “Data Stores” (Simulink)
- “Data Stores in Generated Code” (Simulink Coder)

Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks

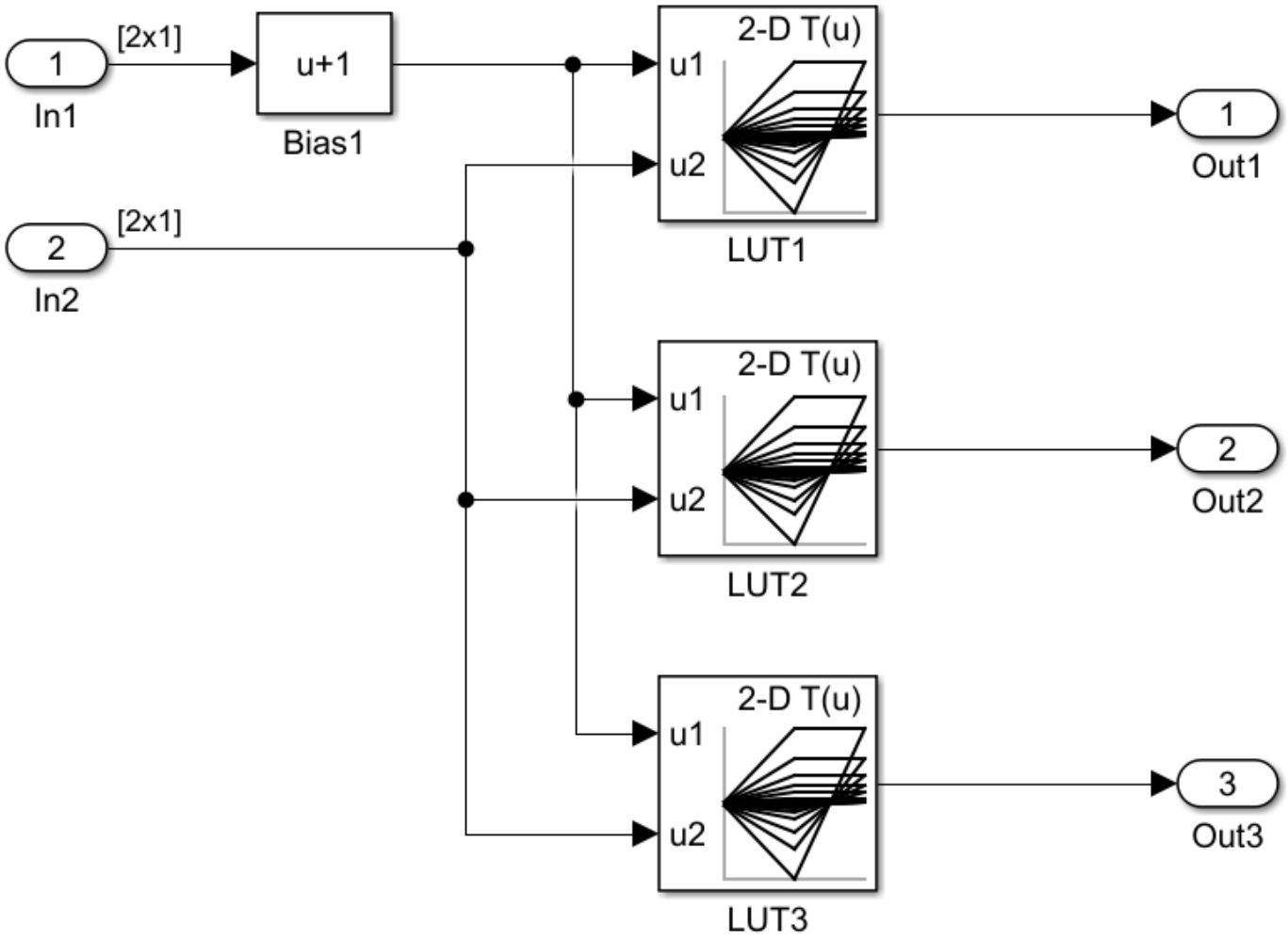
Improve the efficiency of your model simulation by using the Model Transformer tool to identify n-D Lookup Table blocks that qualify for transformation and replacing them with Interpolation blocks and shared Prelookup blocks. Eliminating the redundant Prelookup blocks improves the simulation speed for linear interpolations. The Model Transformer creates a model with these replacements blocks. This new model has the same functionality as the original model.

The Model Transformer can replace Lookup Table blocks that meet the following conditions:

- The same source drives the Lookup Table blocks.
- The Lookup Table blocks share the same breakpoint specification, values, and data types.
- The Lookup Table blocks share the same algorithm parameters in the block parameters dialog box.
- The Lookup Table blocks share the same data type for fractions parameters in the block parameters dialog box.

Example Model

The model `mLutOptim` contains three Lookup Table blocks: `LUT1`, `LUT2` and `LUT3`. The blocks are driven from the same input sources `In1` and `In2`.



Merge Prelookup Operation

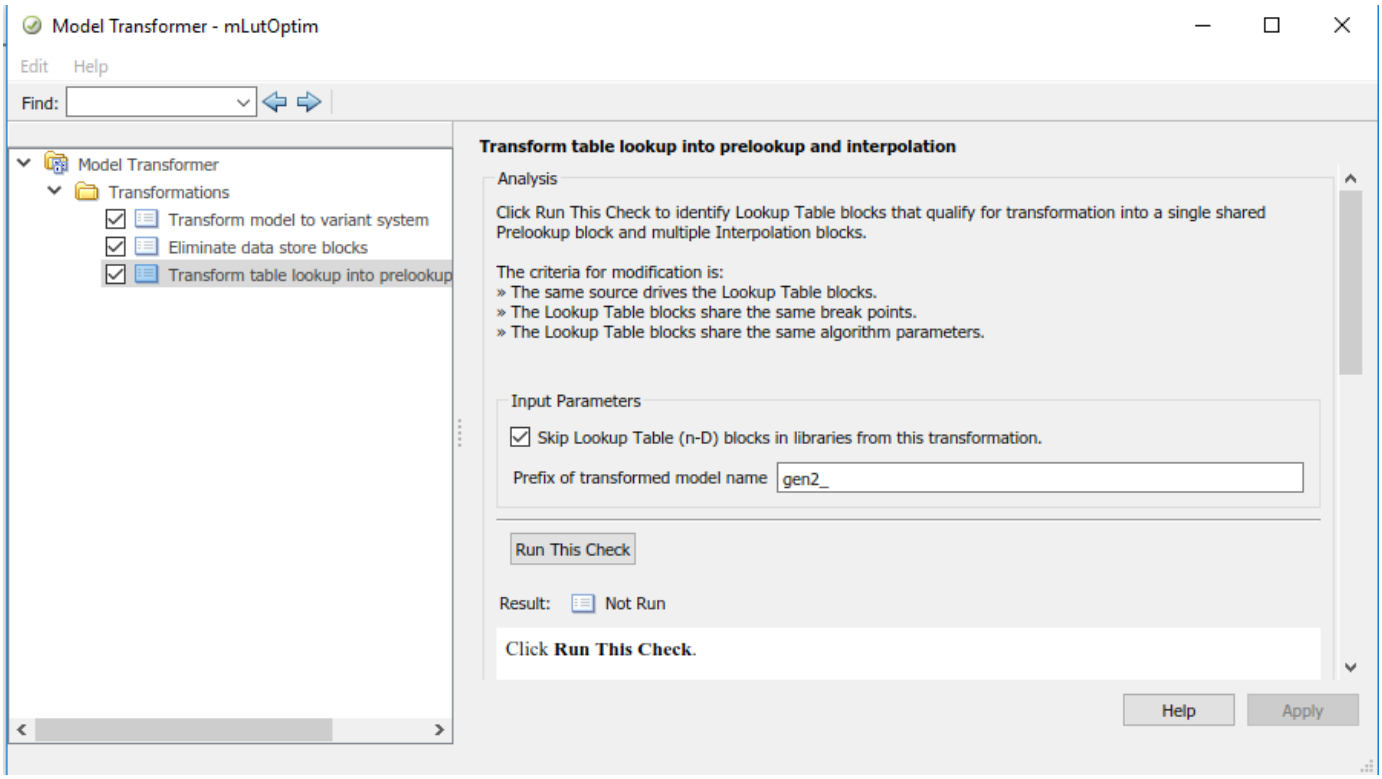
Identify n-D Lookup Table blocks that qualify for transformation and replace them with a single shared Prelookup block and multiple Interpolation blocks.

1. Open the model `mLutOptim`. At the MATLAB command line, enter:

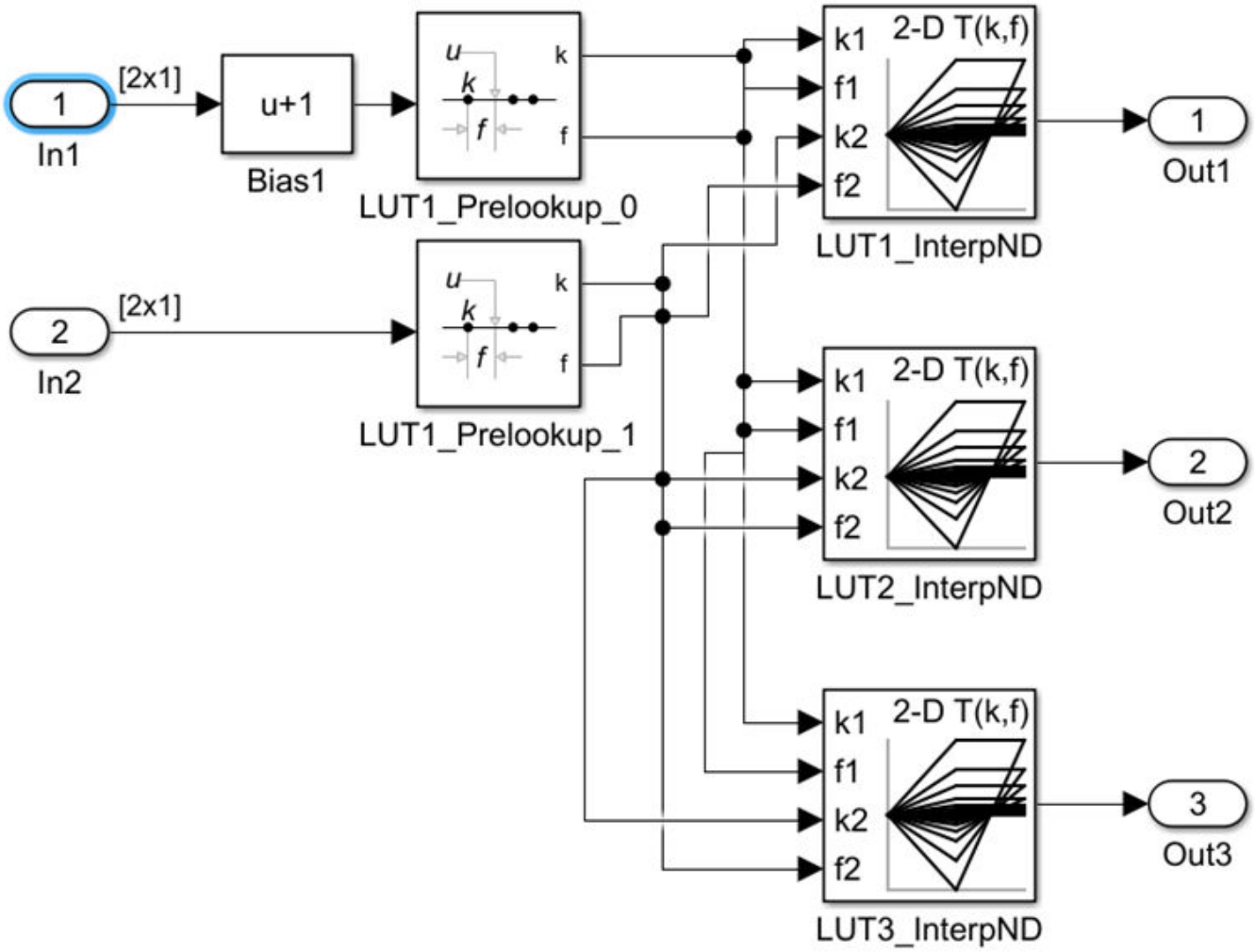
   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))mLutOptim
   ```

2. Save the model to your working folder.
3. In the Simulink Editor, from the **Analysis** menu, select **Refactor Model > Model Transformer**.
4. In the **Transformations** folder, select the “Transform table lookup into prelookup and interpolation” check.
5. Select the **Skip Lookup Table (n-D) blocks in the libraries from this transformation** option to avoid replacing Lookup Table blocks that are linked to a library.
6. In the **Prefix of refactored model** field, specify a prefix for the new refactored model.

7. Click the **Run This Check** button. The top **Result** table contains hyperlinks to the Lookup Table blocks and the corresponding input port indices.
8. Clear the **Candidate Groups** that you do not want to transform.
9. Click the **Refactor Model** button. The **Result** table contains a hyperlink to the new model. The table also contains hyperlinks to the shared Prelookup block and corresponding Interpolation blocks. Those blocks replaced the original Lookup Table blocks. The tool creates an `m2m_mLUTOptim` folder. This folder contains the new `gen_mLUTOptim.slx` model.



The Lookup Table blocks `LUT1`, `LUT2`, and `LUT3` of `gen_mLutOptim.slx` have two shared Prelookup table blocks, `LUT1_Prelookup_1` and `LUT1_Prelookup_2`, one for each data source. There are also three Interpolation blocks `LUT1_InterpND`, `LUT2_InterpND`, and `LUT3_InterpND` that replace the Lookup Table blocks.



Conditions and Limitations

The Model Transformer cannot replace Lookup Table blocks if:

- A Rate Transition block drives the Lookup Table blocks.
- The Lookup Table blocks are commented-out regions and inactive variants.
- The Lookup Table blocks are masked.
- The Output block's data type is set to `Inherit:Same as first input`.
- The Lookup Table block **Interpolation method** and **Extrapolation method** on the **Algorithm** pane of the block parameters dialog box is set to `Cubic spline`.
- The Lookup Table block **Input settings** on the **Algorithm** pane of the block parameters dialog box has **Use one input port for all input data** selected.

  The Lookup Table block **Code generation** on the **Algorithm** pane of the block parameters dialog box has **Support tunable table size in code generation** selected.

The Model Transformer tool does not replace Lookup Table blocks across the boundaries of Atomic subsystems, Referenced Models, and library-linked blocks.

See Also

Related Examples

- “Refactor Models”
- “Transform table lookup into prelookup and interpolation”

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.

To check compliance with DO standards, open the Model Advisor on page 3-4 and run the checks in **By Task > Modeling Standards for DO-178C/DO-331**.

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.

DO-178C/DO-331 Check
“Display model version information”
“Check for Discrete-Time Integrator blocks with initial condition uncertainty” (Simulink)
“Check root model Inport block specifications” (Simulink)
“Identify unconnected lines, input ports, and output ports” (Simulink)
“Check usage of tunable parameters in blocks”
“Check for Strong Data Typing with Simulink I/O”
“Check for blocks that have constraints on tunable parameters” (Simulink Coder)
“Identify questionable subsystem settings” (Embedded Coder)
Check bus signals treated as vectors
Check for potentially delayed function-call subsystem return values
Check usage of Merge blocks
“Check definition of Stateflow data”
Check usage of exclusive and default states in state machines
Identify disabled library links
Identify parameterized library links
Identify unresolved library links
Check for model reference configuration mismatch
Check for parameter tunability information ignored for referenced models
Identify requirement links that specify invalid locations within documents
Identify requirement links with missing documents
Identify requirement links with path type inconsistent with preferences
Identify selection-based links having descriptions that do not match their requirements document text
Check sample times and tasking mode
Check solver for code generation
Check the hardware implementation
Display bug reports for DO Qualification Kit
Display bug reports for Embedded Coder

DO-178C/DO-331 Check
Display bug reports for Polyspace Code Prover
Display bug reports for Polyspace Code Prover Server
Display bug reports for Polyspace Bug Finder
Display bug reports for Polyspace Bug Finder Server
Display bug reports for Simulink Code Inspector
Display bug reports for Simulink Report Generator
Display bug reports for Simulink Check
Display bug reports for Simulink Coverage
Display bug reports for Simulink Design Verifier
Display bug reports for Simulink Test
Display bug reports for Simulink Requirements
Display bug reports for Simulink

The following are the High-Integrity System Modeling checks that are applicable for the DO-178C/ DO-331 standards.

Model Checks for High Integrity Systems Modeling

You can check that your model or subsystem complies with selected aspects of the High Integrity System Model safety standard by running the Model Advisor.

To check compliance with High Integrity System Model standards, run the high-integrity checks from these Model Advisor folders:

- **By Task > Modeling Standards for DO-178C/DO-331 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 61508 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 62304 > High-Integrity Systems**
- **By Task > Modeling Standards for EN 50128 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 26262 > High-Integrity Systems**

For information on the High Integrity System Model Software Considerations in Airborne Systems and Equipment Certification and related standards, see Radio Technical Commission for Aeronautics (RTCA).

The table lists the High Integrity System Model checks and their corresponding modeling guidelines. For more information about the High-Integrity Modeling Guidelines, see “High-Integrity System Modeling” (Simulink).

High Integrity Systems Modeling Checks

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage of lookup table blocks	“hisl_0033: Usage of Lookup Table blocks” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check for inconsistent vector indexing methods	“hisl_0021: Consistent vector indexing method” (Simulink)
Check for variant blocks with 'Generate preprocessor conditionals' active	“hisl_0023: Verification of model and subsystem variants” (Simulink)
Check for root Inports with missing properties	“hisl_0024: Inport interface definition” (Simulink)
Check for Relational Operator blocks that equate floating-point types	“hisl_0017: Usage of blocks that compute relational operators (2)” (Simulink)
Check usage of Relational Operator blocks	“hisl_0016: Usage of blocks that compute relational operators” (Simulink)
Check usage of Logical Operator blocks	“hisl_0018: Usage of Logical Operator block” (Simulink)
Check usage of While Iterator blocks	“hisl_0006: Usage of While Iterator blocks” (Simulink)
Check sample time-dependent blocks	“hisl_0007: Usage of For Iterator or While Iterator subsystems” (Simulink)
Check usage of For Iterator blocks	“hisl_0008: Usage of For Iterator Blocks” (Simulink)
Check usage of If blocks and If Action Subsystem blocks	“hisl_0010: Usage of If blocks and If Action Subsystem blocks” (Simulink)
Check usage Switch Case blocks and Switch Case Action Subsystem blocks	“hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks” (Simulink)
Check safety-related optimization settings for logic signals	“hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)” (Simulink)
Check safety-related block reduction optimization settings	“hisl_0046: Configuration Parameters > Simulation Target > Block reduction” (Simulink)
Check safety-related optimization settings for application lifespan	“hisl_0048: Configuration Parameters > Math and Data Types > Application lifespan (days)” (Simulink)
Check safety-related optimization settings for data initialization	“hisl_0052: Configuration Parameters > Code Generation > Optimization > Data initialization” (Simulink)
Check safety-related optimization settings for data type conversions	“hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values” (Simulink)
Check safety-related optimization settings for division arithmetic exceptions	“hisl_0054: Configuration Parameters > Code Generation > Optimization > Remove code that protects against division arithmetic exceptions” (Simulink)
Check safety-related code generation settings for comments	“hisl_0038: Configuration Parameters > Code Generation > Comments” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related code generation interface settings	“hisl_0039: Configuration Parameters > Code Generation > Interface” (Simulink)
Check safety-related code generation settings for code style	“hisl_0047: Configuration Parameters > Code Generation > Code Style” (Simulink)
Check safety-related code generation identifier settings	“hisl_0049: Configuration Parameters > Code Generation > Identifiers” (Simulink)
Check usage of Abs blocks	“hisl_0001: Usage of Abs block” (Simulink)
Check usage of Math Function blocks (rem and reciprocal functions)	“hisl_0002: Usage of Math Function blocks (rem and reciprocal)” (Simulink)
Check usage of Math Function blocks (log and log10 functions)	“hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)” (Simulink)
Check usage of Assignment blocks	“hisl_0029: Usage of Assignment blocks” (Simulink)
Check usage of Signal Routing blocks	“hisl_0034: Usage of Signal Routing blocks” (Simulink)
Check for root Inports with missing range definitions	“hisl_0025: Design min/max specification of input interfaces” (Simulink)
Check for root Outports with missing range definitions	“hisl_0026: Design min/max specification of output interfaces” (Simulink)
Check state machine type of Stateflow charts	“hisf_0001: State Machine Type” (Simulink)
Check Stateflow charts for transition paths that cross parallel state boundaries	“hisf_0013: Usage of transition paths (crossing parallel state boundaries)” (Simulink)
Check Stateflow charts for ordering of states and transitions	“hisf_0002: User-specified state/transition execution order” (Simulink)
Check Stateflow debugging options	“hisf_0011: Stateflow debugging settings” (Simulink)
Check Stateflow charts for uniquely defined data objects	“hisl_0061: Unique identifiers for clarity” (Simulink)
Check Stateflow charts for strong data typing	“hisf_0015: Strong data typing (casting variables and parameters in expressions)” (Simulink)
Check usage of shift operations for Stateflow data	“hisf_0064: Shift operations for Stateflow data to improve code compliance” (Simulink)
Check assignment operations in Stateflow charts	“hisf_0065: Type cast operations in Stateflow to improve code compliance” (Simulink)
Check Stateflow charts for unary operators	“hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance” (Simulink)
Check for Strong Data Typing with Simulink I/O	“hisf_0009: Strong data typing (Simulink and Stateflow boundary)” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check for MATLAB Function interfaces with inherited properties	“himl_0002: Strong data typing at MATLAB function boundaries” (Simulink)
Check MATLAB Function metrics	“himl_0003: Limitation of MATLAB function complexity” (Simulink)
Check MATLAB Code Analyzer messages	“himl_0004: MATLAB Code Analyzer recommendations for code generation” (Simulink)
Check safety-related model referencing settings	“hisl_0037: Configuration Parameters > Model Referencing” (Simulink)
Check safety-related diagnostic settings for solvers	“hisl_0043: Configuration Parameters > Diagnostics > Solver” (Simulink)
Check safety-related solver settings for simulation time	“hisl_0040: Configuration Parameters > Solver > Simulation time” (Simulink)
Check safety-related solver settings for solver options	“hisl_0041: Configuration Parameters > Solver > Solver options” (Simulink)
Check safety-related solver settings for tasking and sample-time	“hisl_0042: Configuration Parameters > Solver > Tasking and sample time options” (Simulink)
Check safety-related diagnostic settings for sample time	“hisl_0044: Configuration Parameters > Diagnostics > Sample Time” (Simulink)
Check safety-related diagnostic settings for parameters	“hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters” (Simulink)
Check safety-related diagnostic settings for data used for debugging	“hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging” (Simulink)
Check safety-related diagnostic settings for data store memory	“hisl_0013: Usage of data store blocks” (Simulink)
Check safety-related diagnostic settings for type conversions	“hisl_0309: Configuration Parameters > Diagnostics > Type Conversion” (Simulink)
Check safety-related diagnostic settings for signal connectivity	“hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals” (Simulink)
Check safety-related diagnostic settings for bus connectivity	“hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses” (Simulink)
Check safety-related diagnostic settings that apply to function-call connectivity	“hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls” (Simulink)
Check safety-related diagnostic settings for compatibility	“hisl_0301: Configuration Parameters > Diagnostics > Compatibility” (Simulink)
Check safety-related diagnostic settings for model initialization	“hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization” (Simulink)
Check safety-related diagnostic settings for model referencing	“hisl_0310: Configuration Parameters > Diagnostics > Model Referencing” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for saving	“hisl_0036: Configuration Parameters > Diagnostics > Saving” (Simulink)
Check safety-related diagnostic settings for Merge blocks	“hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks” (Simulink)
Check safety-related diagnostic settings for Stateflow	“hisl_0311: Configuration Parameters > Diagnostics > Stateflow” (Simulink)
Check safety-related optimization settings for Loop unrolling threshold	“hisl_0051: Configuration Parameters > Code Generation > Optimization > Loop unrolling threshold” (Simulink)
Check model object names	“hisl_0032: Model object names” (Simulink)
Check for model elements that do not link to requirements	“hisl_0070: Placement of requirement links in a model” (Simulink)
Check for inappropriate use of transition paths	“hisf_0014: Usage of transition paths (passing through states)” (Simulink)
Check usage of Bitwise Operator block	“hisl_0019: Usage of bitwise operations” (Simulink)
Check data types for blocks with index signals	“hisl_0022: Data type selection for index signals” (Simulink)
Check model file name	“hisl_0031: Model file names” (Simulink)
Check if/elseif/else patterns in MATLAB Function blocks	“himl_0006: MATLAB code if / elseif / else patterns” (Simulink)
Check switch statements in MATLAB Function blocks	“himl_0007: MATLAB code switch / case / otherwise patterns” (Simulink)
Check global variables in graphical functions	“hisl_0062: Global variables in graphical functions” (Simulink)
Check for length of user-defined object names	“hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance” (Simulink)
Check usage of Merge blocks	“hisl_0015: Usage of Merge blocks” (Simulink)
Check usage of conditionally executed subsystems	“hisl_0012: Usage of conditionally executed subsystems” (Simulink)
Check usage of standardized MATLAB function headers	“himl_0001: Usage of standardized MATLAB function headers” (Simulink)
Check usage of relational operators in MATLAB Function blocks	“himl_0008: MATLAB code relational operator data types” (Simulink)
Check usage of equality operators in MATLAB Function blocks	“himl_0009: MATLAB code with equal / not equal relational operators” (Simulink)
Check usage of logical operators and functions in MATLAB Function blocks	“himl_0010: MATLAB code with logical operators and functions” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check type and size of conditional expressions	“himl_0011: Data type and size of condition expressions” (Simulink)
Check naming of ports in Stateflow charts	“hisf_0016: Stateflow port names” (Simulink)
Check scoping of Stateflow data objects	“hisf_0017: Stateflow data object scoping” (Simulink)
Check usage of Gain blocks	“hisl_0066: Usage of Gain blocks” (Simulink)
Check usage of bitwise operations in Stateflow charts	“hisf_0003: Usage of bitwise operations” (Simulink)
Check data type of loop control variables	“hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance” (Simulink)
Check configuration parameters for MISRA C:2012	“hisl_0060: Configuration parameters that improve MISRA C:2012 compliance” (Simulink)
Check for blocks not recommended for C/C++ production code deployment Check for blocks not recommended for MISRA C:2012	“hisl_0020: Blocks not recommended for MISRA C:2012 compliance” (Simulink)
Check safety-related optimization settings for specified minimum and maximum values	“hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values” (Simulink)

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” on page 3-4

Model Checks for DO-254 Standard Compliance

You can check that your model or subsystem complies with selected aspects of the DO-254 safety standard by running the Model Advisor.

To check compliance with DO standards, open the Model Advisor on page 3-4 and run the checks in **By Task > Modeling Standards for DO-254**.

For information on the DO-254 Software Considerations in Airborne Systems and Equipment Certification and related standards, see Radio Technical Commission for Aeronautics (RTCA).

The table below lists the DO-254 checks.

DO-254 Checks
Display model version information
Identify disabled library links
Identify parameterized library links
Identify unresolved library links
Check for model reference configuration mismatch
Identify requirement links that specify invalid locations within documents
Identify requirement links with missing documents
Identify requirement links with path type inconsistent with preferences
Identify selection-based links having descriptions that do not match their requirements document text

The following are the High-Integrity System Modeling checks that are applicable for the DO-178C/ DO-331 standards.

Model Checks for High Integrity Systems Modeling

You can check that your model or subsystem complies with selected aspects of the High Integrity System Model safety standard by running the Model Advisor.

To check compliance with High Integrity System Model standards, run the high-integrity checks from these Model Advisor folders:

- **By Task > Modeling Standards for DO-254 > High-Integrity Systems**
- **By Task > Modeling Standards for DO-178C/DO-331 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 61508 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 62304 > High-Integrity Systems**
- **By Task > Modeling Standards for EN 50128 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 26262 > High-Integrity Systems**

For information on the High Integrity System Model Software Considerations in Airborne Systems and Equipment Certification and related standards, see Radio Technical Commission for Aeronautics (RTCA).

The table below lists the High Integrity System Model checks and their corresponding modeling guidelines that support DO-254 Safety Standard. For more information about the High-Integrity Modeling Guidelines, see “High-Integrity System Modeling” (Simulink).

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
Check for inconsistent vector indexing methods	“hisl_0021: Consistent vector indexing method” (Simulink)
Check for variant blocks with 'Generate preprocessor conditionals' active	“hisl_0023: Verification of model and subsystem variants” (Simulink)
Check for root Inports with missing properties	“hisl_0024: Inport interface definition” (Simulink)
Check for Relational Operator blocks that equate floating-point types	“hisl_0017: Usage of blocks that compute relational operators (2)” (Simulink)
Check usage of Relational Operator blocks	“hisl_0016: Usage of blocks that compute relational operators” (Simulink)
Check usage of Logical Operator blocks	“hisl_0018: Usage of Logical Operator block” (Simulink)
Check sample time-dependent blocks	“hisl_0007: Usage of For Iterator or While Iterator subsystems” (Simulink)
Check safety-related block reduction optimization settings	“hisl_0046: Configuration Parameters > Simulation Target > Block reduction” (Simulink)
Check usage of Abs blocks	“hisl_0001: Usage of Abs block” (Simulink)
Check usage of Assignment blocks	“hisl_0029: Usage of Assignment blocks” (Simulink)
Check for root Inports with missing range definitions	“hisl_0025: Design min/max specification of input interfaces” (Simulink)
Check for root Outports with missing range definitions	“hisl_0026: Design min/max specification of output interfaces” (Simulink)
Check Stateflow charts for transition paths that cross parallel state boundaries	“hisf_0013: Usage of transition paths (crossing parallel state boundaries)” (Simulink)
Check Stateflow charts for ordering of states and transitions	“hisf_0002: User-specified state/transition execution order” (Simulink)
Check Stateflow debugging options	“hisf_0011: Stateflow debugging settings” (Simulink)
Check Stateflow charts for uniquely defined data objects	“hisl_0061: Unique identifiers for clarity” (Simulink)
Check usage of shift operations for Stateflow data	“hisf_0064: Shift operations for Stateflow data to improve code compliance” (Simulink)
Check Stateflow charts for unary operators	“hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance” (Simulink)
Check for Strong Data Typing with Simulink I/O	“hisf_0009: Strong data typing (Simulink and Stateflow boundary)” (Simulink)

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
Check MATLAB Code Analyzer messages	“himl_0004: MATLAB Code Analyzer recommendations for code generation” (Simulink)
Check safety-related model referencing settings	“hisl_0037: Configuration Parameters > Model Referencing” (Simulink)
Check safety-related diagnostic settings for parameters	“hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters” (Simulink)
Check safety-related diagnostic settings for type conversions	“hisl_0309: Configuration Parameters > Diagnostics > Type Conversion” (Simulink)
Check safety-related diagnostic settings for signal connectivity	“hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals” (Simulink)
Check safety-related diagnostic settings for bus connectivity	“hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses” (Simulink)
Check safety-related diagnostic settings for model initialization	“hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization” (Simulink)
Check safety-related diagnostic settings for model referencing	“hisl_0310: Configuration Parameters > Diagnostics > Model Referencing” (Simulink)
Check safety-related diagnostic settings for saving	“hisl_0036: Configuration Parameters > Diagnostics > Saving” (Simulink)
Check safety-related diagnostic settings for Stateflow	“hisl_0311: Configuration Parameters > Diagnostics > Stateflow” (Simulink)
Check model object names	“hisl_0032: Model object names” (Simulink)
Check for model elements that do not link to requirements	“hisl_0070: Placement of requirement links in a model” (Simulink)
Check for inappropriate use of transition paths	“hisf_0014: Usage of transition paths (passing through states)” (Simulink)
Check usage of Bitwise Operator block	“hisl_0019: Usage of bitwise operations” (Simulink)
Check data types for blocks with index signals	“hisl_0022: Data type selection for index signals” (Simulink)
Check model file name	“hisl_0031: Model file names” (Simulink)
Check if/elseif/else patterns in MATLAB Function blocks	“himl_0006: MATLAB code if / elseif / else patterns” (Simulink)
Check switch statements in MATLAB Function blocks	“himl_0007: MATLAB code switch / case / otherwise patterns” (Simulink)
Check global variables in graphical functions	“hisl_0062: Global variables in graphical functions” (Simulink)
Check for length of user-defined object names	“hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance” (Simulink)

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
Check usage of conditionally executed subsystems	“hisl_0012: Usage of conditionally executed subsystems” (Simulink)
Check usage of standardized MATLAB function headers	“himl_0001: Usage of standardized MATLAB function headers” (Simulink)
Check usage of relational operators in MATLAB Function blocks	“himl_0008: MATLAB code relational operator data types” (Simulink)
Check usage of equality operators in MATLAB Function blocks	“himl_0009: MATLAB code with equal / not equal relational operators” (Simulink)
Check usage of logical operators and functions in MATLAB Function blocks	“himl_0010: MATLAB code with logical operators and functions” (Simulink)
Check naming of ports in Stateflow charts	“hisf_0016: Stateflow port names” (Simulink)
Check scoping of Stateflow data objects	“hisf_0017: Stateflow data object scoping” (Simulink)
Check usage of Gain blocks	“hisl_0066: Usage of Gain blocks” (Simulink)
Check usage of bitwise operations in Stateflow charts	“hisf_0003: Usage of bitwise operations” (Simulink)
Check data type of loop control variables	“hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance” (Simulink)

HDL Code Advisor Checks

The HDL Code Advisor and the Model Advisor checks in HDL Coder verify and update your Simulink model or subsystem for compatibility with HDL code generation. The Code Advisor has checks for:

- Model configuration settings
- Ports and Subsystem settings
- Blocks and block settings
- Native Floating Point support
- Industry standard guidelines

The following table lists the HDL Code Advisor checks that are supported by DO-254 Safety Standards:

HDL Code Advisor Checks	Description
“Check for infinite and continuous sample time sources” (HDL Coder)	Check source blocks with continuous sample time.
“Check for unsupported blocks” (HDL Coder)	Check for unsupported blocks for HDL code generation.
“Check for large matrix operations” (HDL Coder)	Check for large matrix operations.
“Identify unconnected lines, input ports, and output ports” (Simulink)	Check for unconnected lines or ports.
“Identify disabled library links” (Simulink)	Search model for disabled library links.

HDL Code Advisor Checks	Description
“Identify unresolved library links” (Simulink)	Search the model for unresolved library links, where the specified library block cannot be found.
“Check for MATLAB Function block settings” (HDL Coder)	Check HDL compatible settings for MATLAB Function blocks.
“Check for Stateflow chart settings” (HDL Coder)	Check HDL compatible settings for Stateflow Chart blocks.
“Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition” (Simulink)	Identify Delay, Unit Delay, or Zero-Order Hold blocks that are used for rate transition. Replace these blocks with actual Rate Transition blocks.
“Check for unsupported storage class for signal objects” (HDL Coder)	Check whether signal object storage class is `'ExportedGlobal'` or `'ImportedExtern'` or `'ImportedExternPointer'`
“Check VHDL file extension” (HDL Coder)	Check file extensions of VHDL files containing entities.
“Check naming conventions” (HDL Coder)	Check standard keywords used by EDA tools.
“Check top-level subsystem/port names” (HDL Coder)	Check top-level module/entity and port names.
“Check module/entity names” (HDL Coder)	Check module/entity names.
“Check signal and port names” (HDL Coder)	Check signal and port name lengths.
“Check package file names” (HDL Coder)	Check file name containing packages.
“Check generics” (HDL Coder)	Check generics at top-level subsystem.
“Check clock, reset, and enable signals” (HDL Coder)	Check naming convention for clock, reset, and enable signals.
“Check architecture name” (HDL Coder)	Check VHDL architecture name in the generated HDL code.
“Check entity and architecture” (HDL Coder)	Check whether the VHDL entity and architecture are described in the same file.
“Check clock settings” (HDL Coder)	Check constraints on clock signals.
“Check model for foreign characters” (Simulink)	Search the model for unresolved library links, where the specified library block cannot be found.
“Check for global reset setting for Xilinx and Altera devices” (HDL Coder)	Check asynchronous reset setting for Altera® devices and synchronous reset setting for Xilinx® devices.
“Check inline configurations setting” (HDL Coder)	Check whether you have `InlineConfigurations` enabled.
“Check algebraic loops” (HDL Coder)	Check model for algebraic loops.
“Check for visualization settings” (HDL Coder)	Check model for display settings: port data types and sample time color coding.
“Check delay balancing setting” (HDL Coder)	Check Balance Delays is enabled.

HDL Code Advisor Checks	Description
“Check for safe model parameters” (HDL Coder)	Check for model parameters set up for HDL code generation.
“Check for double datatypes in the model with Native Floating Point” (HDL Coder)	Check for `double` data types in the model.
“Check for Data Type Conversion blocks with incompatible settings” (HDL Coder)	Check conversion mode of Data Type Conversion blocks.
“Check for HDL Reciprocal block usage” (HDL Coder)	Check HDL Reciprocal blocks are not using floating point types.
“Check for Relational Operator block usage” (HDL Coder)	Check Relational Operator blocks which use floating point types have boolean outputs.
“Check for unsupported blocks with Native Floating Point” (HDL Coder)	Check for unsupported blocks with native floating-point.
“Check for blocks with nonzero output latency” (HDL Coder)	Check for blocks that have nonzero output latency with native floating-point.
“Check blocks with nonzero ulp error” (HDL Coder)	Check for blocks that have nonzero ulp error with native floating-point.
“Check for single datatypes in the model” (HDL Coder)	Check for `single` data types in the model.
“Check initial conditions of enabled and triggered subsystems” (HDL Coder)	Check for initial condition of enabled and triggered subsystems.
“Check for invalid top level subsystem” (HDL Coder)	Check for subsystems that cannot be at the top level for HDL code generation.

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” on page 3-4

Model Checks for MAB and JMAAB Compliance

You can use the Model Advisor to verify that your model or subsystem complies with the MathWorks Advisory Board (MAB) and Japan MATLAB Automotive Advisory Board modeling (JMAAB) guidelines.

Accessing the MAB and JMAAB Model Advisor Checks

To open the Model Advisor, in the Simulink editor, click the **Modeling** tab and select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Select the model or system that you want to review and click **OK**.

The MAB checks are available from these Model Advisor folders:

- **By Product > Simulink Check > Modeling Standards > MAB Checks**
- **By Task > Modeling Standards for MAB**

The JMAAB checks are available from these Model Advisor folders:

- **By Product > Simulink Check > Modeling Standards > JMAAB Checks**
- **By Task > Modeling Standards for JMAAB**

For information on using the Model Advisor, see “Run Model Advisor Checks and Review Results” on page 3-4.

Modeling Guidelines and Model Advisor Checks for MAB and JMAAB

This table identifies the MathWorks Advisor Board (MAB) guidelines, the corresponding Japan MathWorksAutomotive Advisor Board (JMAAB) guidelines, and the Model Advisor check that you can use to verify compliance with these guidelines. Additional information in the tables includes:

- **MAB Modeling Guideline** — Links to the MathWorks Advisory Board (MAB) guideline.
- **Corresponding JMAAB Modeling Guideline** — Identifies the JMAAB modelling guideline that corresponds to the MAB guideline. To review the guideline, see *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow* on the MathWorks website.
- **Model Advisor Check** — Provides links to the Model Advisor check that is used to verify compliance to the modeling guideline.

  When applicable, this column clarifies why a guideline does not have a corresponding check:

  - *No check* — Indicates that the guideline can be checked by using a Model Advisor check, however, the check does not currently exist.
  - *Not checkable* — Indicates that it is not possible to verify compliance to this guideline by using a Model Advisor check.

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
ar_0001: Usable characters for file names	ar_0001	“Check file names”

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
ar_0002: Usable characters for folder names	ar_0002	“Check folder names”
jc_0241: Length restriction for model file names	jc_0241	“Check length of model file name”
jc_0242: Length restriction for folder names	jc_0242	“Check length of folder name at every level of model path”
jc_0201: Usable characters for subsystem names	jc_0201	“Check Subsystem names”
jc_0231: Usable characters for block names	jc_0231	“Check character usage in block names”
jc_0211: Usable characters for Inport blocks and Outport block	jc_0211	“Check port block names”
jc_0243: Length restriction for subsystem names	jc_0243	“Check length of subsystem names”
jc_0247: Length restriction for block names	jc_0247	“Check length of block names”
jc_0244: Length restriction for Inport and Outport names	jc_0244	“Check length of Inport and Outport names”
jc_0222: Usable characters for signal and bus names	jc_0222	“Check usable characters for signal names and bus names”
jc_0232: Usable characters for parameter names	jc_0232	“Check usable characters for parameter names”
jc_0245: Length restriction for signal and bus names	jc_0245	“Check length of signal and bus names”
jc_0246: Length restriction for parameter name	jc_0246	“Check length of parameter names”
jc_0795: Usable characters for Stateflow data names	jc_0795	“Check usable characters for Stateflow data names”
jc_0796: Length restriction for Stateflow data names	jc_0796	“Check length of Stateflow data name”
jc_0791: Duplicate data name definitions	jc_0791	“Check duplication of Simulink data names”
jc_0792: Unused Data	jc_0792	“Check unused data in Simulink Model”
jc_0700: Unused data in Stateflow block	jc_0700	“Check for unused data in Stateflow Charts”
na_0019: Restricted variable names	No corresponding JMAAB guideline	“Check usage of restricted variable names”

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0011: Optimization parameters for Boolean data types	jc_0011	“Check Implement logic signals as Boolean data (vs. double)”
jc_0642: Integer rounding mode setting	jc_0642	“Check Signed Integer Division Rounding mode”
jc_0806: Detecting incorrect calculation results	jc_0806	“Check diagnostic settings for incorrect calculation results”
jc_0021: Model diagnostic settings	No corresponding JMAAB guideline	“Check model diagnostic parameters”
na_0004: Simulink model appearance settings	na_0004	“Check for Simulink diagrams using nonstandard display attributes”
db_0043: Model font and font size	db_0043	“Check Model font settings”
jm_0002: Block resizing	jm_0002	*No check*
db_0142: Position of block names	db_0142	“Check whether block names appear below blocks”
jc_0061: Display of block names	jc_0061	“Check the display attributes of block names”
db_0140: Display of block parameters	db_0140	“Check for nondefault block attributes”
jc_0603: Model description	jc_0603	“Check Model Description”
jc_0604: Using block shadow	jc_0604	“Check if blocks are shaded in the model”
db_0081: Unconnected signals and blocks	db_0081	“Check for unconnected ports and signal lines”
db_0032: Signal line connections	db_0032	“Check signal line connections”
db_0141: Signal flow in Simulink models	db_0141	“Check signal flow in model Check position of signal labels”
jc_0110: Direction of block	jc_0110	“Check block orientation”
jc_0171: Clarification of connections between structural subsystems	jc_0171	“Check connections between structural subsystems”
jc_0602: Consistency in model element names	jc_0602	“Check for consistency in model element names”
jc_0281: Trigger signal names	jc_0281	“Check trigger signal names”
db_0143: Usable block types in model hierarchy	db_0143	“Check for mixing basic blocks and subsystems”
db_0144: Use of subsystems	db_0144	*Not checkable*

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0653: Delay block layout in feedback loops	jc_0653	“Check for avoiding algebraic loops between subsystems”
hd_0001: Prohibited Simulink sinks	No corresponding JMAAB guideline	“Check for prohibited sink blocks”
na_0010: Usage of vector and bus signals	na_0010	“Check usage of vector and bus signals”
jc_0008: Definition of signal names	jc_0008	“Check signal line labels”
jc_0009: Signal name propagation	jc_0009	“Check for propagated signal labels”
db_0097: Position of labels for signals and buses	db_0097	“Check signal flow in model Check position of signal labels”
na_0008: Display of labels on signals	No corresponding JMAAB guideline	“Check signal line labels”
na_0009: Entry versus propagation of signal labels	No corresponding JMAAB guideline	“Check for propagated signal labels”
db_0110: Block parameters	db_0110	“Check usage of tunable parameters in blocks”
db_0112: Usage of index	db_0112	“Check Indexing Mode”
jc_0645: Parameter definition for calibration	jc_0645	“Check if tunable block parameters are defined as named constants”
jc_0641: Sample time setting	jc_0641	“Check for sample time setting”
jc_0643: Fixed-point setting	jc_0643	“Check usage of fixed-point data type with non-zero bias”
jc_0644: Type setting	jc_0644	“Check type setting by data objects”
db_0146: Block layout in conditional subsystems	db_0146	“Check position of conditional blocks and iterator blocks”
jc_0640: Initial value settings for Outport blocks in conditional subsystems	jc_0640	“Check undefined initial output for conditional subsystems”
jc_0659: Usage restrictions of signal lines input to Merge blocks	jc_0659	“Check usage of Merge block”
na_0003: Usage of If blocks	na_0003	“Check logical expressions in If blocks”
jc_0656: Usage of Conditional Control blocks	jc_0656	“Check default/else case in Switch Case blocks and If blocks”

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0657: Retention of output value based on conditional control flow blocks and Merge blocks	jc_0657	*Not checkable*
na_0002: Appropriate usage of basic logical and numerical operations	na_0002	“Check fundamental logical and numerical operations”
jc_0121: Usage of add and subtraction blocks	jc_0121	“Check usage of Sum blocks”
jc_0610: Operator order for multiplication and division block	jc_0610	“Check operator order of Product blocks”
jc_0611: Input sign for multiplication and division blocks	jc_0611	“Check signs of input signals in product blocks”
jc_0794: Division in Simulink	jc_0794	*No check*
jc_0805: Numerical operation block inputs	jc_0805	*No check*
jc_0622: Usage of Fcn blocks	jc_0622	“Check for parentheses in Fcn block expressions”
jc_0621: Usage of Logical Operator blocks	jc_0621	“Check icon shape of Logical Operator blocks”
jc_0131: Usage of Relational Operator blocks	jc_0131	“Check usage of Relational Operator blocks”
jc_0800: Comparing floating-point types in Simulink	jc_0800	“Comparing floating point types in Simulink”
jc_0626: Usage of Lookup Table blocks	jc_0626	“Check usage of Lookup Tables”
jc_0623: Usage of continuous-time Delay blocks and discrete-time Delay blocks	jc_0623	“Check usage of Memory and Unit Delay blocks”
jc_0624: Usage of Tapped Delay blocks/Delay blocks	jc_0624	“Check for cascaded Unit Delay blocks”
jc_0627: Usage of Discrete-Time Integrator blocks	jc_0627	“Check usage of Discrete-Time Integrator block”
jc_0628: Usage of Saturation blocks	jc_0628	“Check usage of the Saturation blocks”
jc_0651: Implementing a type conversion	jc_0651	“Check output data type of operation blocks”
db_0042: Usage of Inport and Outport blocks	db_0042	“Check position of Inport and Outport blocks”

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0081: Inport and Outport block icon display	jc_0081	“Check display for port blocks”
na_0011: Scope of Goto and From blocks	na_0011	“Check scope of From and Goto blocks”
jc_0161: Definition of Data Store Memory blocks	jc_0161	“Check for usage of Data Store Memory blocks”
jc_0141: Usage of the Switch blocks	jc_0141	“Check usage of Switch blocks”
jc_0650: Block input/output data type with switching function	jc_0650	“Check input and output datatype for Switch blocks”
jc_0630: Usage of Multiport Switch blocks	jc_0630	“Check settings for data ports in Multiport Switch blocks”
na_0020: Number of inputs to variant subsystems	na_0020	“Check for missing ports in Variant Subsystems”
na_0036: Default variant	na_0036	“Check use of default variants”
na_0037: Use of single variable for variant condition	na_0037	“Check use of single variable variant conditionals”
db_0122: Stateflow and Simulink interface signals and parameters	db_0122	“Check for Strong Data Typing with Simulink I/O”
db_0123: Stateflow port names	No corresponding JMAAB guideline	“Check for names of Stateflow ports and associated signals”
db_0125: Stateflow local data	db_0125	“Check definition of Stateflow data”
db_0126: Defining Stateflow events	db_0126	“Check definition of Stateflow events”
jc_0701: Usable number for first index	jc_0701	“Check usable number for first index”
jc_0712: Execution timing for default transition path	jc_0712	“Check execution timing for default transition path”
jc_0722: Local data definition in parallel states	jc_0722	“Check scope of data in parallel states”
jc_0797: Unconnected transitions / states / connective junctions	jc_0797	“Check for unconnected objects in Stateflow Charts”
db_0137: States in state machines	db_0137	“Check usage of exclusive and default states in state machines”
jc_0721: Usage of parallel states	jc_0721	“Check for parallel Stateflow state used for grouping”

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
db_0129: Stateflow transition appearance	db_0129	“Check for Stateflow transition appearance”
jc_0531: Default transition	jc_0531	“Check default transition placement in Stateflow charts”
jc_0723: Prohibited direct transition from external state to child state	jc_0723	“Check usage of transitions to external states”
jc_0751: Backtracking prevention in state transition	jc_0751	“Check for unexpected backtracking in state transitions”
jc_0760: Starting point of internal transition	jc_0760	“Check starting point of internal transition in Stateflow”
jc_0763: Usage of multiple internal transitions	jc_0763	“Check usage of internal transitions in Stateflow states”
jc_0762: Prohibition of state action and flow chart combination	jc_0762	“Check prohibited combination of state action and flow chart”
db_0132: Transitions in flow charts	db_0132	“Check transition orientations in flow charts”
jc_0773: Unconditional transition of a flow chart	jc_0773	“Check usage of unconditional transitions in flow charts”
jc_0775: Terminating junctions in flow charts	jc_0775	“Check terminal junctions in Stateflow”
jc_0738: Usage of Stateflow comments	jc_0738	“Check usage of Stateflow comments”
jc_0790: Action language of Chart block	jc_0790	“Check Stateflow chart action language”
jc_0702: Use of named Stateflow parameters and constants	jc_0702	“Check usage of numeric literals in Stateflow”
jm_0011: Pointers in Stateflow	jm_0011	“Check for pointers in Stateflow charts”
jc_0491: Reuse of Stateflow data	jc_0491	*Not checkable*
jm_0012: Usage restrictions of events and broadcasting events	jm_0012	“Check for usage of events and broadcasting events in Stateflow charts”
jc_0733: Order of state action types	jc_0733	“Check order of state action types”
jc_0734: Number of state action types	jc_0734	“Check repetition of Action types”

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0740: Limitation on use of exit state action	jc_0740	“Check if state action type 'exit' is used in the model”
jc_0741: Timing to update data used in state chart transition conditions	jc_0741	“Check updates to variables used in state transition conditions”
jc_0772: Execution order and transition conditions of transition lines	jc_0772	“Check usage of transition conditions in Stateflow transitions”
jc_0753: Condition actions and transition actions in Stateflow	jc_0753	“Check condition actions and transition actions in Stateflow”
jc_0711: Division in Stateflow	jc_0711	*Not checkable*
db_0127: Limitation on MATLAB commands in Stateflow blocks	db_0127	“Check for MATLAB expressions in Stateflow charts”
jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow	jc_0481	“Check usage of floating-point expressions in Stateflow charts”
na_0001: Standard usage of Stateflow operators	na_0001	“Check Stateflow operators”
jc_0655: Prohibition of logical value comparison in Stateflow	jc_0655	“Check prohibited comparison operation of logical type signals”
jc_0451: Use of unary minus on unsigned integers	jc_0451	“Check usage of unary minus operations in Stateflow charts”
jc_0802: Prohibited use of implicit type casting in Stateflow	jc_0802	“Check for implicit type casting in Stateflow”
jc_0803: Passing values to library functions	jc_0803	*Not checkable*
jc_0732: Distinction between state names, data names, and event names	jc_0732	“Check uniqueness of Stateflow State and Data names”
jc_0730: Unique state name in Stateflow blocks	jc_0730	“Check uniqueness of State names”
jc_0731: State name format	jc_0731	“Check usage of State names”
jc_0501: Format of entries in a State block	jc_0501	“Check entry formatting in State blocks in Stateflow charts”
jc_0736: Uniform indentations in Stateflow blocks	jc_0736	“Check indentation of code in Stateflow states”
jc_0739: Describing texts inside states	jc_0739	“Check for usage of text inside states”

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0770: Position of transition label	jc_0770	“Check placement of Label String in Transitions”
jc_0771: Comment position in transition labels	jc_0771	“Check position of comments in transition labels”
jc_0752: Condition action in transition label	jc_0752	“Check usage of parentheses in Stateflow transitions”
jc_0774: Comments for through transition	jc_0774	“Check for comments in unconditional transitions”
jc_0511: Return values from a graphical function	jc_0511	“Check return value assignments in Stateflow graphical functions”
jc_0804: Prohibited use of recursive calls with graphical functions	jc_0804	*No check*
na_0042: Usage of Simulink functions	na_0042	“Check usage of Simulink function in Stateflow”
na_0039: Limitation on Simulink functions in Chart blocks	na_0039	“Check use of Simulink in Stateflow charts”
na_0018: Number of nested if/ else and case statement	No corresponding JMAAB guideline	“Check MATLAB Function metrics”
na_0025: MATLAB Function header	No corresponding JMAAB guideline	*No check*
na_0024: Shared data in MATLAB functions	na_0024	“Check MATLAB code for global variables”
na_0031: Definition of default enumerated value	na_0031	“Check usage of enumerated values”
na_0034: MATLAB Function block input/output settings	na_0034	“Check input and output settings of MATLAB Functions”
na_0016: Source lines of MATLAB Functions	No corresponding JMAAB guideline	“Check MATLAB Function metrics”
na_0017: Number of called function levels	No corresponding JMAAB guideline	“Check the number of function calls in MATLAB Function blocks”
na_0021: Strings in MATLAB functions	na_0021	“Check usage of character vector inside MATLAB Function block”
na_0022: Recommended patterns for Switch/Case statements	No corresponding JMAAB guideline	“Check usage of recommended patterns for Switch/Case statements”
jc_0801: Prohibited use of the /* and */ comment symbols	jc_0801	“Check for use of C-style comment symbols”

See Also

- “Run Model Advisor Checks and Review Results” (Simulink)

Model Checks for High Integrity Systems Modeling

You can check that your model or subsystem complies with selected aspects of the High Integrity System Model safety standard by running the Model Advisor.

To check compliance with High Integrity System Model standards, run the high-integrity checks from these Model Advisor folders:

- **By Task > Modeling Standards for DO-178C/DO-331 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 61508 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 62304 > High-Integrity Systems**
- **By Task > Modeling Standards for EN 50128 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 26262 > High-Integrity Systems**

For information on the High Integrity System Model Software Considerations in Airborne Systems and Equipment Certification and related standards, see Radio Technical Commission for Aeronautics (RTCA).

The table lists the High Integrity System Model checks and their corresponding modeling guidelines. For more information about the High-Integrity Modeling Guidelines, see “High-Integrity System Modeling” (Simulink).

High Integrity Systems Modeling Checks

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage of lookup table blocks	“hisl_0033: Usage of Lookup Table blocks” (Simulink)
Check for inconsistent vector indexing methods	“hisl_0021: Consistent vector indexing method” (Simulink)
Check for variant blocks with 'Generate preprocessor conditionals' active	“hisl_0023: Verification of model and subsystem variants” (Simulink)
Check for root Inports with missing properties	“hisl_0024: Inport interface definition” (Simulink)
Check for Relational Operator blocks that equate floating-point types	“hisl_0017: Usage of blocks that compute relational operators (2)” (Simulink)
Check usage of Relational Operator blocks	“hisl_0016: Usage of blocks that compute relational operators” (Simulink)
Check usage of Logical Operator blocks	“hisl_0018: Usage of Logical Operator block” (Simulink)
Check usage of While Iterator blocks	“hisl_0006: Usage of While Iterator blocks” (Simulink)
Check sample time-dependent blocks	“hisl_0007: Usage of For Iterator or While Iterator subsystems” (Simulink)
Check usage of For Iterator blocks	“hisl_0008: Usage of For Iterator Blocks” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage of If blocks and If Action Subsystem blocks	“hisl_0010: Usage of If blocks and If Action Subsystem blocks” (Simulink)
Check usage Switch Case blocks and Switch Case Action Subsystem blocks	“hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks” (Simulink)
Check safety-related optimization settings for logic signals	“hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)” (Simulink)
Check safety-related block reduction optimization settings	“hisl_0046: Configuration Parameters > Simulation Target > Block reduction” (Simulink)
Check safety-related optimization settings for application lifespan	“hisl_0048: Configuration Parameters > Math and Data Types > Application lifespan (days)” (Simulink)
Check safety-related optimization settings for data initialization	“hisl_0052: Configuration Parameters > Code Generation > Optimization > Data initialization” (Simulink)
Check safety-related optimization settings for data type conversions	“hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values” (Simulink)
Check safety-related optimization settings for division arithmetic exceptions	“hisl_0054: Configuration Parameters > Code Generation > Optimization > Remove code that protects against division arithmetic exceptions” (Simulink)
Check safety-related code generation settings for comments	“hisl_0038: Configuration Parameters > Code Generation > Comments” (Simulink)
Check safety-related code generation interface settings	“hisl_0039: Configuration Parameters > Code Generation > Interface” (Simulink)
Check safety-related code generation settings for code style	“hisl_0047: Configuration Parameters > Code Generation > Code Style” (Simulink)
Check safety-related code generation identifier settings	“hisl_0049: Configuration Parameters > Code Generation > Identifiers” (Simulink)
Check usage of Abs blocks	“hisl_0001: Usage of Abs block” (Simulink)
Check usage of Math Function blocks (rem and reciprocal functions)	“hisl_0002: Usage of Math Function blocks (rem and reciprocal)” (Simulink)
Check usage of Math Function blocks (log and log10 functions)	“hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)” (Simulink)
Check usage of Assignment blocks	“hisl_0029: Usage of Assignment blocks” (Simulink)
Check usage of Signal Routing blocks	“hisl_0034: Usage of Signal Routing blocks” (Simulink)
Check for root Inports with missing range definitions	“hisl_0025: Design min/max specification of input interfaces” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check for root Outports with missing range definitions	“hisl_0026: Design min/max specification of output interfaces” (Simulink)
Check state machine type of Stateflow charts	“hisf_0001: State Machine Type” (Simulink)
Check Stateflow charts for transition paths that cross parallel state boundaries	“hisf_0013: Usage of transition paths (crossing parallel state boundaries)” (Simulink)
Check Stateflow charts for ordering of states and transitions	“hisf_0002: User-specified state/transition execution order” (Simulink)
Check Stateflow debugging options	“hisf_0011: Stateflow debugging settings” (Simulink)
Check Stateflow charts for uniquely defined data objects	“hisl_0061: Unique identifiers for clarity” (Simulink)
Check Stateflow charts for strong data typing	“hisf_0015: Strong data typing (casting variables and parameters in expressions)” (Simulink)
Check usage of shift operations for Stateflow data	“hisf_0064: Shift operations for Stateflow data to improve code compliance” (Simulink)
Check assignment operations in Stateflow charts	“hisf_0065: Type cast operations in Stateflow to improve code compliance” (Simulink)
Check Stateflow charts for unary operators	“hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance” (Simulink)
Check for Strong Data Typing with Simulink I/O	“hisf_0009: Strong data typing (Simulink and Stateflow boundary)” (Simulink)
Check for MATLAB Function interfaces with inherited properties	“himl_0002: Strong data typing at MATLAB function boundaries” (Simulink)
Check MATLAB Function metrics	“himl_0003: Limitation of MATLAB function complexity” (Simulink)
Check MATLAB Code Analyzer messages	“himl_0004: MATLAB Code Analyzer recommendations for code generation” (Simulink)
Check safety-related model referencing settings	“hisl_0037: Configuration Parameters > Model Referencing” (Simulink)
Check safety-related diagnostic settings for solvers	“hisl_0043: Configuration Parameters > Diagnostics > Solver” (Simulink)
Check safety-related solver settings for simulation time	“hisl_0040: Configuration Parameters > Solver > Simulation time” (Simulink)
Check safety-related solver settings for solver options	“hisl_0041: Configuration Parameters > Solver > Solver options” (Simulink)
Check safety-related solver settings for tasking and sample-time	“hisl_0042: Configuration Parameters > Solver > Tasking and sample time options” (Simulink)
Check safety-related diagnostic settings for sample time	“hisl_0044: Configuration Parameters > Diagnostics > Sample Time” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for parameters	“hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters” (Simulink)
Check safety-related diagnostic settings for data used for debugging	“hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging” (Simulink)
Check safety-related diagnostic settings for data store memory	“hisl_0013: Usage of data store blocks” (Simulink)
Check safety-related diagnostic settings for type conversions	“hisl_0309: Configuration Parameters > Diagnostics > Type Conversion” (Simulink)
Check safety-related diagnostic settings for signal connectivity	“hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals” (Simulink)
Check safety-related diagnostic settings for bus connectivity	“hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses” (Simulink)
Check safety-related diagnostic settings that apply to function-call connectivity	“hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls” (Simulink)
Check safety-related diagnostic settings for compatibility	“hisl_0301: Configuration Parameters > Diagnostics > Compatibility” (Simulink)
Check safety-related diagnostic settings for model initialization	“hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization” (Simulink)
Check safety-related diagnostic settings for model referencing	“hisl_0310: Configuration Parameters > Diagnostics > Model Referencing” (Simulink)
Check safety-related diagnostic settings for saving	“hisl_0036: Configuration Parameters > Diagnostics > Saving” (Simulink)
Check safety-related diagnostic settings for Merge blocks	“hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks” (Simulink)
Check safety-related diagnostic settings for Stateflow	“hisl_0311: Configuration Parameters > Diagnostics > Stateflow” (Simulink)
Check safety-related optimization settings for Loop unrolling threshold	“hisl_0051: Configuration Parameters > Code Generation > Optimization > Loop unrolling threshold” (Simulink)
Check model object names	“hisl_0032: Model object names” (Simulink)
Check for model elements that do not link to requirements	“hisl_0070: Placement of requirement links in a model” (Simulink)
Check for inappropriate use of transition paths	“hisf_0014: Usage of transition paths (passing through states)” (Simulink)
Check usage of Bitwise Operator block	“hisl_0019: Usage of bitwise operations” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check data types for blocks with index signals	“hisl_0022: Data type selection for index signals” (Simulink)
Check model file name	“hisl_0031: Model file names” (Simulink)
Check if/elseif/else patterns in MATLAB Function blocks	“himl_0006: MATLAB code if / elseif / else patterns” (Simulink)
Check switch statements in MATLAB Function blocks	“himl_0007: MATLAB code switch / case / otherwise patterns” (Simulink)
Check global variables in graphical functions	“hisl_0062: Global variables in graphical functions” (Simulink)
Check for length of user-defined object names	“hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance” (Simulink)
Check usage of Merge blocks	“hisl_0015: Usage of Merge blocks” (Simulink)
Check usage of conditionally executed subsystems	“hisl_0012: Usage of conditionally executed subsystems” (Simulink)
Check usage of standardized MATLAB function headers	“himl_0001: Usage of standardized MATLAB function headers” (Simulink)
Check usage of relational operators in MATLAB Function blocks	“himl_0008: MATLAB code relational operator data types” (Simulink)
Check usage of equality operators in MATLAB Function blocks	“himl_0009: MATLAB code with equal / not equal relational operators” (Simulink)
Check usage of logical operators and functions in MATLAB Function blocks	“himl_0010: MATLAB code with logical operators and functions” (Simulink)
Check type and size of conditional expressions	“himl_0011: Data type and size of condition expressions” (Simulink)
Check naming of ports in Stateflow charts	“hisf_0016: Stateflow port names” (Simulink)
Check scoping of Stateflow data objects	“hisf_0017: Stateflow data object scoping” (Simulink)
Check usage of Gain blocks	“hisl_0066: Usage of Gain blocks” (Simulink)
Check usage of bitwise operations in Stateflow charts	“hisf_0003: Usage of bitwise operations” (Simulink)
Check data type of loop control variables	“hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance” (Simulink)
Check configuration parameters for MISRA C:2012	“hisl_0060: Configuration parameters that improve MISRA C:2012 compliance” (Simulink)
Check for blocks not recommended for C/C++ production code deployment Check for blocks not recommended for MISRA C:2012	“hisl_0020: Blocks not recommended for MISRA C:2012 compliance” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related optimization settings for specified minimum and maximum values	“hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values” (Simulink)

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 with these standards, open the Model Advisor on page 3-4 and run the checks in these 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.

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks
Display configuration management data
Display model metrics and complexity report
Check for unconnected objects
Display bug reports for Embedded Coder
Display bug reports for IEC Certification Kit
Display bug reports for Polyspace Code Prover
Display bug reports for Polyspace Code Prover Server
Display bug reports for Polyspace Bug Finder
Display bug reports for Polyspace Bug Finder Server
Display bug reports for Simulink Design Verifier
Display bug reports for Simulink PLC Coder
Display bug reports for Simulink Check
Display bug reports for Simulink Coverage
Display bug reports for Simulink Test
Display bug reports for Simulink Requirements
Display bug reports for AUTOSAR Blockset

Following are the High-Integrity System Modeling checks that are applicable for the IEC 61508, IEC 62304, ISO 26262, and EN 50128 standards.

Model Checks for High Integrity Systems Modeling

You can check that your model or subsystem complies with selected aspects of the High Integrity System Model safety standard by running the Model Advisor.

To check compliance with High Integrity System Model standards, run the high-integrity checks from these Model Advisor folders:

- **By Task > Modeling Standards for DO-178C/DO-331 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 61508 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 62304 > High-Integrity Systems**
- **By Task > Modeling Standards for EN 50128 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 26262 > High-Integrity Systems**

For information on the High Integrity System Model Software Considerations in Airborne Systems and Equipment Certification and related standards, see Radio Technical Commission for Aeronautics (RTCA).

The table lists the High Integrity System Model checks and their corresponding modeling guidelines. For more information about the High-Integrity Modeling Guidelines, see “High-Integrity System Modeling” (Simulink).

High Integrity Systems Modeling Checks

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage of lookup table blocks	“hisl_0033: Usage of Lookup Table blocks” (Simulink)
Check for inconsistent vector indexing methods	“hisl_0021: Consistent vector indexing method” (Simulink)
Check for variant blocks with 'Generate preprocessor conditionals' active	“hisl_0023: Verification of model and subsystem variants” (Simulink)
Check for root Inports with missing properties	“hisl_0024: Inport interface definition” (Simulink)
Check for Relational Operator blocks that equate floating-point types	“hisl_0017: Usage of blocks that compute relational operators (2)” (Simulink)
Check usage of Relational Operator blocks	“hisl_0016: Usage of blocks that compute relational operators” (Simulink)
Check usage of Logical Operator blocks	“hisl_0018: Usage of Logical Operator block” (Simulink)
Check usage of While Iterator blocks	“hisl_0006: Usage of While Iterator blocks” (Simulink)
Check sample time-dependent blocks	“hisl_0007: Usage of For Iterator or While Iterator subsystems” (Simulink)
Check usage of For Iterator blocks	“hisl_0008: Usage of For Iterator Blocks” (Simulink)
Check usage of If blocks and If Action Subsystem blocks	“hisl_0010: Usage of If blocks and If Action Subsystem blocks” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage Switch Case blocks and Switch Case Action Subsystem blocks	“hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks” (Simulink)
Check safety-related optimization settings for logic signals	“hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)” (Simulink)
Check safety-related block reduction optimization settings	“hisl_0046: Configuration Parameters > Simulation Target > Block reduction” (Simulink)
Check safety-related optimization settings for application lifespan	“hisl_0048: Configuration Parameters > Math and Data Types > Application lifespan (days)” (Simulink)
Check safety-related optimization settings for data initialization	“hisl_0052: Configuration Parameters > Code Generation > Optimization > Data initialization” (Simulink)
Check safety-related optimization settings for data type conversions	“hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values” (Simulink)
Check safety-related optimization settings for division arithmetic exceptions	“hisl_0054: Configuration Parameters > Code Generation > Optimization > Remove code that protects against division arithmetic exceptions” (Simulink)
Check safety-related code generation settings for comments	“hisl_0038: Configuration Parameters > Code Generation > Comments” (Simulink)
Check safety-related code generation interface settings	“hisl_0039: Configuration Parameters > Code Generation > Interface” (Simulink)
Check safety-related code generation settings for code style	“hisl_0047: Configuration Parameters > Code Generation > Code Style” (Simulink)
Check safety-related code generation identifier settings	“hisl_0049: Configuration Parameters > Code Generation > Identifiers” (Simulink)
Check usage of Abs blocks	“hisl_0001: Usage of Abs block” (Simulink)
Check usage of Math Function blocks (rem and reciprocal functions)	“hisl_0002: Usage of Math Function blocks (rem and reciprocal)” (Simulink)
Check usage of Math Function blocks (log and log10 functions)	“hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)” (Simulink)
Check usage of Assignment blocks	“hisl_0029: Usage of Assignment blocks” (Simulink)
Check usage of Signal Routing blocks	“hisl_0034: Usage of Signal Routing blocks” (Simulink)
Check for root Inports with missing range definitions	“hisl_0025: Design min/max specification of input interfaces” (Simulink)
Check for root Outports with missing range definitions	“hisl_0026: Design min/max specification of output interfaces” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check state machine type of Stateflow charts	“hisf_0001: State Machine Type” (Simulink)
Check Stateflow charts for transition paths that cross parallel state boundaries	“hisf_0013: Usage of transition paths (crossing parallel state boundaries)” (Simulink)
Check Stateflow charts for ordering of states and transitions	“hisf_0002: User-specified state/transition execution order” (Simulink)
Check Stateflow debugging options	“hisf_0011: Stateflow debugging settings” (Simulink)
Check Stateflow charts for uniquely defined data objects	“hisl_0061: Unique identifiers for clarity” (Simulink)
Check Stateflow charts for strong data typing	“hisf_0015: Strong data typing (casting variables and parameters in expressions)” (Simulink)
Check usage of shift operations for Stateflow data	“hisf_0064: Shift operations for Stateflow data to improve code compliance” (Simulink)
Check assignment operations in Stateflow charts	“hisf_0065: Type cast operations in Stateflow to improve code compliance” (Simulink)
Check Stateflow charts for unary operators	“hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance” (Simulink)
Check for Strong Data Typing with Simulink I/O	“hisf_0009: Strong data typing (Simulink and Stateflow boundary)” (Simulink)
Check for MATLAB Function interfaces with inherited properties	“himl_0002: Strong data typing at MATLAB function boundaries” (Simulink)
Check MATLAB Function metrics	“himl_0003: Limitation of MATLAB function complexity” (Simulink)
Check MATLAB Code Analyzer messages	“himl_0004: MATLAB Code Analyzer recommendations for code generation” (Simulink)
Check safety-related model referencing settings	“hisl_0037: Configuration Parameters > Model Referencing” (Simulink)
Check safety-related diagnostic settings for solvers	“hisl_0043: Configuration Parameters > Diagnostics > Solver” (Simulink)
Check safety-related solver settings for simulation time	“hisl_0040: Configuration Parameters > Solver > Simulation time” (Simulink)
Check safety-related solver settings for solver options	“hisl_0041: Configuration Parameters > Solver > Solver options” (Simulink)
Check safety-related solver settings for tasking and sample-time	“hisl_0042: Configuration Parameters > Solver > Tasking and sample time options” (Simulink)
Check safety-related diagnostic settings for sample time	“hisl_0044: Configuration Parameters > Diagnostics > Sample Time” (Simulink)
Check safety-related diagnostic settings for parameters	“hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for data used for debugging	“hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging” (Simulink)
Check safety-related diagnostic settings for data store memory	“hisl_0013: Usage of data store blocks” (Simulink)
Check safety-related diagnostic settings for type conversions	“hisl_0309: Configuration Parameters > Diagnostics > Type Conversion” (Simulink)
Check safety-related diagnostic settings for signal connectivity	“hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals” (Simulink)
Check safety-related diagnostic settings for bus connectivity	“hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses” (Simulink)
Check safety-related diagnostic settings that apply to function-call connectivity	“hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls” (Simulink)
Check safety-related diagnostic settings for compatibility	“hisl_0301: Configuration Parameters > Diagnostics > Compatibility” (Simulink)
Check safety-related diagnostic settings for model initialization	“hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization” (Simulink)
Check safety-related diagnostic settings for model referencing	“hisl_0310: Configuration Parameters > Diagnostics > Model Referencing” (Simulink)
Check safety-related diagnostic settings for saving	“hisl_0036: Configuration Parameters > Diagnostics > Saving” (Simulink)
Check safety-related diagnostic settings for Merge blocks	“hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks” (Simulink)
Check safety-related diagnostic settings for Stateflow	“hisl_0311: Configuration Parameters > Diagnostics > Stateflow” (Simulink)
Check safety-related optimization settings for Loop unrolling threshold	“hisl_0051: Configuration Parameters > Code Generation > Optimization > Loop unrolling threshold” (Simulink)
Check model object names	“hisl_0032: Model object names” (Simulink)
Check for model elements that do not link to requirements	“hisl_0070: Placement of requirement links in a model” (Simulink)
Check for inappropriate use of transition paths	“hisf_0014: Usage of transition paths (passing through states)” (Simulink)
Check usage of Bitwise Operator block	“hisl_0019: Usage of bitwise operations” (Simulink)
Check data types for blocks with index signals	“hisl_0022: Data type selection for index signals” (Simulink)
Check model file name	“hisl_0031: Model file names” (Simulink)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check if/elseif/else patterns in MATLAB Function blocks	“himl_0006: MATLAB code if / elseif / else patterns” (Simulink)
Check switch statements in MATLAB Function blocks	“himl_0007: MATLAB code switch / case / otherwise patterns” (Simulink)
Check global variables in graphical functions	“hisl_0062: Global variables in graphical functions” (Simulink)
Check for length of user-defined object names	“hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance” (Simulink)
Check usage of Merge blocks	“hisl_0015: Usage of Merge blocks” (Simulink)
Check usage of conditionally executed subsystems	“hisl_0012: Usage of conditionally executed subsystems” (Simulink)
Check usage of standardized MATLAB function headers	“himl_0001: Usage of standardized MATLAB function headers” (Simulink)
Check usage of relational operators in MATLAB Function blocks	“himl_0008: MATLAB code relational operator data types” (Simulink)
Check usage of equality operators in MATLAB Function blocks	“himl_0009: MATLAB code with equal / not equal relational operators” (Simulink)
Check usage of logical operators and functions in MATLAB Function blocks	“himl_0010: MATLAB code with logical operators and functions” (Simulink)
Check type and size of conditional expressions	“himl_0011: Data type and size of condition expressions” (Simulink)
Check naming of ports in Stateflow charts	“hisf_0016: Stateflow port names” (Simulink)
Check scoping of Stateflow data objects	“hisf_0017: Stateflow data object scoping” (Simulink)
Check usage of Gain blocks	“hisl_0066: Usage of Gain blocks” (Simulink)
Check usage of bitwise operations in Stateflow charts	“hisf_0003: Usage of bitwise operations” (Simulink)
Check data type of loop control variables	“hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance” (Simulink)
Check configuration parameters for MISRA C:2012	“hisl_0060: Configuration parameters that improve MISRA C:2012 compliance” (Simulink)
Check for blocks not recommended for C/C++ production code deployment Check for blocks not recommended for MISRA C:2012	“hisl_0020: Blocks not recommended for MISRA C:2012 compliance” (Simulink)
Check safety-related optimization settings for specified minimum and maximum values	“hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values” (Simulink)

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” on page 3-4

Model Checks for MISRA C:2012 Compliance

To check that your model or subsystem has a likelihood of generating MISRA C:2012 compliant code, open the Model Advisor on page 3-4 and run the checks in **By Task > Modeling Guidelines for MISRA C:2012**:

- “Check usage of Assignment blocks”
- “Check for blocks not recommended for MISRA C:2012”
- “Check for blocks not recommended for C/C++ production code deployment”
- “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”
- “Check for missing error ports for AUTOSAR receiver interfaces”
- “Check for missing const qualifiers in model functions”
- “Check integer word length”
- “Check bus object names that are used as bus element names”

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” on page 3-4

Model Checks for Secure Coding (CERT C, CWE, and ISO/IEC TS 17961 Standards)

To check that your code complies with the CERT C, CWE, and ISO/IEC TS 17961 (Embedded Coder) secure coding standards, open the Model Advisor on page 3-4 and run the checks in **By Task > Modeling Guidelines for Secure Coding (CERT C, CWE, ISO/IEC TS 17961)**:

- “Check configuration parameters for secure coding standards”
- “Check for blocks not recommended for C/C++ production code deployment”
- “Check for blocks not recommended for secure coding standards”
- “Check usage of Assignment blocks”
- “Check for switch case expressions without a default case”
- “Check for bitwise operations on signed integers”
- “Check for equality and inequality operations on floating-point values”
- “Check integer word length”
- “Detect Dead Logic”
- “Detect Integer Overflow”
- “Detect Division by Zero”
- “Detect Out Of Bound Array Access”
- “Detect Specified Minimum and Maximum Value Violations”

See Also

Related Examples

- “Run Model Advisor Checks and Review Results” on page 3-4

Model Checks for Requirements Links

To check that every requirements link in your model has a valid target in a requirements document, from the Simulink Toolstrip, open the **Requirements** app. Click **Check Consistency** to run the Requirements Consistency Checking checks in the Model Advisor.

In the Model Advisor, the requirements consistency checks are available in:

- **By Product > Simulink Requirements > Requirements Consistency**
- **By Task > Requirements Consistency Checking**

For more information about these Model Advisor checks, see “Requirements Consistency Checks” (Simulink Requirements)

When modeling for high-integrity systems, to check that model elements link to requirement documents, run Check for model elements that do not link to requirements.

See Also

Related Examples

- “Validate Requirements Links in a Model” (Simulink Requirements)
- “Run Model Advisor Checks and Review Results” on page 3-4
- “High-Integrity System Modeling” (Simulink)

Check Systems Programmatically

Checking Systems Programmatically

The Simulink Check 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 4-7

More About

- “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3

Create a Function for Checking Multiple Systems

You can create a function to programmatically run multiple checks on a model. The function returns the number of failures and warnings.

1. In the MATLAB window, select **New** > **Function**.
2. Save the file as `run_configuration.m`.
3. In the function, right-click on `untitled` and select **Replace function name by file name**. The function name is updated to `run_configuration`.

   ```
   function [outputArg1, outputArg2] = run_configuration(inputArg1,inputArg2)
   ```
4. Define the output and input arguments. For the output arguments, press **Shift-Enter** after entering each value to automatically update inlining instances in the function.

   - `output_Arg1` as `fail`
   - `output_Arg2` as `warn`
   - `inputArg1, inputArg2` to `SysList`

   ```
   function [fail, warn] = run_configuration(SysList)
   fail = inputArg1;
   warn = inputArg2;
   ```
5. Inside the function, specify the list of checks to run using the example Model Advisor configuration file:

   ```
   fileName = 'slvnvdemo_mdladv_config.mat';
   ```
6. Call the `ModelAdvisor.run` function:

   ```
   SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName);
   ```
7. 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;
   ```

   The function should now look like this:

   ```
   end

   function [fail, warn] = run_configuration(SysList)

   %RUN_CONFIGURATION Check systems with Model Advisor
   %   Check systems given as input and return number of warnings and
   %   failures.

   fail = 0;
   warn = 0;

   ModelAdvisor.setDefaultConfiguration('C:\temp\ma_configuration_standards.json');

   SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName);

   for i=1:length(SysResultObjArray)
       fail = fail + SysResultObjArray{i}.numFail;
       warn = warn + SysResultObjArray{i}.numWarn;
   end

   end
   function
   ```

8. Save the function.
9. 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');
```

10. Review the results. Click the Summary Report link to open the Model Advisor Command-Line Summary report.

See Also

`ModelAdvisor.run`

Related Examples

- “Create a Function for Checking Multiple Systems in Parallel” on page 4-5

Create a Function for Checking Multiple Systems in Parallel

Checking multiple systems in parallel reduces the processing time required by the Model Advisor to check multiple systems. If you have the 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. In parallel mode, load the model data from the model workspace or data dictionary. The Model Advisor in parallel mode does not support model data in the base workspace.

1. In the MATLAB window, select **New > Function**.
2. Save the file as `run_fast_configuration.m`.
3. In the function, right-click on `untitled` and select **Replace function name by file name**. The function name is updated to `run_fast_configuration`

   ```
   function [outputArg1, outputArg2] = run_fast_configuration(inputArg1,inputArg2)
   ```
4. Define the output and input arguments. For the output arguments, press **Shift-Enter** after entering each value to automatically update all instances in the function.

   - `output_Arg1` as `fail`
   - `output_Arg2` as `warn`
   - `inputArg1, inputArg2` to `SysList`

   ```
   function [fail, warn] = run_fast_configuration(SysList)
   fail = inputArg1;
   warn = inputArg2;
   ```
5. Inside the function, specify the list of checks to run using the example Model Advisor configuration file:

   ```
   fileName = 'slvnvdemo_mdladv_config.mat';
   ```
6. Call the `ModelAdvisor.run` function and set `'ParallelMode'` to `'On'`.

   ```
   SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName,...
       'ParallelMode','On');
   ```
7. 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;
   ```

   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 = 'slvnvdemo_mdladv_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
```

8. Save the function.
9. 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'};
```

10. Run `run_fast_configuration` on the list of systems:

```
[failures, warnings] = run_fast_configuration(SysList);
```

11. Review the results. Click the Summary Report link to open the Model Advisor Command-Line Summary report.

See Also

`ModelAdvisor.run`

Related Examples

- “Create a Function for Checking Multiple Systems” on page 4-3

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','slvnvdemo_mdladv_config.mat','DisplayResults','Details');
```

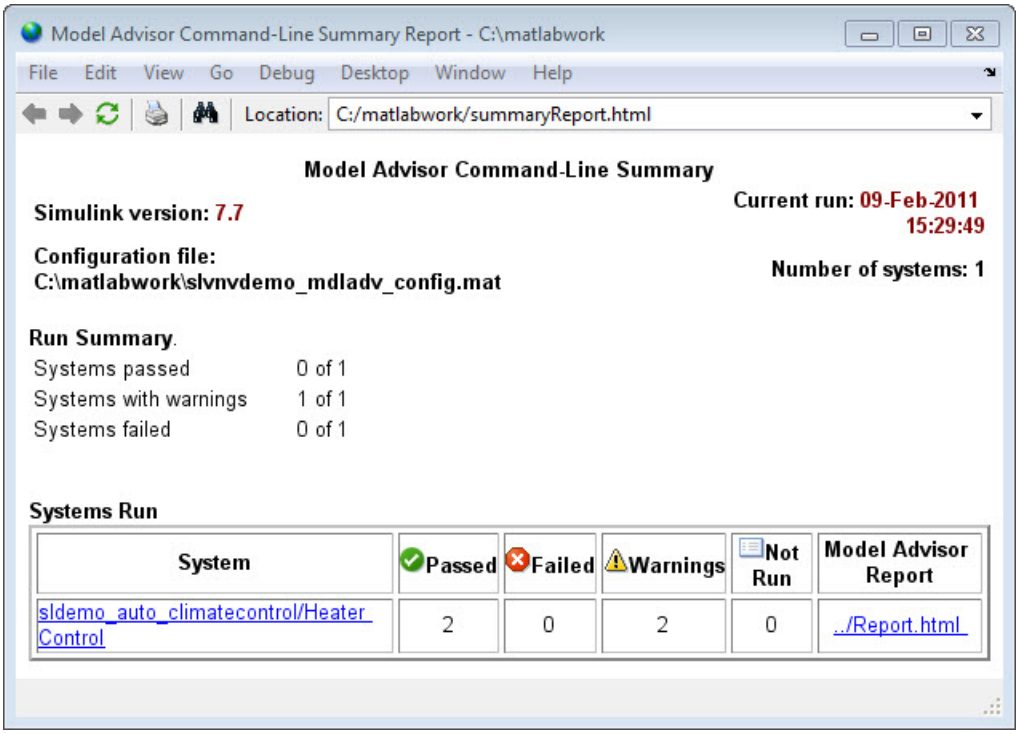
The results displayed in the Command Window are:

```
 Running Model Advisor
         Running Model Advisor on sldemo_auto_climatecontrol/Heater Control
         ============================================================
         Model Advisor run: 26-Jun-2019 15:01:13
         Configuration: slvnvdemo_mdladv_config.mat
         System: sldemo_auto_climatecontrol/Heater Control
         System version: 10
         Created by: The MathWorks, Inc.
         ============================================================
         (1) Warning: Check model diagnostic parameters [check ID: mathworks.maab.jc_0021]
         ------------------------------------------------------------
         (2) Pass: Check for root Inports with missing properties [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
                      3        1          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`.



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 4-10
- “Create a Function for Checking Multiple Systems” on page 4-3
- “Create a Function for Checking Multiple Systems in Parallel” on page 4-5

More About

- “Run Model Advisor Checks and Review Results” on page 3-4
- “Address Model Check Results” (Simulink)
- “Generate Model Advisor Reports” on page 3-19
- “Save and View Model Advisor Check Reports” (Simulink)
- “Find Model Advisor Check IDs” (Simulink)
- “Run Model Advisor Checks in Background” (Simulink)
- “Save and Load Process for Objects” (MATLAB)

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','slvnvdemo_mdladv_config.mat');
```

2. Save the `SystResulObj` for use at a later time:

```
save my_model_advisor_run SysResultObjArray
```

3. Clear the workspace to simulate viewing the results at a different time:

```
clear
```

4. Load the results of the Model Advisor run:

```
load my_model_advisor_run SysResultObjArray
```

5. View the results in the Model Advisor:

```
viewReport(SysResultObjArray{1},'MA')
```

See Also

`ModelAdvisor.run`

Related Examples

- “Archive and View Results” on page 4-7

Model Metrics

Collect and Explore Metric Data by Using the Metrics Dashboard

The Metrics Dashboard collects and integrates quality metric data from multiple Model-Based Design tools to provide you with an assessment of your project quality status. To open the dashboard:

- In the Apps gallery, click **Metrics Dashboard**.
- At the command line, enter `metricsdashboard(`*`system`*`)`. The *`system`* can be either a model name or a block path to a subsystem. The system cannot be a Configurable Subsystem block.

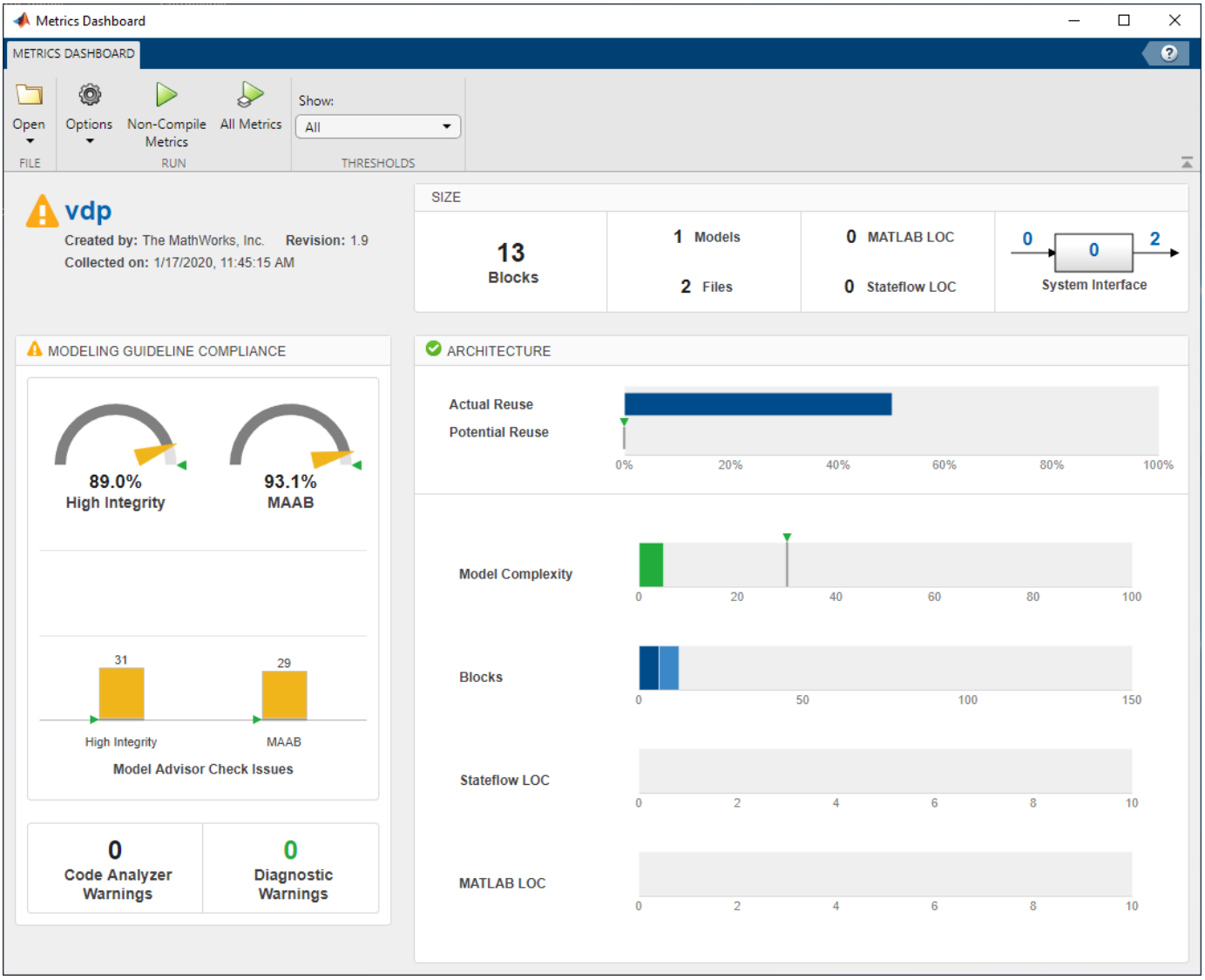
You can collect metric data by using the dashboard or programmatically by using the `slmetric.Engine` API. When you open the dashboard, if you have previously collected metric data for a particular model, the dashboard populates from existing data in the database.

If you want to use the dashboard to collect (or recollect) metric data, in the toolbar:

- Use the **Options** menu to specify whether to include model references and libraries in the data collection.
- Click **All Metrics**. If you do not want to collect metrics that require compiling the model, click **Non-Compile Metrics**.

The Metrics Dashboard provides the system name and a data collection timestamp. If there were issues during data collection, click the alert icon to see warnings.

You can only have one dashboard open per model or subsystem at once. Also, if a dashboard is open for a model or subsystem, and you programmatically collect metric data for that model or subsystem, the dashboard automatically closes.



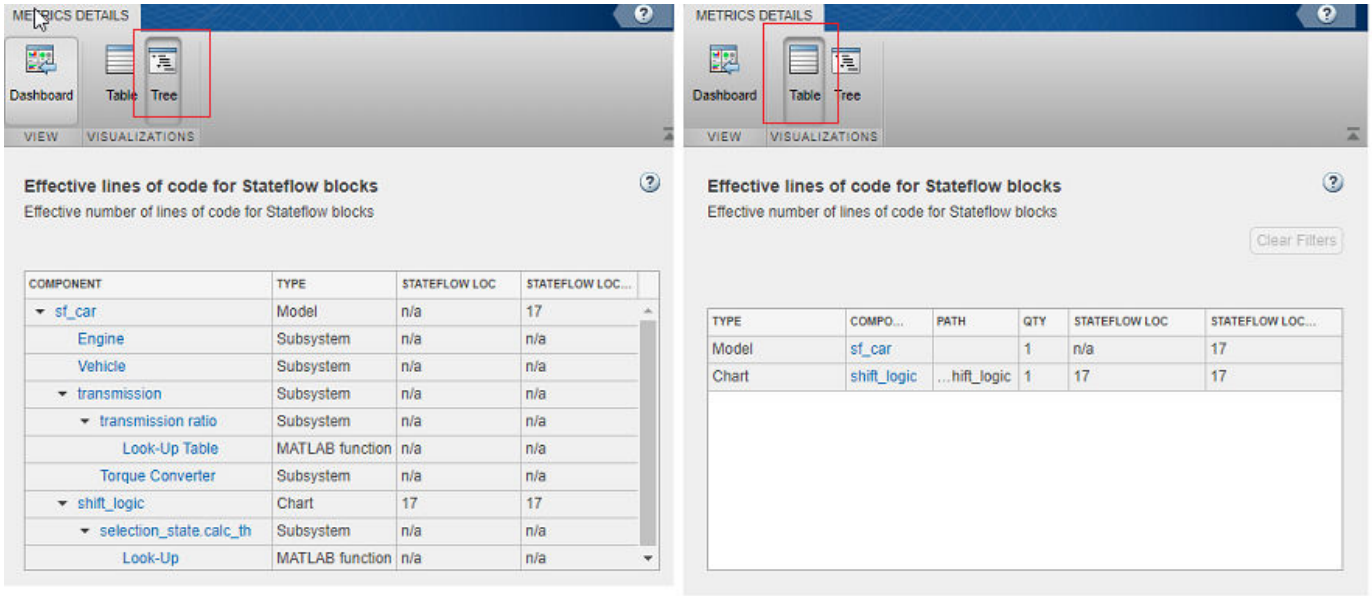
You can only have one dashboard open per model or subsystem at once. Also, if a dashboard is open for a model or subsystem, and you programmatically collect metric data for that model or subsystem, the dashboard automatically closes.

Metrics Dashboard Widgets

The Metrics Dashboard contains widgets that provide visualization of metric data in these categories: size, modeling guideline compliance, and architecture. To explore the data in more detail, click an individual metric widget. For your selected metric, a table displays the value, aggregated value, and measures (if applicable) at the model component level. From the table, the dashboard provides traceability and hyperlinks to the data source so that you can get detailed results and recommended actions for troubleshooting issues. When exploring drill-in data, note that:

- The Metrics Dashboard calculates metric data per component. A component can be a model, subsystem, chart, or MATLAB Function block.

- You can view results in either a **Tree** or **Table** view. For the **High Integrity** and **MAB** compliance widgets, you can also choose a **Grid** view. To view highlighted results, in the grid view, click a cell.
- To sort the results by value or aggregated value, click the corresponding value column header.
- For metrics other than the **High Integrity** and **MAB** compliance widgets, you can filter results. To filter results, in the **Table** view, select the context menu on the right side of the **TYPE**, **COMPONENT**, and **PATH** column headers. From the **TYPE** menu, select applicable components. From the **COMPONENT** and **PATH** menus, type a component name or path in the search bar. The Metrics Dashboard saves the filters for a widget, so you can view metric details for other widgets and return to the filtered results.
- In the **Table** and **Tree** view, a value or aggregated value of n/a indicates that results are not available for that component. If the value and aggregated value are n/a, the **Table** view does not list the component. The **Tree** view does list such a component. For the **Stateflow LOC** widget, the image shows the comparison.



- The metric data that is collected quantifies the overall system, including instances of the same model. For aggregated values, the metric engine aggregates data from each instance of a model in the referencing hierarchy. For example, if the same model is referenced twice in the system hierarchy, its block count contributes twice to the overall system block count.
- If a subsystem, chart, or MATLAB Function block uses a parameter or is flagged for an issue, then the parameter count or issue count is increased for the parent component.
- The Metrics Dashboard analyzes variants.

For custom metrics, you can specify widgets to add to the dashboard. You can also remove widgets. To learn more about customizing the Metrics Dashboard, see “Customize Metrics Dashboard Layout and Functionality” on page 5-39.

Size

This table lists the Metrics Dashboard widgets that provide an overall picture of the size of your system. When you drill into a widget, this table also lists the detailed information available.

Widget	Metric	Drill-In Data
Blocks	**Simulink block count** (`mathworks.metrics.SimulinkBlockCount`)	Number of blocks by component
Models	**Model file count** (`mathworks.metrics.ModelFileCount`)	Number of model files by component
Files	**File count** (`mathworks.metrics.FileCount`)	Number of model and library files by component
MATLAB LOC	**Effective lines of MATLAB code** (`mathworks.metrics.MatlabLOCCount`)	Effective lines of code, in MATLAB Function block and MATLAB functions in Stateflow, by component
Stateflow LOC	**Effective lines of code for Stateflow blocks** (`mathworks.metrics.StateflowLOCCount`)	Effective lines of code for Stateflow blocks by component
System Interface	• **Input and Output count** (`mathworks.metrics.ExplicitIOCount`) • **Parameter count** (`mathworks.metrics.ParameterCount`)	• Number of inputs and outputs by component (includes trigger ports) • Number of parameters by component

Modeling Guideline Compliance

For this particular system, the model compliance widgets indicate the level of compliance with industry standards and guidelines. This table lists the Metrics Dashboard widgets related to modeling guideline compliance and the detailed information available when you drill into the widget.

Widget	Metric	Drill-In Data
High Integrity Compliance	**Model Advisor standards check compliance - High Integrity** (`mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178`)	For each component: • Percentage of checks passed • Status of each check Integration with the Model Advisor for more detailed results.
MAB Compliance	**Model Advisor standards check compliance - MAB** (`mathworks.metrics.ModelAdvisorCheckCompliance.maab`)	For each component: • Percentage of checks passed • Status of each check Integration with the Model Advisor for more detailed results.

Widget	Metric	Drill-In Data
High Integrity Check Issues	**Model Advisor standards issues - High Integrity** (`mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178`)	• Number of compliance check issues by component (see the following Note below). • Components without issues or aggregated issues are not listed.
MAB Check Issues	**Model Advisor standards issues - MAB** (`mathworks.metrics.ModelAdvisorCheckIssues.maab`)	• Number of compliance check issues by component (see the following Note below). • Components without issues or aggregated issues are not listed.
Code Analyzer Warnings	**Warnings from MATLAB Code Analyzer** (`mathworks.metrics.MatlabCodeAnalyzerWarnings`)	Number of Code Analyzer warnings by component.
Diagnostic Warnings	**Simulink diagnostic warning count** (`mathworks.metrics.DiagnosticWarningsCount`)	• Number of Simulink diagnostic warnings by component. • If there are warnings, at the top of the dashboard, there is a hyperlink that opens the Diagnostic Viewer.

Note An issue with a compliance check that analyzes configuration parameters adds to the issue count for the model that fails the check.

You can use the Metrics Dashboard to perform compliance and issues checking on your own group of Model Advisor checks. For more information, see “Customize Metrics Dashboard Layout and Functionality” on page 5-39.

Architecture

These widgets provide a view of your system architecture:

- The **Potential Reuse/Actual Reuse** widget shows the percentage of total number of subcomponents that are clones and the percentage of total number of components that are linked library blocks. Orange indicates potential reuse. Blue indicates actual reuse.
- The other system architecture widgets use a value scale. For each value range for a metric, a colored bar indicates the number of components that fall within that range. Darker colors indicate more components.

This table lists the Metrics Dashboard widgets related to architecture and the detailed information available when you select the widget.

Widget	Metric	Drill-In Data
Potential Reuse / **Actual Reuse**	**Potential Reuse**(`mathworks.metrics.CloneContent`) and **Actual Reuse**(`mathworks.metrics.LibraryContent`)	Fraction of total number of subcomponents that are clones as a percentage Fraction of total number of components that are linked library blocks as a percentage Integrate with the Identify Modeling Clones tool by clicking the **Open Conversion Tool** button.
Model Complexity	**Cyclomatic complexity** (`mathworks.metrics.CyclomaticComplexity`)	Model complexity by component
Blocks	**Simulink block count** (`mathworks.metrics.SimulinkBlockCount`)	Number of blocks by component
Stateflow LOC	**Effective lines of code for Stateflow blocks** (`mathworks.metrics.StateflowLOCCount`)	Effective lines of code for Stateflow blocks by component
MATLAB LOC	**Effective lines of MATLAB code** (`mathworks.metrics.MatlabLOCCount`)	Effective lines of code, in MATLAB Function block and MATLAB functions in Stateflow, by component

Metric Thresholds

For the Model Complexity, Modeling Guideline Compliance, and Reuse widgets, the Metrics Dashboard contains default threshold values. These values indicate whether your data is Compliant or requires review (Warning). For Compliant data, the widget contains green. For warning data, the widget contains yellow. Widgets that do not have Metric threshold values contain blue.

- For the Modeling Guideline Compliance metrics, the metric threshold value is zero Model Advisor issues. If you model has issues, the widgets contain yellow. If there are no issues, the widgets contain green.
- If your model has warnings, the **Code Analyzer** and **Diagnostic** widgets are yellow. If there are no warnings, the widgets contain green.
- For the reuse widgets, the metric threshold value is zero. If your model has potential clones, the widget contains yellow. If there are no potential clones, the widget contains green.
- For the **Model Complexity** widget, the metric threshold value is 30. If your model has a cyclomatic complexity greater than 30, the widget contains yellow. If the value is less than or equal to 30, the widget contains green.

You can specify your own metric threshold values for all of the widgets in the Metrics Dashboard. You can also specify values corresponding to a noncompliant range. For more information, see “Customize Metrics Dashboard Layout and Functionality” on page 5-39.

Dashboard Limitations

When using the Metrics Dashboard, note these considerations:

- The analysis root for the Metrics Dashboard cannot be a Configurable Subsystem block.
- The Model Advisor, a tool that the Metrics Dashboard uses for data collection, cannot have more than one open session per model. For this reason, when the dashboard collects data, it closes an existing Model Advisor session.
- If you use an `sl_customization.m` file to customize Model Advisor checks, these customizations can change your dashboard results. For example, if you hide Model Advisor checks that the dashboard uses to collect metrics, the dashboard does not collect results for those metrics.
- When the dashboard collects metrics that require a model compilation, the software changes to a temporary folder. Because of this folder change, relative path dependencies in your model can become invalid.
- The Metrics Dashboard does not count MAB checks that are not about blocks as issues. Examples include checks that warn about font formatting or file names. In the Model Advisor Check Issues widget, the tool might report zero MAB issues, but still report issues in the MAB Modeling Guideline Compliance widget. For more information about these issues, click the MAB Modeling Guideline Compliance widget.

See Also

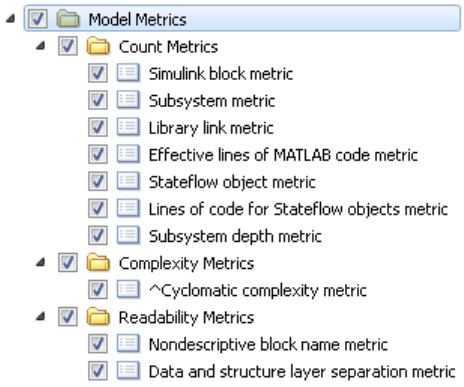
More About

- “Collect Model Metrics Programmatically” on page 5-15
- “Model Metrics”
- “Collect Compliance Data and Explore Results in the Model Advisor” on page 5-24
- “Collect Metric Data Programmatically and View Data Through the Metrics Dashboard” on page 5-29

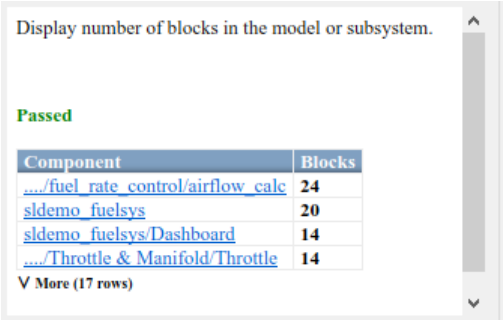
Collect Model Metrics Using the Model Advisor

To help you assess your model for size, complexity, and readability, you can run model metrics in the Model Advisor **By Task > Model Metrics** subfolder.

1. Open the sldemo_fuelsys model.
2. In the model window, open the Modeling tab and click **Model Advisor**. A System Selector — Model Advisor dialog box opens. Click **OK**.
3. In the left pane of the Model Advisor, navigate to **By Task > Model Metrics**. Select the model metrics to run on your model.



4. Click **Run Selected Checks**.
5. After the Model Advisor runs an analysis, in the left pane of the Model Advisor window, select a model metric to explore the result. Select the metric **Simulink block metric**. A summary table provides the number of blocks at the root model level and subsystem level.



   Alternatively, you can view the analysis results in the Model Advisor report.

After reviewing the metric results, you can update your model to meet size, complexity, and readability recommendations.

See Also

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-18

- “Collect Model Metrics Programmatically” on page 5-15
- “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11
- “Run Model Advisor Checks and Review Results” on page 3-4

Create a Custom Model Metric for Nonvirtual Block Count

This example shows how to use the model metric API to create a custom model metric for counting nonvirtual blocks in a model. After creating the metric, you can collect data for the metric, access the results, and export the results.

Create Metric Class

To create a custom model metric, use the `slmetric.metric.createNewMetricClass` function to create a new metric class derived from the base class `slmetric.metric.Metric`. The slmetric.metric.createNewMetricClass function creates a file that contains a constructor and an empty metric algorithm method.

1. For this example, make sure that you are in a writeable folder and create a new metric class named `nonvirtualblockcount`.

```
className = 'nonvirtualblockcount';
slmetric.metric.createNewMetricClass(className);
```

2. Write the metric algorithm into the `slmetric.metric.Metric` method, `algorithm`. The algorithm calculates the metric 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. For this example, the file `nonvirtualblockcount_orig.m` contains the logic to create a metric that counts the nonvirtual blocks. Copy this file to the `nonvirtualblockcount.m` file.

```
copyfile nonvirtualblockcount_orig.m nonvirtualblockcount.m f
```

When creating a custom metric, you must set the following properties of the slmetric.metric.Metric class:

- `ID`: Unique metric identifier that retrieves the new metric data.
- `Name`: Name of the metric algorithm.
- `ComponentScope`: Model components for which the metric is calculated.
- `CompileContext`: Compile mode for metric calculation. If your model requires model compilation, specify `PostCompile`. Collecting metric data for compiled models slows performance.
- `ResultCheckSumCoverage`: Specify whether you want the metric data regenerated if source file and `Version` have not changed.
- `AggregationMode`: How the metric algorithm aggregates metric data

Optionally, you can set these additional properties:

- `Description`: Description of the metric.
- `Version`: Metric version.

3. Now that your new model metric is defined in nonvirtualblockcount.m, you can register the new metric in the metric repository.

```
[id_metric,err_msg] = slmetric.metric.registerMetric(className);
```

Collect Metric Data

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 `sldemo_mdlref_bus` model.

1. Load the sldemo_mdlref_bus model.

```
model = 'sldemo_mdlref_bus';
load_system(model);
```

2. Create a metric engine object and set the analysis root.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root',model,'RootType','Model');
```

3. Collect metric data for the nonvirtual block count metric.

```
execute(metric_engine);
rc = getMetrics(metric_engine,id_metric);
```

Display and Export Results

To access the metrics for your model, use instance of `slmetric.metric.Result`. In this example, display the nonvirtual block count metrics for the sldemo_mdlref_bus 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
    disp(' ');
end
```

```
MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_bus
  Value: 13

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_bus/More Info3
  Value: 0

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_bus/More Info4
  Value: 0

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_bus/More Info1
  Value: 0
```

```
MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_bus/More Info2
  Value: 0

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus
  Value: 2

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/COUNTER
  Value: 6

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/COUNTER/Counter
  Value: 3

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/COUNTER/Counter/ResetCheck
  Value: 4

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/COUNTER/Counter/ResetCheck/NoReset
  Value: 2

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/COUNTER/Counter/ResetCheck/Reset
  Value: 3

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/COUNTER/Counter/SaturationCheck
  Value: 5

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/COUNTER/LimitsProcess
  Value: 1

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/More Info1
  Value: 0

MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_counter_bus/More Info2
  Value: 0
```

To export 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='MyMetricData.xml';
exportMetrics(metric_engine,filename);
```

For this example, unregister the nonvirtual block count metric.

```
slmetric.metric.unregisterMetric(id_metric);
```

Close the model.

```
clear;
bdclose('all');
```

Limitations

Custom metric algorithms do not support the path property on component objects:

- Linked Stateflow charts
- MATLAB Function blocks

Custom metric algorithms do not follow library links.

Copyright 2019 The MathWorks, Inc.

See Also

`Advisor.component.Component` | `Advisor.component.Types` | `slmetric.Engine` | `slmetric.metric.Metric` | `slmetric.metric.Result` | `slmetric.metric.createNewMetricClass`

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-18
- “Collect Model Metrics Programmatically” on page 5-15

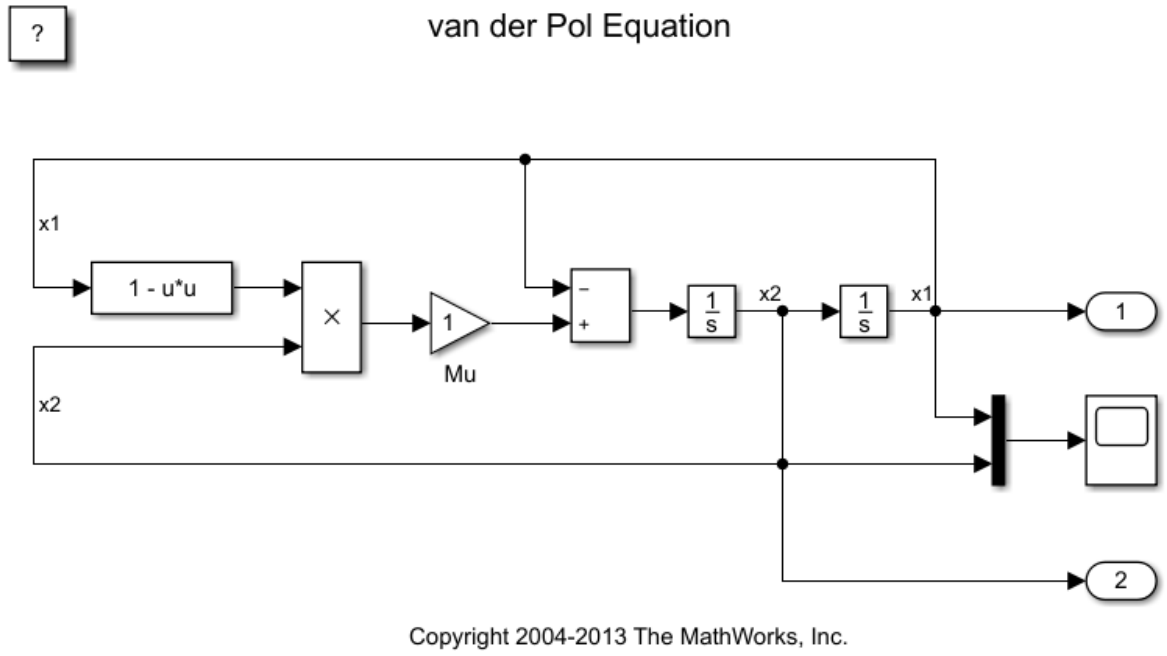
Collect Model Metrics Programmatically

This example shows how to use the model metric API to programmatically collect subsystem and block count metrics for a model. After collecting metrics for the model, you can access the results and export them to a file.

Example Model

Open model vdp.

```
model = 'vdp';
open_system(model);
```



Collect Metrics

To collect metric data on a model, create a metric engine object and call `execute`.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root','vdp','RootType','Model');
execute(metric_engine);
```

```
Updating Model Advisor cache...
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.Manage
```

Access Results

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. `getMetrics` returns an array of `slmetric.metric.ResultCollection` objects.

```
res_col = getMetrics(metric_engine,{'mathworks.metrics.SimulinkBlockCount',...
'mathworks.metrics.SubSystemCount'});
```

Store and Display Results

Create cell array `metricData` to store the `MetricID`, `ComponentPath`, and `Value` for the metric results. The `MetricID` is the identifier for the metric, the `ComponentPath` is the path to component for which the metric is calculated, and the `Value` is the metric value. Write a loop to display the results.

```
metricData ={'MetricID','ComponentPath','Value'};
cnt = 1;
for n=1:length(res_col)
    if res_col(n).Status == 0
        results = res_col(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:',res_col(n).MetricID]);
    end
    disp(' ');
end
```

```
MetricID: mathworks.metrics.SimulinkBlockCount
  ComponentPath: vdp
  Value: 11
MetricID: mathworks.metrics.SimulinkBlockCount
  ComponentPath: vdp/More Info
  Value: 1
MetricID: mathworks.metrics.SimulinkBlockCount
  ComponentPath: vdp/More Info/Model Info
  Value: 1
MetricID: mathworks.metrics.SimulinkBlockCount
  ComponentPath: vdp/More Info/Model Info/EmptySubsystem
  Value: 0

MetricID: mathworks.metrics.SubSystemCount
  ComponentPath: vdp
  Value: 1
MetricID: mathworks.metrics.SubSystemCount
  ComponentPath: vdp/More Info
  Value: 0
MetricID: mathworks.metrics.SubSystemCount
  ComponentPath: vdp/More Info/Model Info
  Value: 1
MetricID: mathworks.metrics.SubSystemCount
  ComponentPath: vdp/More Info/Model Info/EmptySubsystem
  Value: 0
```

Export Results

To export the `metricData` results `MetricID`, `ComponentPath`, and `Value` to a spreadsheet, use `writetable` to write the contents of `metricData` to `MySpreadsheet.xlsx`.

```
filename = 'MySpreadsheet.xlsx';
T=table(metricData);
writetable(T,filename);
```

To export 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='MyMetricResults.xml';
exportMetrics(metric_engine,filename)
```

Close the model `vdp`.

```
bdclose(model);
```

Limitations

For one model, you cannot collect metric data into the same database file (that is, the `Metrics.db` file) on multiple platforms.

See Also

`slmetric.Engine` | `slmetric.metric.Result` | `slmetric.metric.ResultCollection`

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-18
- “Collect Model Metrics Using the Model Advisor” on page 5-9
- “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11

Model Metric Data Aggregation

You can better understand the size, complexity, and readability of a model and its components by analyzing aggregated model metric data. Aggregated metric data is available in the `AggregatedValue` and `AggregatedMeasures` properties of an `slmetric.metric.Result` object. The `AggregatedValue` property aggregates the metric scalar values. The `AggregatedMeasures` property aggregates the metric measures (that is, the detailed information about the metric values).

How Model Metric Aggregation Works

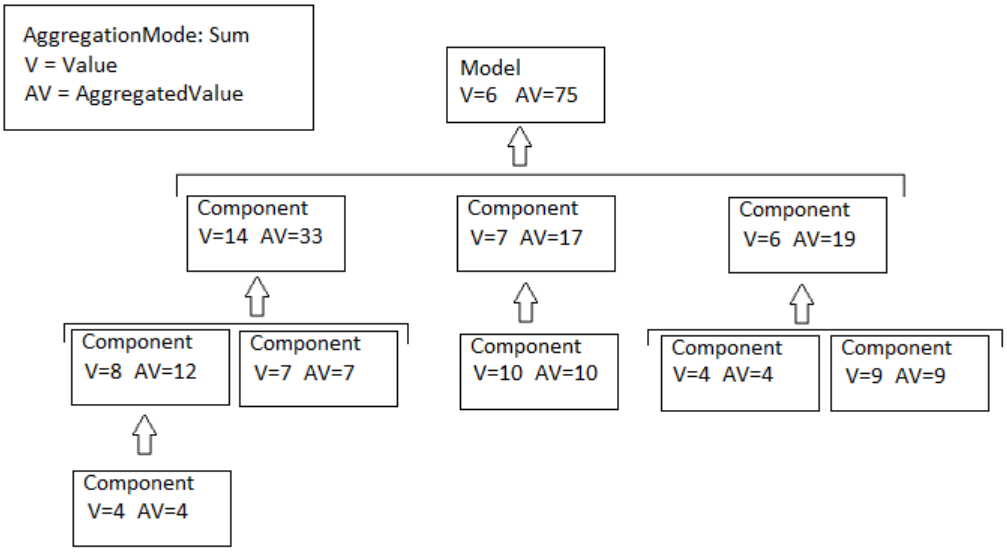
The implementation of a model metric defines how a metric aggregates data across a component hierarchy. For MathWorks model metrics, the `slmetric.metric.Metric` class defines model metric aggregation. This class includes the `AggregationMode` property, which has these options:

- `Sum`: Returns the sum of the `Value` property and the `Value` properties of its children components across the component hierarchy. Returns the sum of the `Meaures` property and the `Measures` properties of its children components across the component hierarchy.
- `Max`: Returns the maximum of the `Value` property and the `Value` properties of its children components across the component hierarchy. Returns the maximum of the `Measures` property and the `Measures` properties of its children components across the component hierarchy.
- `None`: No aggregation of metric values.

You can find descriptions of MathWorks model metrics and their `AggregationMode` property setting in “Model Metrics”. For custom metrics, as part of the `algorithm` method, you can define how the metric aggregates data. For more information, see “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11.

This diagram shows how the software aggregates metric data across the components of a model hierarchy. The parent model is at the top of the hierarchy. The components can be the following:

- Model
- Subsystem block
- Chart
- MATLAB function block
- Protected model



Access Aggregated Metric Data

This example shows how to collect metric data programmatically in the metric engine, and then access aggregated metric data.

1. Load the `sldemo_applyVarStruct` model.

```
model = 'sldemo_applyVarStruct';
open(model);
load_system(model);
```

2. Create an `slmetric.Engine` object and set the analysis root.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root',model,'RootType','Model');
```

3. Collect data for the Input output model metric.

```
execute(metric_engine,'mathworks.metrics.IOCount');
```

4. Get the model metric data that returns an array of `slmetric.metric.ResultCollection` objects, `res_col`. Specify the input argument for `AggregationDepth`.

```
res_col = getMetrics(metric_engine,'mathworks.metrics.IOCount',...
'AggregationDepth','All');
```

   The `AggregationDepth` input argument has two options: `All` and `None`. If you do not want the `getMetrics` method to aggregate measures and values, specify `None`.

5. Display the results.

```
metricData ={'MetricID','ComponentPath','Value',...
   'AggregatedValue','Measures','AggregatedMeasures'};
cnt = 1;
for n=1:length(res_col)
    if res_col(n).Status == 0
        results = res_col(n).Results;
```

```
            for m=1:length(results)
                disp(['MetricID: ',results(m).MetricID]);
                disp(['  ComponentPath: ',results(m).ComponentPath]);
                disp(['  Value: ',num2str(results(m).Value)]);
                disp(['  Aggregated Value: ',num2str(results(m).AggregatedValue)]);
                disp(['  Measures: ',num2str(results(m).Measures)]);
                disp(['  Aggregated Measures: ',...
                    num2str(results(m).AggregatedMeasures)]);
                metricData{cnt+1,1} = results(m).MetricID;
                metricData{cnt+1,2} = results(m).ComponentPath;
                metricData{cnt+1,3} = results(m).Value;
                tdmetricData{cnt+1,4} = results(m).Measures;
                metricData{cnt+1,5} = results(m).AggregatedMeasures;
                cnt = cnt + 1;
            end
        else
            disp(['No results for:',res_col(n).MetricID]);
        end
        disp(' ');
    end
```

Here are the results:

```
MetricID: mathworks.metrics.IOCount
  ComponentPath: sldemo_applyVarStruct
  Value: 3
  Aggregated Value: 5
  Measures: 1  2  0  0
  Aggregated Measures: 3  2  0  0
MetricID: mathworks.metrics.IOCount
  ComponentPath: sldemo_applyVarStruct/Controller
  Value: 4
  Aggregated Value: 4
  Measures: 3  1  0  0
  Aggregated Measures: 3  1  0  0
MetricID: mathworks.metrics.IOCount
  ComponentPath: sldemo_applyVarStruct/Aircraft
Dynamics
Model
  Value: 5
  Aggregated Value: 5
  Measures: 3  2  0  0
  Aggregated Measures: 3  2  0  0
MetricID: mathworks.metrics.IOCount
  ComponentPath: sldemo_applyVarStruct/Dryden Wind
Gust Models
  Value: 2
  Aggregated Value: 2
  Measures: 0  2  0  0
  Aggregated Measures: 0  2  0  0
MetricID: mathworks.metrics.IOCount
  ComponentPath: sldemo_applyVarStruct/Nz pilot
calculation
  Value: 3
  Aggregated Value: 3
  Measures: 2  1  0  0
  Aggregated Measures: 2  1  0  0
MetricID: mathworks.metrics.IOCount
```

```
  ComponentPath: sldemo_applyVarStruct/More Info2
  Value: 0
  Aggregated Value: 0
  Measures: 0  0  0  0
  Aggregated Measures: 0  0  0  0
```

For the Input output metric, the `AggregationMode` is `Max`. For each component, the `AggregatedValue` and `AggregatedMeasures` properties are the maximum number of inputs and outputs of itself and its children components. For example, for `sldemo_applyVarStruct`, the `AggregatedValue` property is `5`, which is the `sldemo_applyVarStruct/Aircraft Dynamics Model` component value.

See Also

`slmetric.Engine` | `slmetric.metric.Metric` | `slmetric.metric.Result` | `slmetric.metric.ResultCollection`

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-18
- “Collect Model Metrics Using the Model Advisor” on page 5-9
- “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11

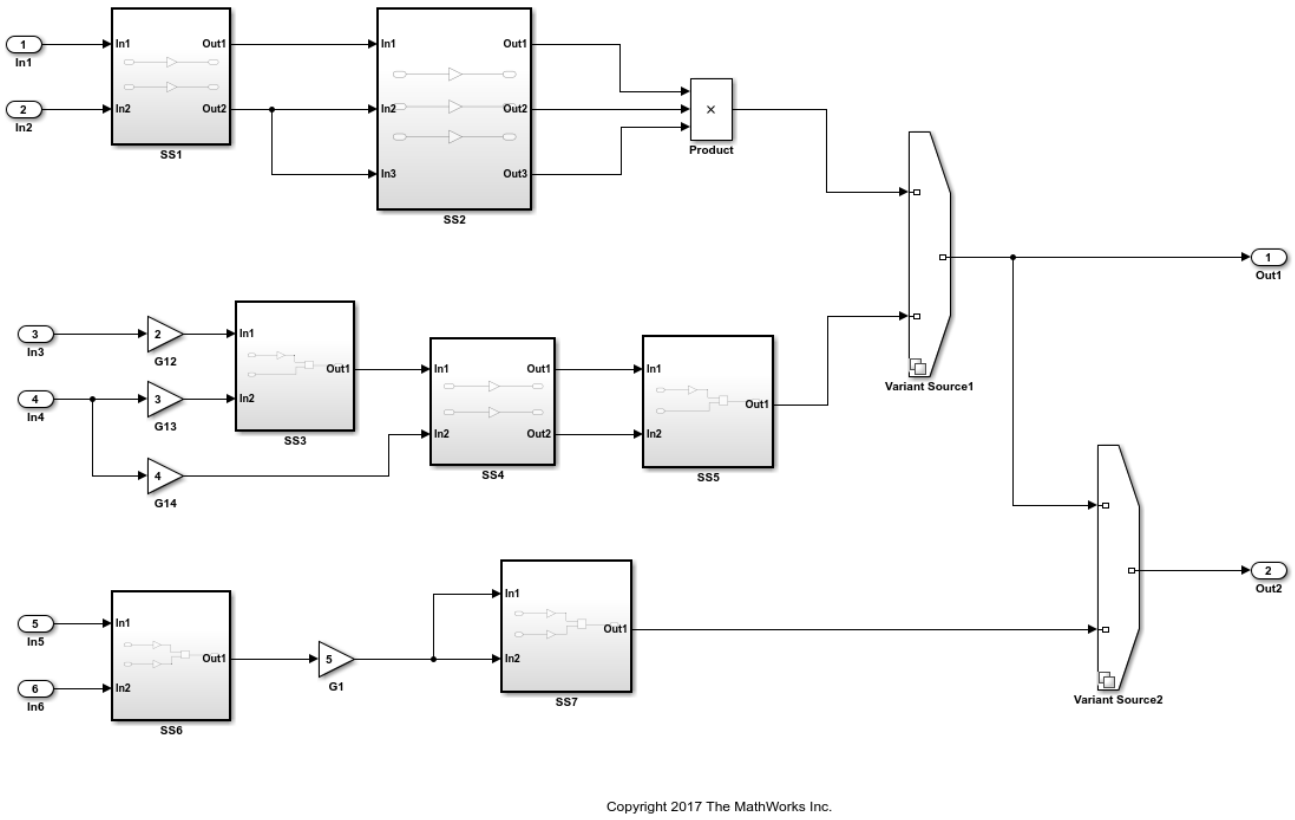
Identify Modeling Clones with the Metrics Dashboard

You can use the Metrics Dashboard tool to help you reuse subsystems by identifying clones across a model hierarchy. Clones are identical MATLAB Function blocks, identical Stateflow charts, and subsystems that have identical block types and connections. The clones can have different parameter settings and values. To replace clones with links to library blocks, from the Metrics Dashboard, you can open the Clone Detector app.

Identify Clones

To open the example model `ex_clone_detection`, change your directory to the `matlabroot\help\toolbox\simulink\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox','simulink','examples'));
```



1. Save the `ex_clone_detection.slx` model to a local working folder.
2. On the **Apps** tab, click **Metrics Dashboard**.
3. In the Metrics Dasbhoard, click **All Metrics**.
4. In the **Architecture** section, the yellow bar in the **Potential Reuse** row indicates that the model contains clones. The percentage is the fraction of the total number of subsystems, including Stateflow Charts and MATLAB Function blocks, that are clones. To see details, click the yellow bar.

The model contains three clone groups. `SS1` and `SS4` are part of clone group one. `SS3` and `SS5` are part of clone group two. `SS6` and `SS7` are part of clone group three.

Replace Clones with Links to Library Blocks

1. To replace clones with links to library blocks, open the Clone Detector app by clicking **Open Conversion Tool**. The Clone Detection app opens. For more information on the app, see “Enable Component Reuse by Using Clone Detection” on page 3-26.
2. Click **Find Clones**. The app contains a list of clone groups on the **Clone Detection Actions and Results** pane, on the **Map Clone Groups to Library** tab.
3. Click **Replace Clones**. The Clone Detector app replaces the clones with links to library blocks. The library blocks are in the library specified by the **Library to place clones** parameter. This parameter is on the **Map Clone Groups to Library** tab. The library is on the MATLAB path. It has a default name of `newLibraryFile`.

After you refactor, you can remove the latest changes from the model. In the **Clone Detection Actions and Results** pane, in the **Logs** tab, open the latest log and click **Restore**. Each time you refactor a model, the tool creates a backup model in the folder named with the prefix `m2m_` plus the model name.

If you have a Simulink Test license, you can verify the equivalency of the refactored model and the original model. Click **Check Equivalency**.

Run Model Metrics on the Refactored Model

1. Navigate to the **Metrics Dashboard**.
2. Click **All Metrics**.
3. In the **Architecture** section, the blue bar in the **Actual Reuse** row indicates that 75% of model components are links to library subsystems. The **Potential Reuse** row indicates that the model does not contain any clones that do not have links to library blocks.

See Also

More About

- “Collect Model Metrics”

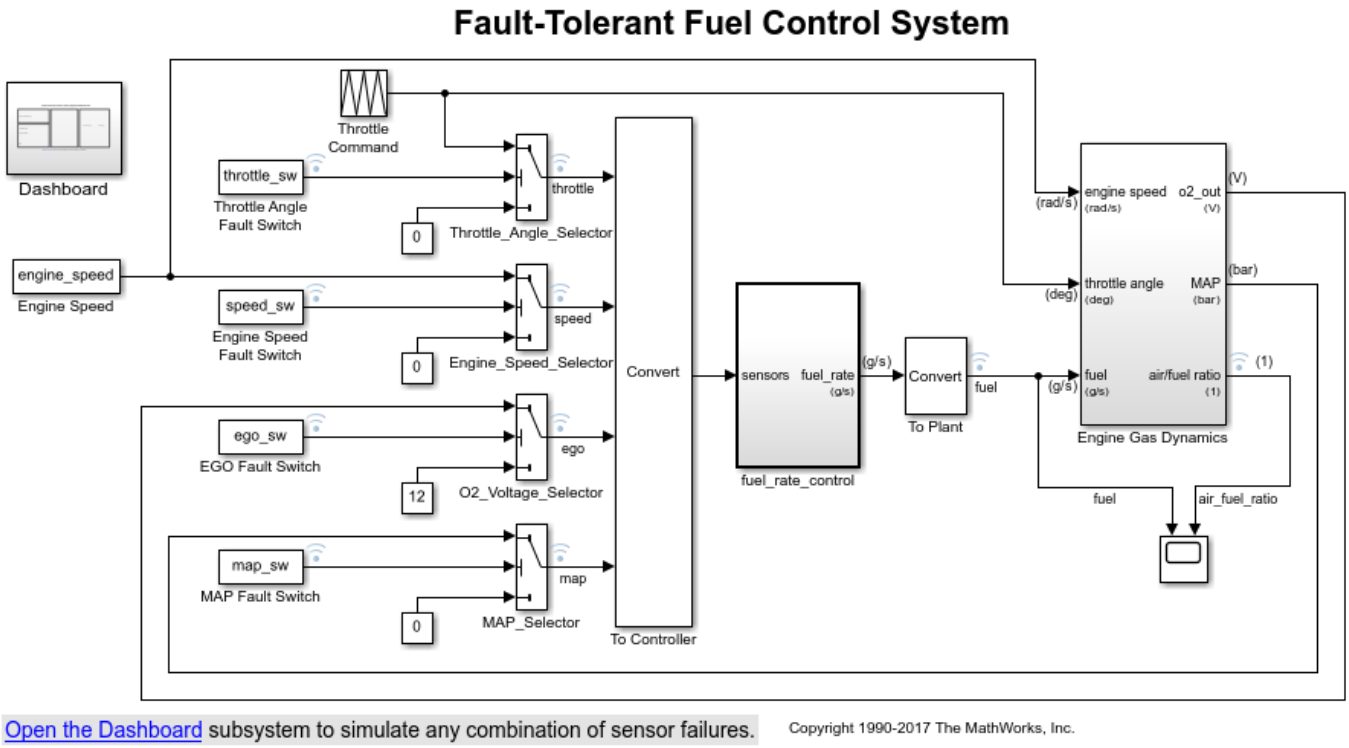
Collect Compliance Data and Explore Results in the Model Advisor

This example shows how to collect model metric data by using the Metrics Dashboard. From the dashboard, explore detailed compliance results and, fix compliance issues by using the Model Advisor.

Open the Example Model

Open the example model `sldemo_fuelsys` and save the model to a local folder.

```
open_system('sldemo_fuelsys');
```



Open the Metrics Dashboard

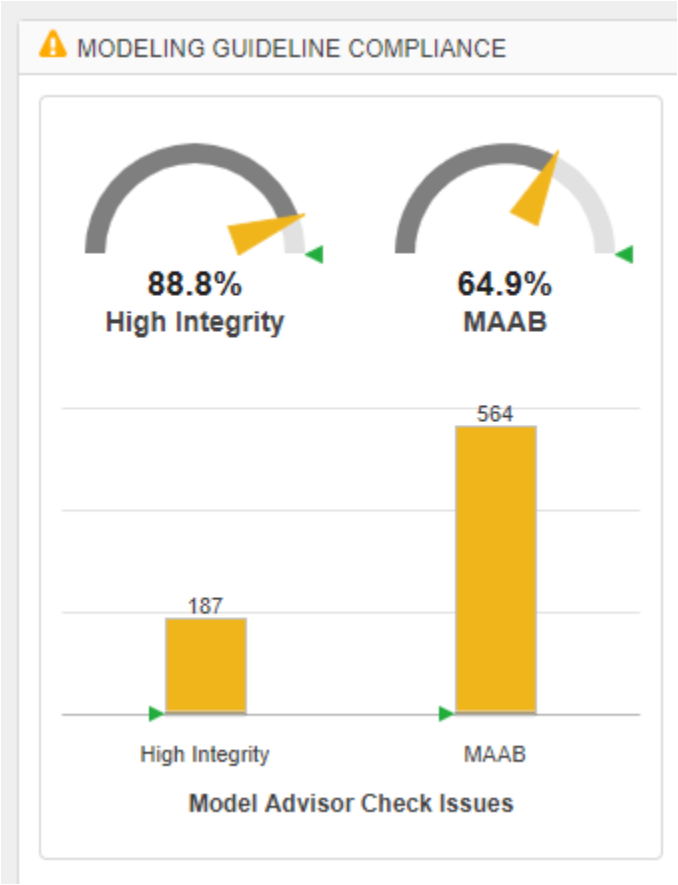
On the **Apps** tab, open the Metrics Dashboard by clicking **Metrics Dashboard**.

Collect Model Metrics

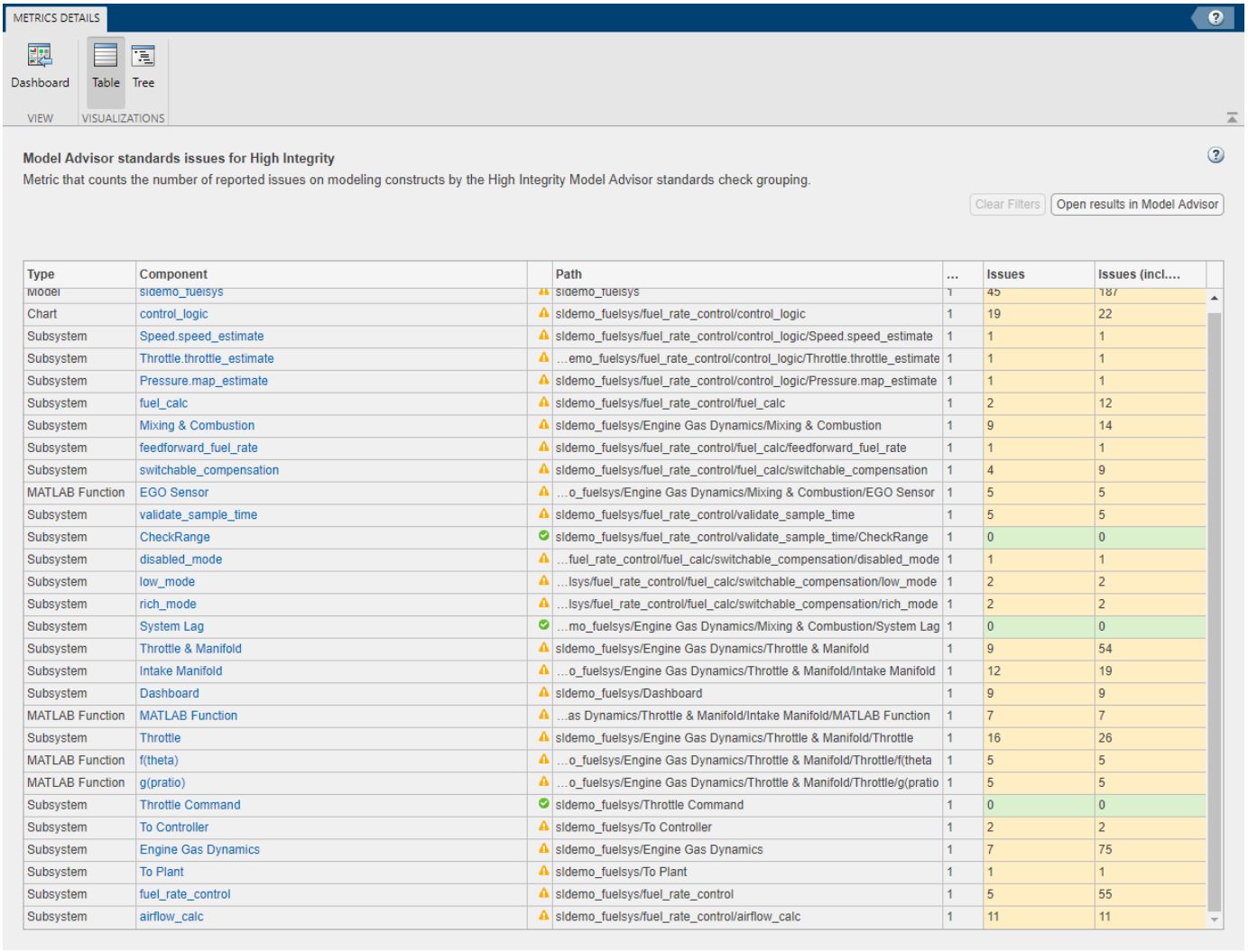
To collect the metric data for this model, click the **All Metrics** icon.

Explore Compliance Results

Locate the **MODELING GUIDELINE COMPLIANCE** section of the dashboard. This section displays the percentage of High Integrity and MAB compliance checks that passed on all systems. The bars chart show the number of issues reported by the checks in the corresponding check group.

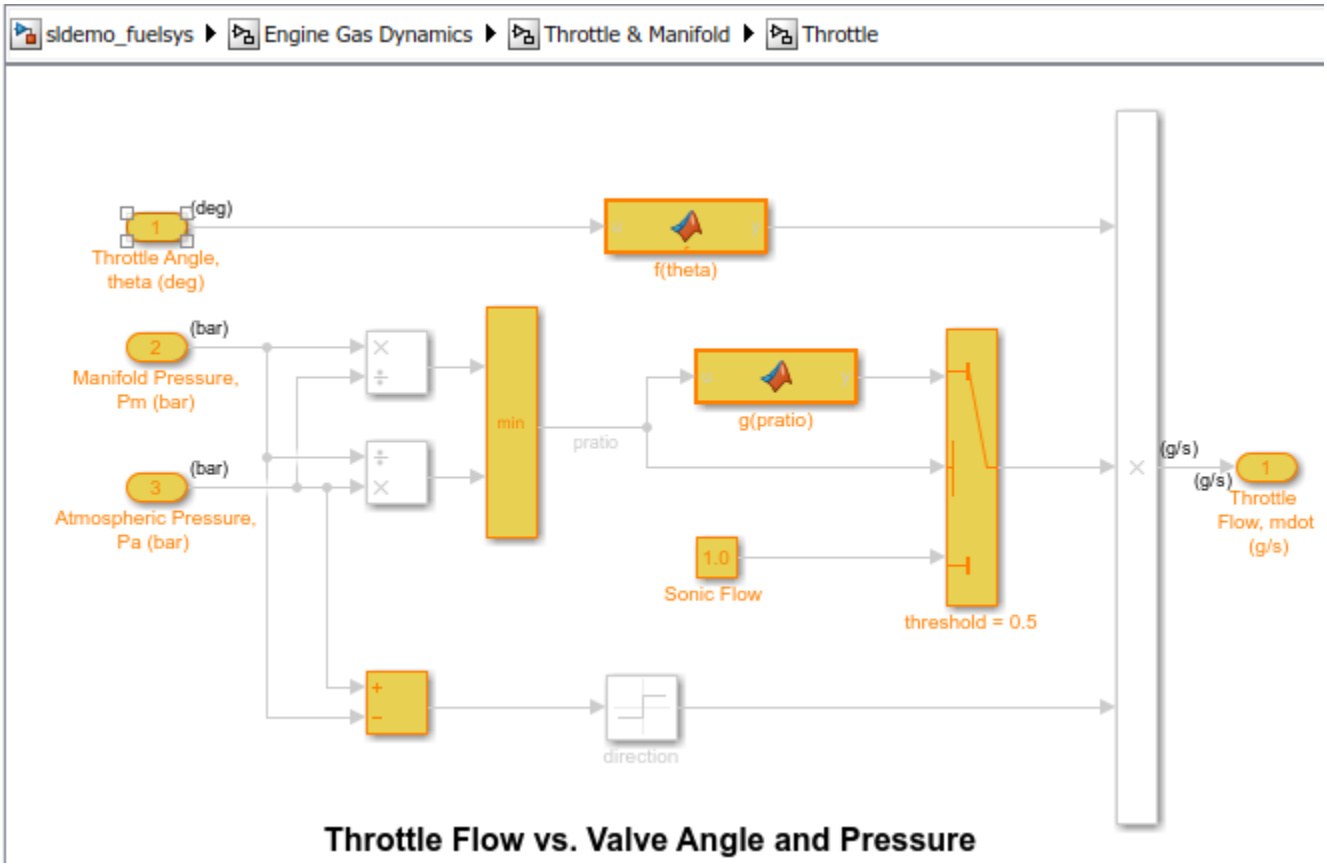


To see a table that details the number of compliance issues by component, click anywhere on the **High Integrity** bar chart. For compliance checks that analyze configuration settings, each check that does not pass adds 1 issue to the model on which it failed.



Type	Component		Path	...	Issues	Issues (incl....
Model	sldemo_fuelsys	⚠	sldemo_fuelsys	1	45	187
Chart	control_logic	⚠	sldemo_fuelsys/fuel_rate_control/control_logic	1	19	22
Subsystem	Speed.speed_estimate	⚠	sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate	1	1	1
Subsystem	Throttle.throttle_estimate	⚠	...emo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate	1	1	1
Subsystem	Pressure.map_estimate	⚠	sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate	1	1	1
Subsystem	fuel_calc	⚠	sldemo_fuelsys/fuel_rate_control/fuel_calc	1	2	12
Subsystem	Mixing & Combustion	⚠	sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	9	14
Subsystem	feedforward_fuel_rate	⚠	sldemo_fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate	1	1	1
Subsystem	switchable_compensation	⚠	sldemo_fuelsys/fuel_rate_control/fuel_calc/switchable_compensation	1	4	9
MATLAB Function	EGO Sensor	⚠	...o_fuelsys/Engine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	5	5
Subsystem	validate_sample_time	⚠	sldemo_fuelsys/fuel_rate_control/validate_sample_time	1	5	5
Subsystem	CheckRange	✓	sldemo_fuelsys/fuel_rate_control/validate_sample_time/CheckRange	1	0	0
Subsystem	disabled_mode	⚠	...fuel_rate_control/fuel_calc/switchable_compensation/disabled_mode	1	1	1
Subsystem	low_mode	⚠	...lsys/fuel_rate_control/fuel_calc/switchable_compensation/low_mode	1	2	2
Subsystem	rich_mode	⚠	...lsys/fuel_rate_control/fuel_calc/switchable_compensation/rich_mode	1	2	2
Subsystem	System Lag	✓	...mo_fuelsys/Engine Gas Dynamics/Mixing & Combustion/System Lag	1	0	0
Subsystem	Throttle & Manifold	⚠	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	9	54
Subsystem	Intake Manifold	⚠	...o_fuelsys/Engine Gas Dynamics/Throttle & Manifold/Intake Manifold	1	12	19
Subsystem	Dashboard	⚠	sldemo_fuelsys/Dashboard	1	9	9
MATLAB Function	MATLAB Function	⚠	...as Dynamics/Throttle & Manifold/Intake Manifold/MATLAB Function	1	7	7
Subsystem	Throttle	⚠	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	16	26
MATLAB Function	f(theta)	⚠	...o_fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle/f(theta	1	5	5
MATLAB Function	g(pratio)	⚠	...o_fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio	1	5	5
Subsystem	Throttle Command	✓	sldemo_fuelsys/Throttle Command	1	0	0
Subsystem	To Controller	⚠	sldemo_fuelsys/To Controller	1	2	2
Subsystem	Engine Gas Dynamics	⚠	sldemo_fuelsys/Engine Gas Dynamics	1	7	75
Subsystem	To Plant	⚠	sldemo_fuelsys/To Plant	1	1	1
Subsystem	fuel_rate_control	⚠	sldemo_fuelsys/fuel_rate_control	1	5	55
Subsystem	airflow_calc	⚠	sldemo_fuelsys/fuel_rate_control/airflow_calc	1	11	11

From the table, open the `Throttle` component in the model editor by clicking the component hyperlink in the table. The model editor highlights blocks in the component that have compliance issues.



Explore Compliance Results in the Model Advisor

1. In the Metrics Dashboard, return to the main dashboard page by clicking the **Dashboard** icon.
2. Click the **High Integrity** percentage gauge.
3. To see the status for each compliance check, click the **Table** view.
4. Expand the `sldemo_fuelsys` node.
5. To explore check results in more detail, click the **Check safety-related diagnostic settings for sample time** hyperlink.
6. In the Model Advisor Highlight dialog box, click **Check safety-related diagnostic settings for sample time** hyperlink.

Fix a Compliance Issue

1. In the Model Advisor Report, the check results show the Current Value and Recommended Value of diagnostic parameters.
2. To change the Current Value to the Recommended Value, click the parameter. The Model Configuration Parameters dialog box opens.
3. Change the parameter settings.
4. Save your changes and close the dialog box.
5. Save the changes to the model.

Recollect Metrics

1. Return to the Metrics Dashboard.
2. To recollect the model metrics, click the **All Metrics** icon.
3. To return to the main dashboard page, click the **Dashboard** icon.
4. Confirm that the number of **High Integrity** check issues is reduced.

See Also

More About

- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2
- “Collect Model Metrics Programmatically” on page 5-15

Collect Metric Data Programmatically and View Data Through the Metrics Dashboard

This example shows how to use the model metrics API to collect model metric data for your model, and then explore the results by using the Metrics Dashboard.

Collect Metric Data Programmatically

To collect all of the available metrics for the model `sldemo_fuelsys`, use the `slmetric.Engine` API. The metrics engine stores the results in the metric repository file in the current Simulation Cache Folder, `slprj`.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root','sldemo_fuelsys','RootType','Model');
evalc('execute(metric_engine)');
```

Determine Model Compliance with MAB Guidelines

To determine the percentage of MAB checks that pass, use the metric compliance results.

```
metricID = 'mathworks.metrics.ModelAdvisorCheckCompliance.maab';
metricResult = getAnalysisRootMetric(metric_engine, metricID);
disp(['MAAB compliance: ', num2str(100 * metricResult.AggregatedValue, 3),'%']);
```

```
MAAB compliance: 65.6%
```

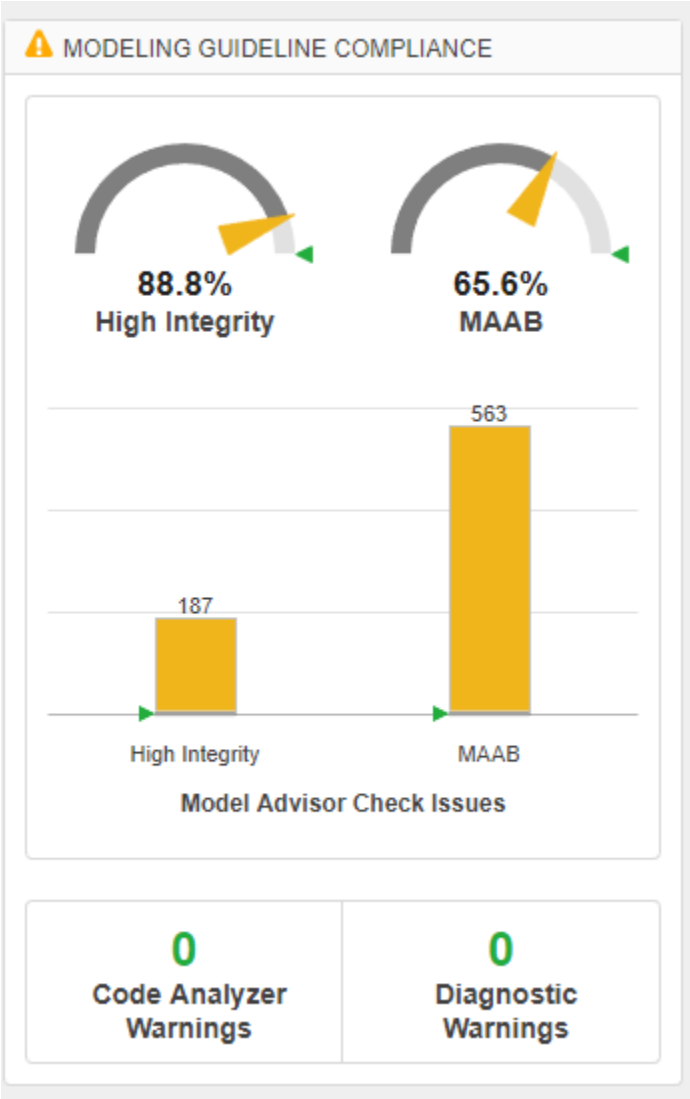
Open the Metrics Dashboard

To explore the collected compliance metrics in more detail, open the Metrics Dashboard for the model.

```
metricsdashboard('sldemo_fuelsys');
```

The Metrics Dashboard opens data for the model from the active metric repository, inside the active Simulation Cache Folder. To view the previously collected data, the `slprj` folder must be the same.

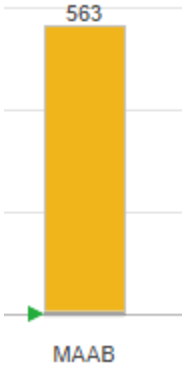
Find the **MODELING GUIDELINE COMPLIANCE** section of the dashboard. For each category of compliance checks, the gauge indicates the percentage of compliance checks that passed.



The dashboard reports the same MAB compliance percentage as the `slmetric.Engine` API reports.

Explore the MAB Compliance Results

Underneath the percentage gauges, the bar chart indicates the number of compliance check issues. Click anywhere in the MAB bar chart for Model Advisor Check Issues.



The table details the number of check issues per model component. To sort the components by number of check issues, click the **Issues** column.

Model Advisor standards issues for MAAB

Metric that counts the number of reported issues on modeling constructs by the MAAB Model Advisor standards check grouping.

Clear Filters | Open results in Model Advisor

Type	Component		Path	...	Issues	Issues (incl....
Model	sldemo_fuelsys	⚠	sldemo_fuelsys	1	59	563
Chart	control_logic	⚠	sldemo_fuelsys/fuel_rate_control/control_logic	1	232	265
Subsystem	Speed.speed_estimate	⚠	...ys/fuel_rate_control/control_logic/Speed.speed_estimate	1	13	13
Subsystem	Throttle.throttle_estimate	⚠	.../fuel_rate_control/control_logic/Throttle.throttle_estimate	1	10	10
Subsystem	Pressure.map_estimate	⚠	...s/fuel_rate_control/control_logic/Pressure.map_estimate	1	10	10
Subsystem	fuel_calc	⚠	sldemo_fuelsys/fuel_rate_control/fuel_calc	1	5	40
Subsystem	Mixing & Combustion	⚠	...emo_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	10	10
Subsystem	feedforward_fuel_rate	⚠	...fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate	1	9	9
Subsystem	switchable_compensation	⚠	...sys/fuel_rate_control/fuel_calc/switchable_compensation	1	11	26
MATLAB Function	EGO Sensor	✓	...ngine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	0	0
Subsystem	validate_sample_time	⚠	sldemo_fuelsys/fuel_rate_control/validate_sample_time	1	3	3
Subsystem	CheckRange	✓	...sys/fuel_rate_control/validate_sample_time/CheckRange	1	0	0
Subsystem	disabled_mode	⚠	...ntrol/fuel_calc/switchable_compensation/disabled_mode	1	3	3
Subsystem	low_mode	⚠	...te_control/fuel_calc/switchable_compensation/low_mode	1	6	6
Subsystem	rich_mode	⚠	...e_control/fuel_calc/switchable_compensation/rich_mode	1	6	6
Subsystem	System Lag	✓	...Engine Gas Dynamics/Mixing & Combustion/System Lag	1	0	0
Subsystem	Throttle & Manifold	⚠	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	20	62
Subsystem	Intake Manifold	⚠	...ngine Gas Dynamics/Throttle & Manifold/Intake Manifold	1	15	15
Subsystem	Dashboard	⚠	sldemo_fuelsys/Dashboard	1	41	41
MATLAB Function	MATLAB Function	✓	...ics/Throttle & Manifold/Intake Manifold/MATLAB Function	1	0	0
Subsystem	Throttle	⚠	...elsys/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	27	27
MATLAB Function	f(theta)	✓	...ngine Gas Dynamics/Throttle & Manifold/Throttle/f(theta	1	0	0
MATLAB Function	g(pratio)	✓	...ngine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio	1	0	0
Subsystem	Throttle Command	✓	sldemo_fuelsys/Throttle Command	1	0	0
Subsystem	To Controller	⚠	sldemo_fuelsys/To Controller	1	9	9
Subsystem	Engine Gas Dynamics	⚠	sldemo_fuelsys/Engine Gas Dynamics	1	10	82
Subsystem	To Plant	⚠	sldemo_fuelsys/To Plant	1	5	5

See Also

More About

- “Collect Model Metrics Programmatically” on page 5-15
- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2

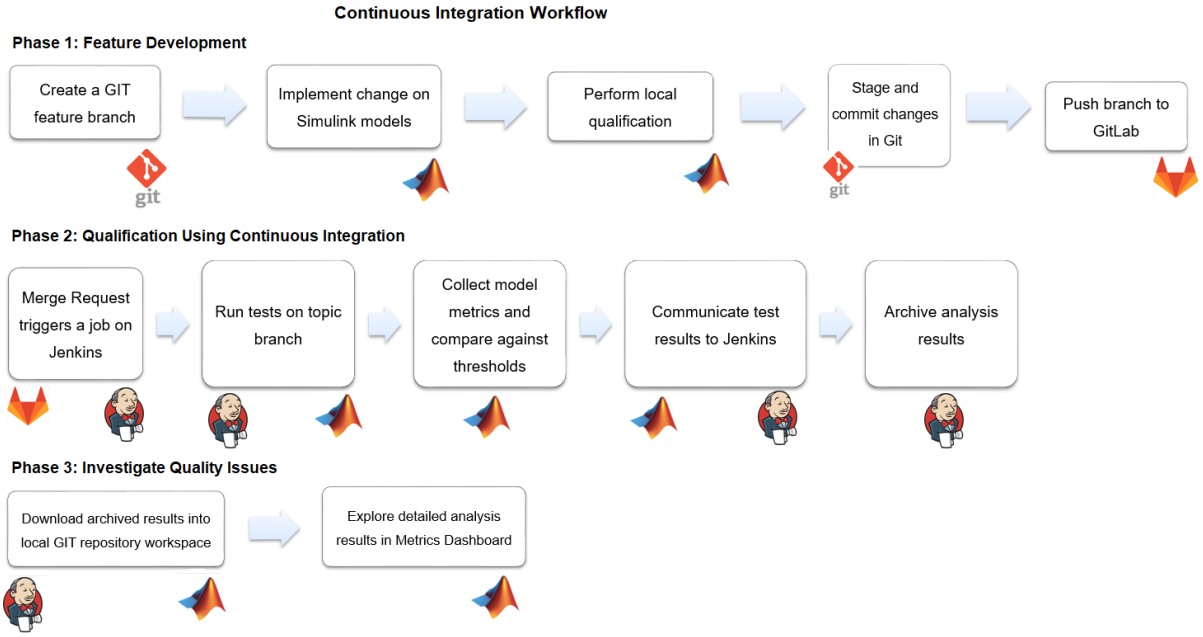
Fix Metric Threshold Violations in a Continuous Integration Systems Workflow

This example shows how to use the Metrics Dashboard with open-source tools GitLab and Jenkins to test and refine your model in a continuous integration systems workflow. Continuous integration is the practice of merging all developer working copies of project files to a shared mainline. This workflow saves time and improves quality by maintaining version control and automating and standardizing testing.

This example refers to a project that contains the shipped project matlab:sldemo_slproject_airframe and these additional files which are relevant to this example:

- A MATLAB script that specifies metric thresholds and customizes the Metrics Dashboard.
- A MATLAB unit test that collects metric data and checks whether there are metric threshold violations.

The example uses the Jenkins continuous integration server to run the MATLAB unit test to determine if there are metric threshold violations. Jenkins archives test results for you to download and investigate locally. GitLab is an online Git repository manager that you can configure to work with Jenkins. This diagram shows how Simulink Check, GitLab, and Jenkins work together in a continuous integration workflow.



Project Setup

The project contains all model, data, and configuration files including these files which are required for this example:

- A MATLAB unit test that collects metric data for the project and checks that the model files contain no metric threshold violations. For more information on the MATLAB Unit tests, see “Script-Based Unit Tests” (MATLAB).

- A `setup.m` file that activates the configuration XML files that define metric thresholds, set custom metric families, and customizes the Metrics Dashboard layout. For this example, this code is the `setup.m` script:

  ```
  function setup
      % refresh Model Advisor customizations
      Advisor.Manager.refresh_customizations();

      % set metric configuration with thresholds
      configFile = fullfile(pwd, 'config', 'MyConfiguration.xml');
      slmetric.config.setActiveConfiguration(configFile);

      uiconf = fullfile(pwd, 'config', 'MyDashboardConfiguration.xml');
      slmetric.dashboard.setActiveConfiguration(uiconf);
  end
  ```

  On the **Project** tab, click **Startup Shudown**. For the **Startup files** field, specify the `setup.m` file.

  For more information on how to customize the Metrics Dashboard, see “Customize Metrics Dashboard Layout and Functionality” on page 5-39.
- An `sl_customization.m` file that activates the Model Advisor configuration file to customize the Model Advisor checks.
- A `run` script that executes during a Jenkins build. For this example, this code is in the `run.m` file:

  ```
  % script executed during Jenkins build
  function run(IN_CI)
       if (IN_CI)
          jenkins_workspace = getenv('WORKSPACE');
          cd(jenkins_workspace);
      end

      % open the sl project
      slproj = simulinkproject(pwd);

      % execute tests
      runUnitTest();

      slproj.close();

      if IN_CI
          exit
      end
  end
  ```
- A `cleanup.m` file that resets the active metric configuration to the default configuration. For this example, this code is in the `cleanup.m` file script:

  ```
  function cleanup
      rmpath(fullfile(pwd, 'data'));
      Advisor.Manager.refresh_customizations();

      % reset active metric configuration to default
      slmetric.config.setActiveConfiguration('');
      slmetric.dashboard.setActiveConfiguration('');
  end
  ```

  On the **Project** tab, click **Startup Shudown**. For the **Shutdown files** field, specify the `cleanup.m` file.

- A `.gitignore` file that verifies that derived artifacts are not checked into GitLab. This code is in the `.gitignore` file:

  ```
  work/**
  reports/**
  *.asv
  *.autosave
  ```

GitLab Setup

Create a GitLab project for source-controlling your Project. For more information, see https://docs.gitlab.com/ee/README.html.

1. Install the Git Client.
2. Set up a branching workflow. With GitLab, from the main branch, create a temporary branch for implementing changes to the model files. Integration engineers can use Jenkins test results to decide whether to merge a temporary branch into the master branch. For more information, see

   https://git-scm.com/book/en/v1/Git-Branching-Branching-Workflows.
3. Under **Settings** > **Repository**, protect the master branch by enforcing the use of merge requests when developers want to merge their changes into the master branch.
4. Under **Settings**, on the **Integrations** page, add a webhook to the URL of your Jenkins project. This webhook triggers a build job on the Jenkins server.

Jenkins Setup

Install GitLab and Tap plugins. The MATLAB unit test uses the TAPPlugin to stream results to a `.tap` file. To enable communication of test status from MATLAB to the Jenkins job, Jenkins imports the `.tap` file.

Create a Jenkins project. Specify these configurations:

1. In your Jenkins project, click **Configure**.
2. On the **General** tab, specify a project name.
3. On the **Source Code Management** tab, for the **Repository URL** field, specify the URL of your GitLab repository.
4. On the **Build Triggers** tab, select **Build when a change is pushed to GitLab**.
5. On the **Build** tab, execute MATLAB to call the run script. The run script opens the project and runs all unit tests. For the project in this example, the code is:

   ```
   matlab -nodisplay -r...
    "cd /var/lib/jenkins/workspace/'18b Metrics CI Demo'; run(true)"
   ```
6. In the **Post-build Actions** tab, configure the TAP plugin to publish TAP results to Jenkins. In the **Test Results** field, specify `reports/*.tap`. For **Files to archive**, specify `reports/**,work/**`.

   The TAP plugin shows details from the MATLAB Unit test in the extended results of the job. The Jenkins archiving infrastructure saves derived artifacts that are generated during a Jenkins build.

Continuous Integration Workflow

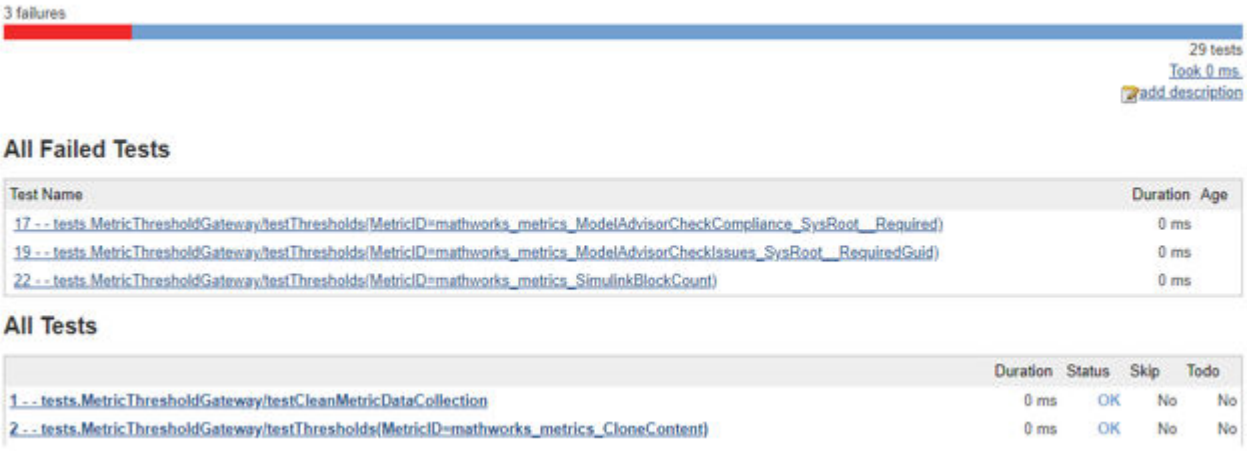
After setting up your project, Jenkins, and GitLab, follow the continuous integration workflow.

Phase 1: Feature Development

1. Create a local clone of the GitLab repository. See “Clone from Git Repository” (MATLAB).
2. In Simulink, navigate to the local GitLab repository.
3. Create a feature branch and fetch and check-out files. See “Branch and Merge Files with Git” (Simulink) and “Pull, Push, and Fetch Files with Git” (Simulink).
4. Make any necessary changes to the project files.
5. Simulate the model and validate the output in the Simulation Data Inspector.
6. Run MATLAB unit tests. For more information, see `runtests`.
7. Add and commit the modified models to the feature branch. See “Branch and Merge Files with Git” (Simulink) and “Pull, Push, and Fetch Files with Git” (Simulink).
8. Push changes to the GitLab repository. See “Branch and Merge Files with Git” (Simulink) and “Pull, Push, and Fetch Files with Git” (Simulink).
9. In GitLab, create a merge request. Select the feature branch as source branch and the target branch as master. Click **Compare Branches and Continue**.
10. If the feature is not fully implemented, mark the merge request as a work in progress by adding the letters WIP: at the beginning of the request. If the merge request is not marked WIP:, it immediately triggers a build after creation.
11. Click **Submit Merge Request**.

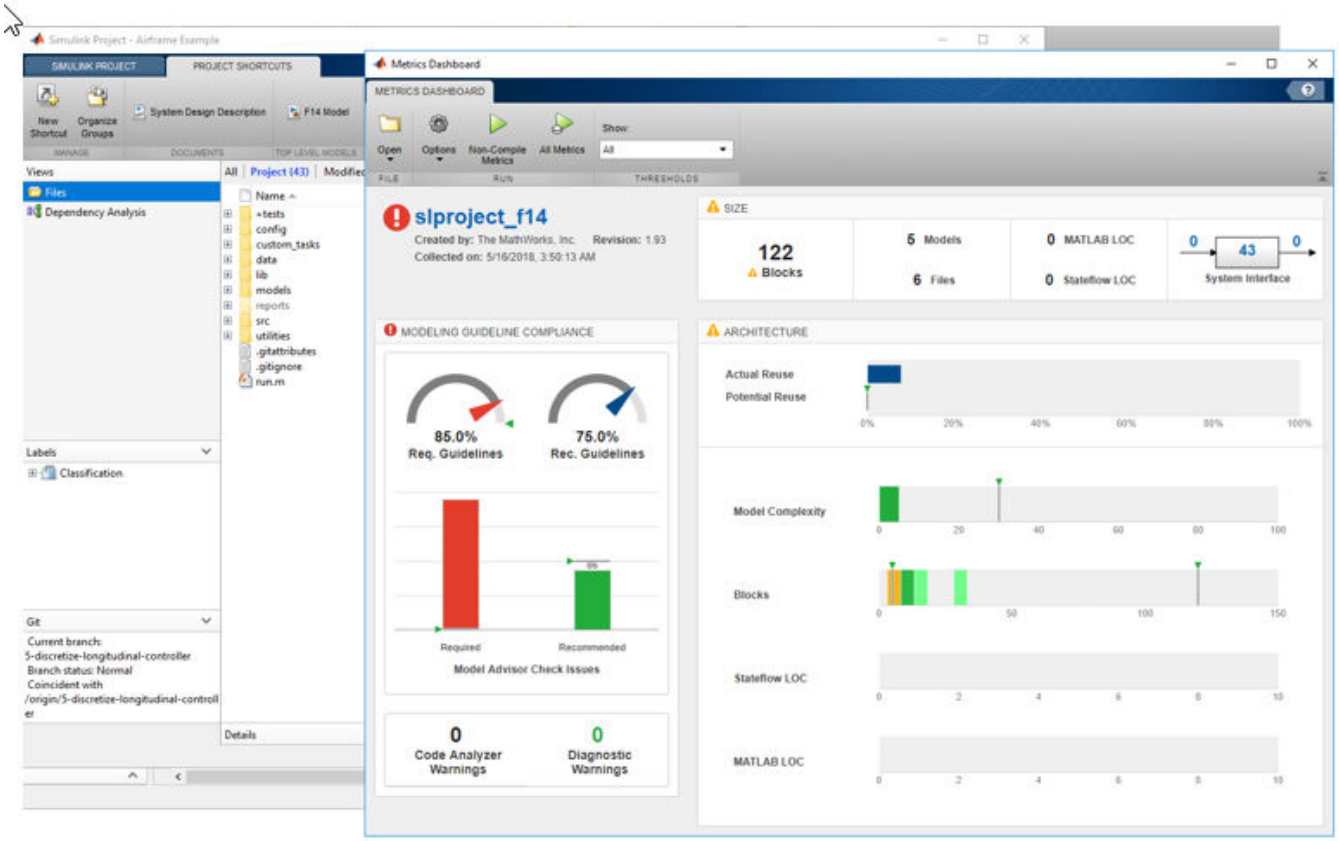
Phase 2: Qualification by Using Continuous Integration

1. If the letters WIP: are not at the beginning of the merge request, the push command triggers a Jenkins build. In the Jenkins Setup part of this example, you configured Jenkins to perform a build when you pushed changes to GitLab. To remove the letters, click **Resolve WIP status**.
2. Navigate to the Jenkins project. In Build History, you can see the build status.
3. Click the Build.
4. Click **Tap Test Results**.
5. For this example, the `MetricThresholdGateway.m` unit test did not pass for three metrics because these metrics did not meet the thresholds. To investigate this data, you must download the data locally.



Phase 3: Investigate Quality Issues Locally

1. Download the archived results to a local Git repository workspace.
2. Unzip the downloaded files. Copy the `reports/` and `work/` folders to the respective folders in the local repository.
3. To explore the results, open the project and the Metrics Dashboard.



4. To resolve the test failures, make the necessary updates to the models. Push the changes to the feature branch in GitLab.
5. Integration engineers can use Jenkins test results to decide when it is acceptable to perform the merge of the temporary branch into the master branch.

See Also

`slmetric.config.setActiveConfiguration` | `slmetric.dashboard.setActiveConfiguration`

More About

- “Collect Model Metric Data by Using the Metrics Dashboard” on page 1-8
- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2

Customize Metrics Dashboard Layout and Functionality

Customize the Metrics Dashboard by using the model metric programming interface. Customizing the dashboard extends your ability to use model metrics to assess that your model and code comply with size, complexity, and readability requirements. You can perform these Metrics Dashboard customizations:

- Configure compliance metrics to obtain compliance and issues metric data on your Model Advisor configuration.
- Customize the dashboard layout by adding custom metrics, removing widgets, and configuring existing widgets.
- Categorize metric data as compliant, warning, and noncompliant by specifying metric threshold values.

Configure Compliance Metrics

Use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration or on an existing check group such as the MISRA checks. To set up your own Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3. After you have set up your Model Advisor configuration, follow these steps to specify the check groups for which you want to obtain compliance and issues metric data:

1. To open the model, at the MATLAB command prompt, enter this command:

```
sf_car
```

2. Open the default configuration (that is, the one that is shipped with the Metrics Dashboard). Add a corresponding `slmetric.config.Configuration` object to the base workspace.

```
metricconfig=slmetric.config.Configuration.openDefaultConfiguration();
```

3. Create a cell array consisting of the Check Group IDs that correspond to those check groups. Obtain a Check Group ID by opening the Model Advisor Configuration Editor and selecting the folder that contains the group of checks. The folder contains a **Check Group ID** parameter.

```
values = {'maab', 'hisl_do178', '_SYSTEM_By Task_misra_c'};
```

This cell array specifies MAAB, High-Integrity, and MISRA check groups. The values `maab` and `hisl_do178` correspond to a subset of MAAB and High-Integrity System checks. To include all checks, specify the value for the **Check Group ID** parameter from the Model Advisor Configuration Editor.

4. To set the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method. The `'ModelAdvisorStandard'` string is a standard string that you must supply to the`setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(metricconfig,'ModelAdvisorStandard', values);
```

5. Open the default configuration for the Metrics Dashboard layout (that is, the one that ships with the Metrics Dashboard).

```
dashboardconfig = slmetric.dashboard.Configuration.openDefaultConfiguration();
```

6. Obtain the `slmetric.dashboard.Layout`object from the `slmetric.dashboard.Configuration` object.

```
layout = getDashboardLayout(dashboardconfig);
```

7. Obtain widget objects that are in the layout object.

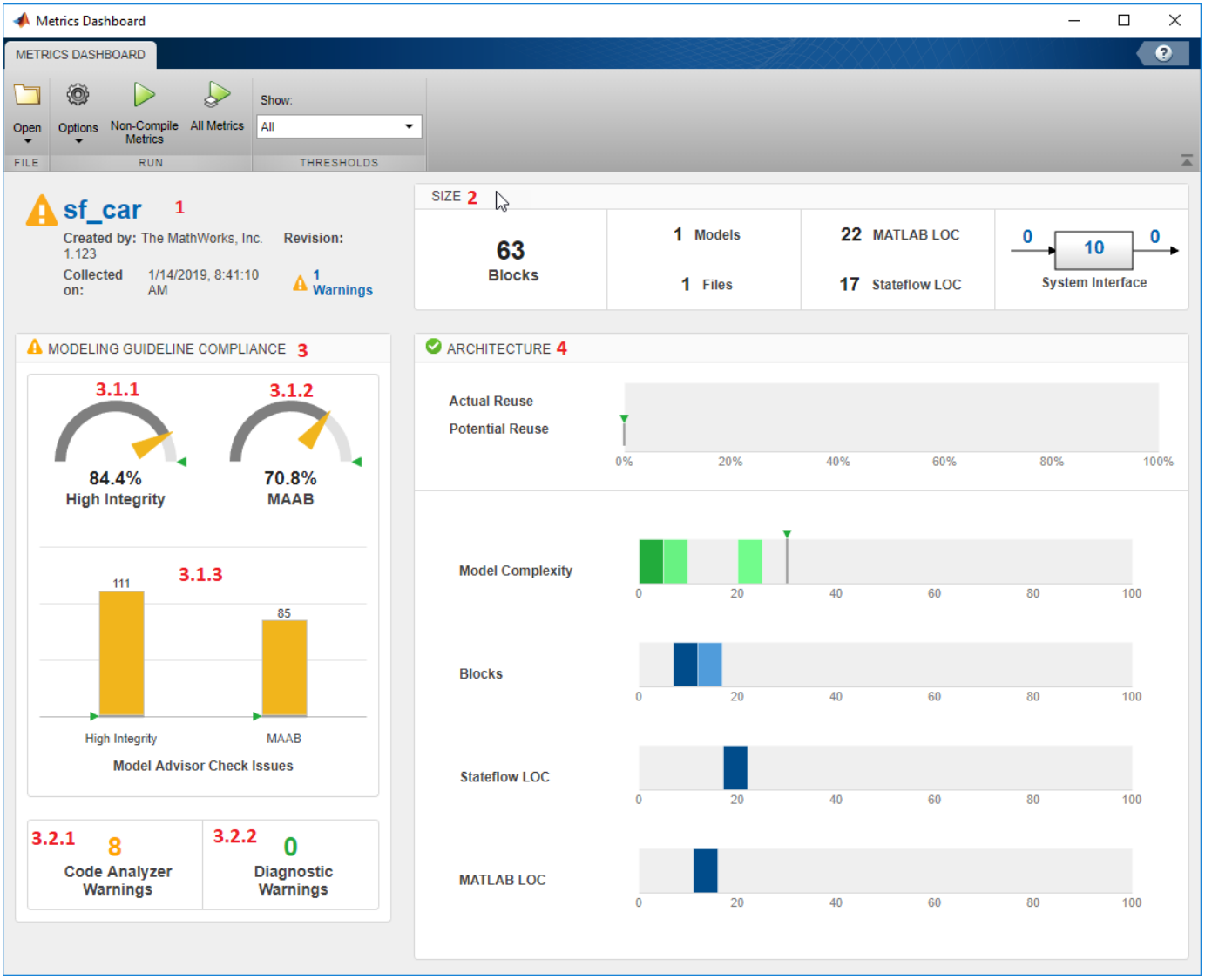
```
layoutWidget = getWidgets(layout);
```

8. The `slmetric.dashboard.Layout` object contains these objects:

- An `slmetric.dashboard.Container` object that holds an `slmetrics.dashboard.Widget` object of type `SystemInfo`. The red number one in the diagram below indicates the `SystemInfo` widget.
- An `slmetric.dashboard.Group`object that has the title **SIZE**.
- An `slmetrics.dashboard.Group` object that has the title **MODELING GUIDELINE COMPLIANCE**.
- An `slmetrics.dashboard.Group` object that has the title **ARCHITECTURE**.

In the diagram, the red numbers 1, 2, 3, and 4 indicate their order in the `layoutWidget` array. Obtain the compliance group from the layout.

```
complianceGroup = layoutWidget(3);
```



9. The modeling guideline compliance group contains two containers. The top container contains the **High Integrity** and **MAAB** compliance and check issues widgets. The red numbers 3.1.1, 3.1.2, and 3.1.3 indicate the order of the three widgets in the first container. The second container contains the **Code Analyzer Warnings** and **Diagnostic Warnings** widgets.

Remove the **High Integrity** compliance widget.

```
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
```

10. Create a custom widget for visualizing MISRA check issues metrics.

```
misraWidget = complianceContainers(1).addWidget('Custom', 1);
misraWidget.Title=('MISRA');
misraWidget.VisualizationType = 'RadialGauge';
misraWidget.setMetricIDs('mathworks.metrics.ModelAdvisorCheckCompliance._SYSTEM_By Task_misra_c'
misraWidget.setWidths(slmetric.dashboard.Width.Medium);
```

11. The bar chart widget visualizes the High Integrity and MAAB check groups. Point this widget to the MISRA and MAAB check groups.

```
setMetricIDs(complianceContainerWidgets(3),...
({'mathworks.metrics.ModelAdvisorCheckIssues._SYSTEM_By Task_misra_c',...
'mathworks.metrics.ModelAdvisorCheckIssues.maab'}));
complianceContainerWidgets(3).Labels = {'MISRA', 'MAAB'};
```

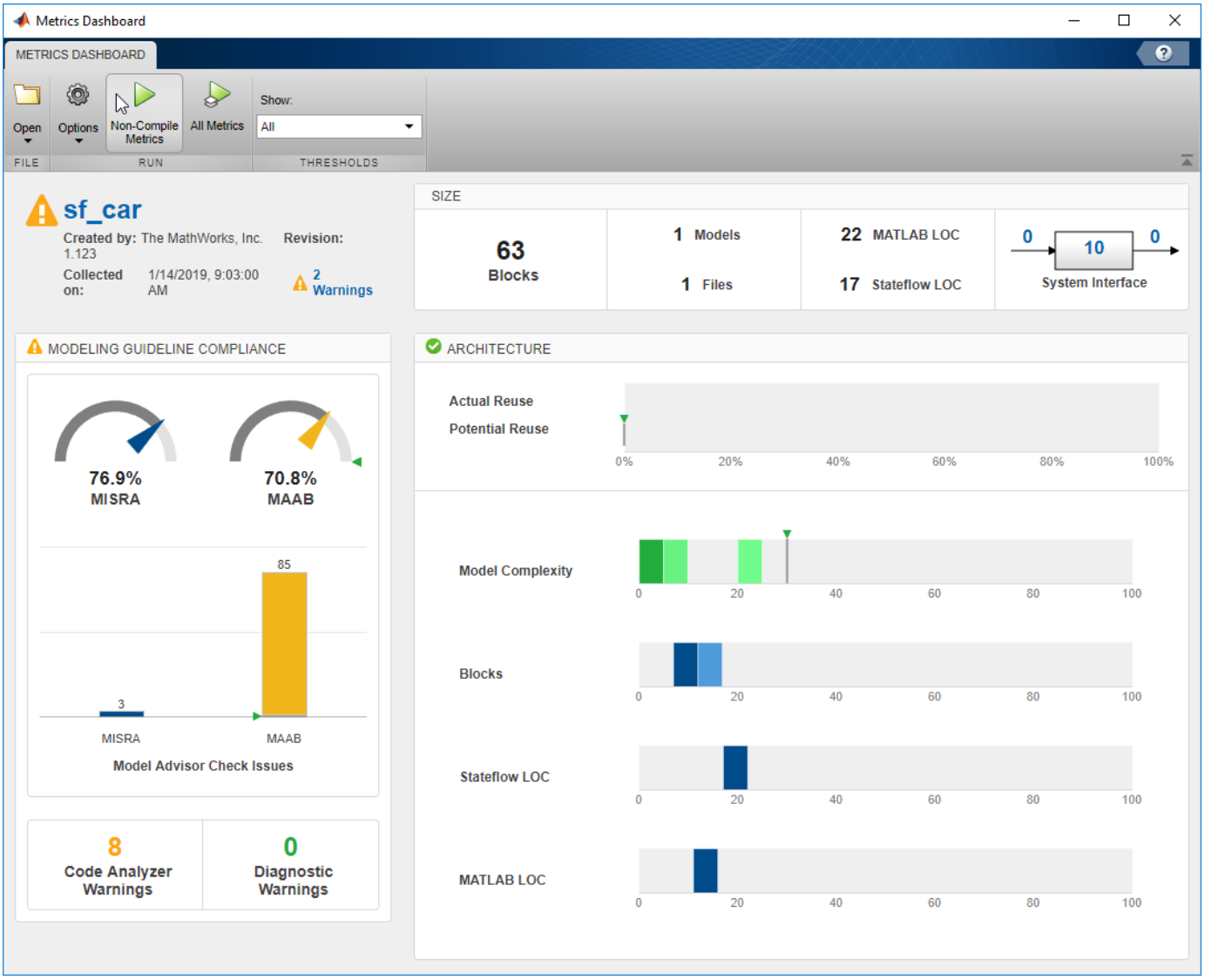
12. To run the Metrics Dashboard at this point in the example, uncomment out the following lines of code. The save commands serialize the API information to XML files. The `slmetric.config.setActiveConfiguration` and `slmetric.dashboard.setActiveConfiguration` commands set the active configuration objects.

```
% save(metricconfig,'FileName','MetricConfig.xml');
% save(dashboardconfig,'Filename','DashboardConfig.xml');
% slmetric.config.setActiveConfiguration(fullfile(pwd,'MetricConfig.xml'));
% slmetric.dashboard.setActiveConfiguration(fullfile(pwd,'DashboardConfig.xml'));
```

14. To open the Metrics Dashboard, uncomment this code.

```
% metricsdashboard sf_car
```

15. Click the **All Metrics** button and run all metrics. The Metrics Dashboard displays results for the MISRA checks instead of the High Integrity checks.



16. Close the Metrics Dashboard.

Add a Custom Metric to Dashboard

Create a custom metric that counts nonvirtual blocks. To display this metric on the Metrics Dashboard, specify a widget. Add it to the size group.

1. Using the `createNewMetricClass` function, create a new metric class named `nonvirtualblockcount`. The function creates a file, `nonvirtualblockcount.m`, in the current working folder. The file contains a constructor and empty metric algorithm method. For this example, make sure you are in a writable folder.

```
className = 'nonvirtualblockcount';
slmetric.metric.createNewMetricClass(className);
```

2. To write the metric algorithm, open the `nonvirtualblockcount.m` file and add the metric to the file. For this example, the file `nonvirtualblockcount_orig.m` contains the logic to create a metric that counts the nonvirtual blocks. Copy this file to the to `nonvirtualblockcount.m`.

```
copyfile nonvirtualblockcount_orig.m nonvirtualblockcount.m f
```

3. Register the new metric in the metric repository.

```
[id_metric,err_msg] = slmetric.metric.registerMetric(className);
```

4. Remove the widget that represents the Simulink block count metric. This widget is the first one in the size group. The size group is second in the `layoutWidget` array.

```
sizeGroup = layoutWidget(2);
sizeGroupWidgets = sizeGroup.getWidgets();
sizeGroup.removeWidget(sizeGroupWidgets(1));
```

5. Add a widget that displays the nonvirtual block count metric. For custom widgets, the default visualization type is single value. If you want to use a different visualization type, specify a different value for the `VisualizationType` property.

```
newWidget = sizeGroup.addWidget('Custom', 1);
newWidget.Title=('Nonvirtual Block Count');
newWidget.setMetricIDs('nonvirtualblockcount');
newWidget.setWidths(slmetric.dashboard.Width.Medium);
newWidget.setHeight(70);
```

6. Specify whether there are lines separating the custom widget from other widgets in the group. These commands specify that there is a line to the right of the widget.

```
s.top = false;
s.bottom = false;
s.left= false;
s.right= true;
newWidget.setSeparators([s, s, s, s]);
```

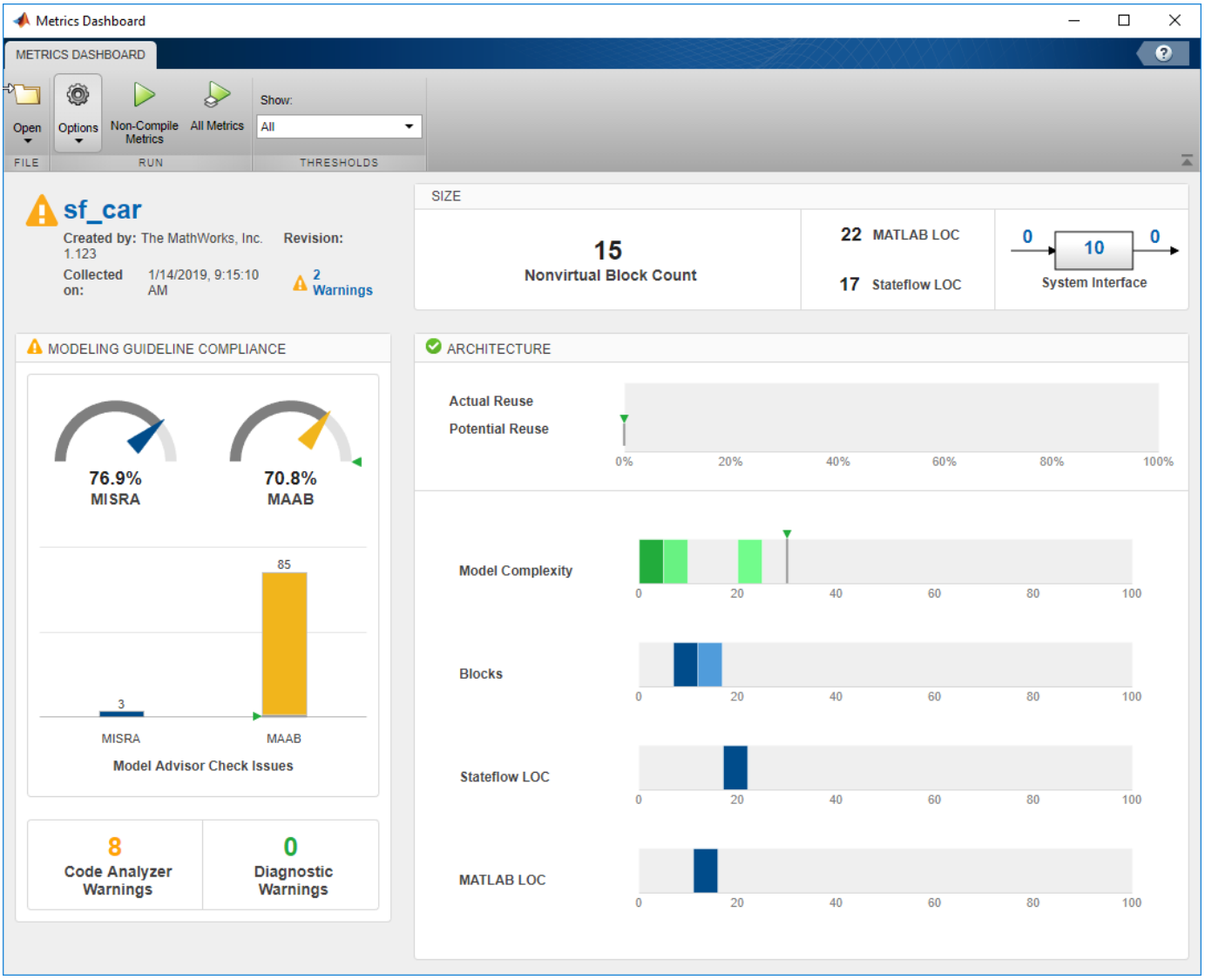
7. To run the Metrics Dashboard at this point in the example, uncomment out the following lines of code. The save commands serialize the API information to XML files. The `slmetric.config.setActiveConfiguration` and `slmetric.dashboard.setActiveConfiguration` commands set the active configuration objects.

```
% save(metricconfig,'FileName','MetricConfig.xml');
% save(dashboardconfig,'Filename','DashboardConfig.xml');
% slmetric.config.setActiveConfiguration(fullfile(pwd,'MetricConfig.xml'));
% slmetric.dashboard.setActiveConfiguration(fullfile(pwd,'DashboardConfig.xml'));
```

8. To open the Metrics Dashboard, uncomment this code.

```
% metricsdashboard sf_car
```

9. Click the **All Metrics** button and run all metrics. The Metrics Dashboard displays results for the nonvirtual block count metric instead of the Simulink block count metric.



10. Close the Metrics Dashboard.

Add Metric Thresholds

For the nonvirtual block count and MISRA metrics, specify metric threshold values. Specifying these values enables you to access the quality of your model by categorizing your metric data as follows:

- Compliant — Metric data that is in an acceptable range.
- Warning — Metric data that requires review.
- Noncompliant — Metric data that requires you to modify your model.

1. Access the `slmetric.config.ThresholdConfiguration` object in the `slmetric.config.Configuration` object `metricconfig`. Create the corresponding `slmetric.config.ThresholdConfiguration` object (`TC`) in the base workspace.

```
TC=getThresholdConfigurations(metricconfig);
```

2. Add two `slmetric.config.Threshold` objects to `TC`. Each `slmetric.config.Threshold` object contains a default `slmetric.config.Classification` object that is compliant. Specify the compliant metric ranges.

```
T1=addThreshold(TC,'mathworks.metrics.ModelAdvisorCheckIssues._SYSTEM_By Task_misra_c',...
 'AggregatedValue');
C=getClassifications(T1);
C.Range.Start=-inf;
C.Range.End=0;
C.Range.IncludeStart=0;
C.Range.IncludeEnd=1;

T2=addThreshold(TC,'mathworks.metrics.ModelAdvisorCheckCompliance._SYSTEM_By Task_misra_c',...
 'AggregatedValue');
C=getClassifications(T2);
C.Range.Start=1;
C.Range.End=inf;
C.Range.IncludeStart=1;
C.Range.IncludeEnd=0;
```

3. For each `slmetric.config.Threshold` object, specify the Warning ranges.

```
C=addClassification(T1,'Warning');
C.Range.Start=0;
C.Range.End=inf;
C.Range.IncludeStart=0;
C.Range.IncludeEnd=1;

C=addClassification(T2,'Warning');
C.Range.Start=-inf;
C.Range.End=1;
C.Range.IncludeStart=0;
C.Range.IncludeEnd=0;
```

These commands specify that if the MISRA checks have issues, the model status is warning. If there are no issues, the model status is compliant.

4. Add a third `slmetric.config.Threshold` object to `TC`. Specify compliant, warning, and noncompliant ranges for this `slmetric.config.Threshold` object.

```
T3=addThreshold(TC,'nonvirtualblockcount', 'AggregatedValue');
C=getClassifications(T3);
C.Range.Start=-inf;
C.Range.End=20;
C.Range.IncludeStart=1;
C.Range.IncludeEnd=1;

C=addClassification(T3, 'Warning');
C.Range.Start=20;
C.Range.End=30;
C.Range.IncludeStart=0;
C.Range.IncludeEnd=1;

C=addClassification(T3, 'NonCompliant');
C.Range.Start=30;
C.Range.End=inf;
C.Range.IncludeStart=0;
C.Range.IncludeEnd=1;
```

These commands specify that the compliant range is less than or equal to 20. The warning range is from 20 up to but not including 30. The noncompliant range is greater than 30.

5. Save the configuration objects. These commands serialize the API information to XML files.

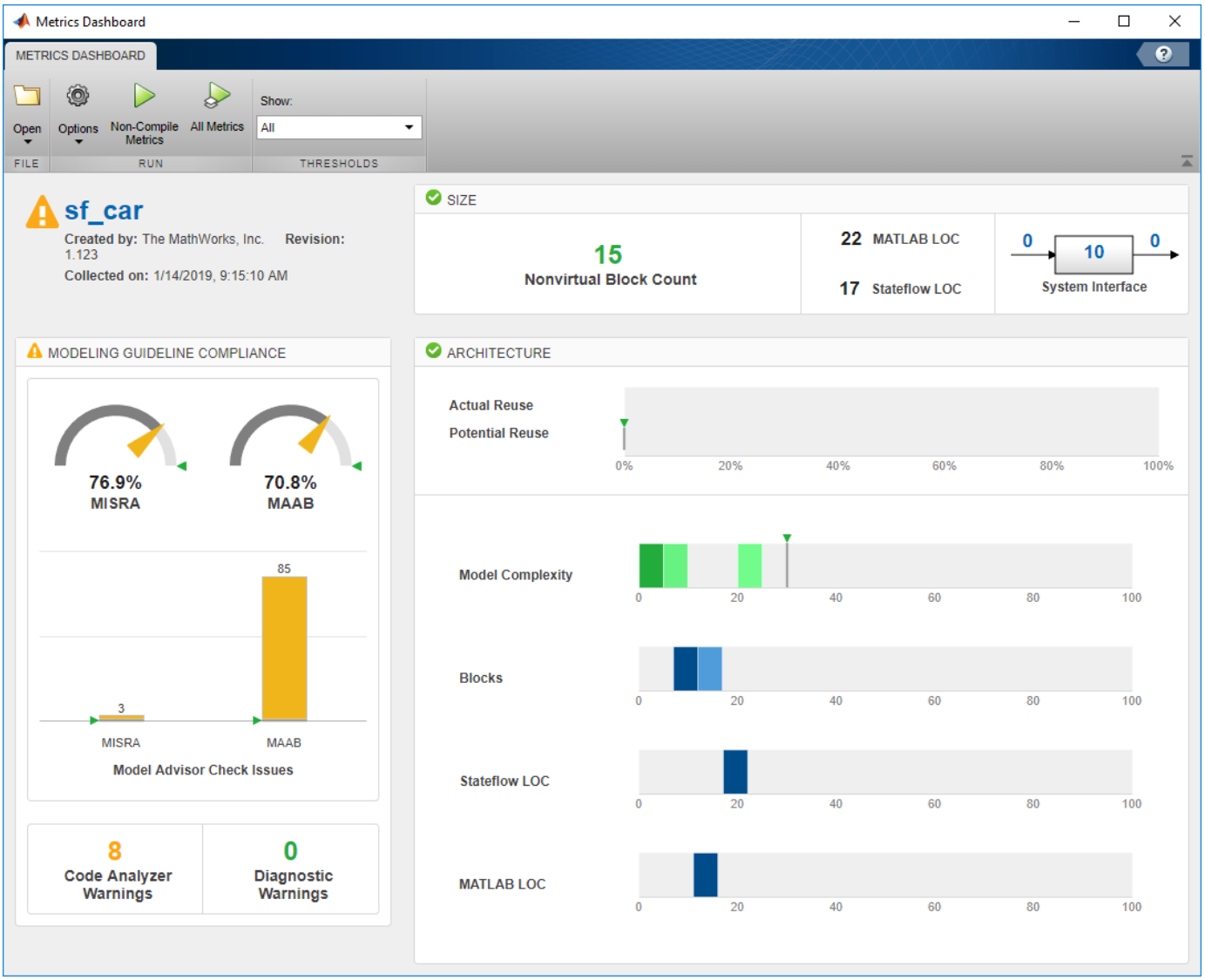
```
save(metricconfig,'FileName','MetricConfig.xml');
save(dashboardconfig,'Filename','DashboardConfig.xml');
```

6. Set the active configurations.

```
slmetric.config.setActiveConfiguration(fullfile(pwd, 'MetricConfig.xml'));
slmetric.dashboard.setActiveConfiguration(fullfile(pwd, 'DashboardConfig.xml'));
```

7. For your model, open the Metrics Dashboard.

```
metricsdashboard sf_car
```



For the MISRA check compliance issues, the gauge is yellow because 76% of the checks pass. Any percentage less than 100% is a warning. The bar chart also displays a yellow because the model contains three MISRA check issues. Any number greater than zero is a warning.

The **Nonvirtual Block Count** widget is in the compliant range because there are 15 nonvirtual blocks.

8. To reset the configuration and unregister the metric, uncomment and execute these commands:

```
% slmetric.metric.unregisterMetric(className);
% slmetric.dashboard.setActiveConfiguration('');
% slmetric.config.setActiveConfiguration('');
```

See Also

`slmetric.dashboard.Configuration` | `slmetric.config.Configuration`

More About

- “Collect Model Metrics”
- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2

Compare Model Complexity and Code Complexity Metrics

Analyze the complexity of your system by using the cyclomatic complexity metrics. The metrics indicate the structural complexity of a system by measuring the number of linearly independent paths in the system. By limiting the cyclomatic complexity of your system, you can make it more readable, maintainable, and portable. You can measure the cyclomatic complexity for both your model and the code generated from your model. Note that differences between the code and the model may result in different levels of cyclomatic complexity. To measure the cyclomatic complexity of a model, use the Metrics Dashboard and the “Cyclomatic complexity metric”.

Metric Threshold Values

Code Complexity Threshold

When you develop an algorithm by hand-writing code, you assess the readability of the code by measuring the cyclomatic complexity of the code. Code that has higher cyclomatic complexity can be more difficult to understand and maintain. To standardize code maintainability, your organization may select a threshold value that limits the cyclomatic complexity of your code. For example, if you write code that conforms to the “HIS Code Complexity Metrics” (Polyspace Bug Finder), you check that the cyclomatic complexity of the code is at or below the threshold of 10.

Model Complexity Threshold

When you use the model-based design workflow to model an algorithm and generate code, you can assess the readability of the system by using the cyclomatic complexity metric of the model instead of measuring the cyclomatic complexity of the generated code. The graphical modeling of Simulink allows you to manage complex algorithms better than traditional hand code does. To account for this, the default cyclomatic complexity metric threshold for the model is 30, which is higher than the standard code complexity threshold of 10. To change the model metric threshold value, see “Customize Metrics Dashboard Layout and Functionality” on page 5-39.

Comparing Code and Model Complexity Metric Results

The cyclomatic complexity of a model can be higher or lower than the cyclomatic complexity of the generated code. This variation depends on your model and on your code generation customizations. Some of the patterns that generate different complexity measurements include:

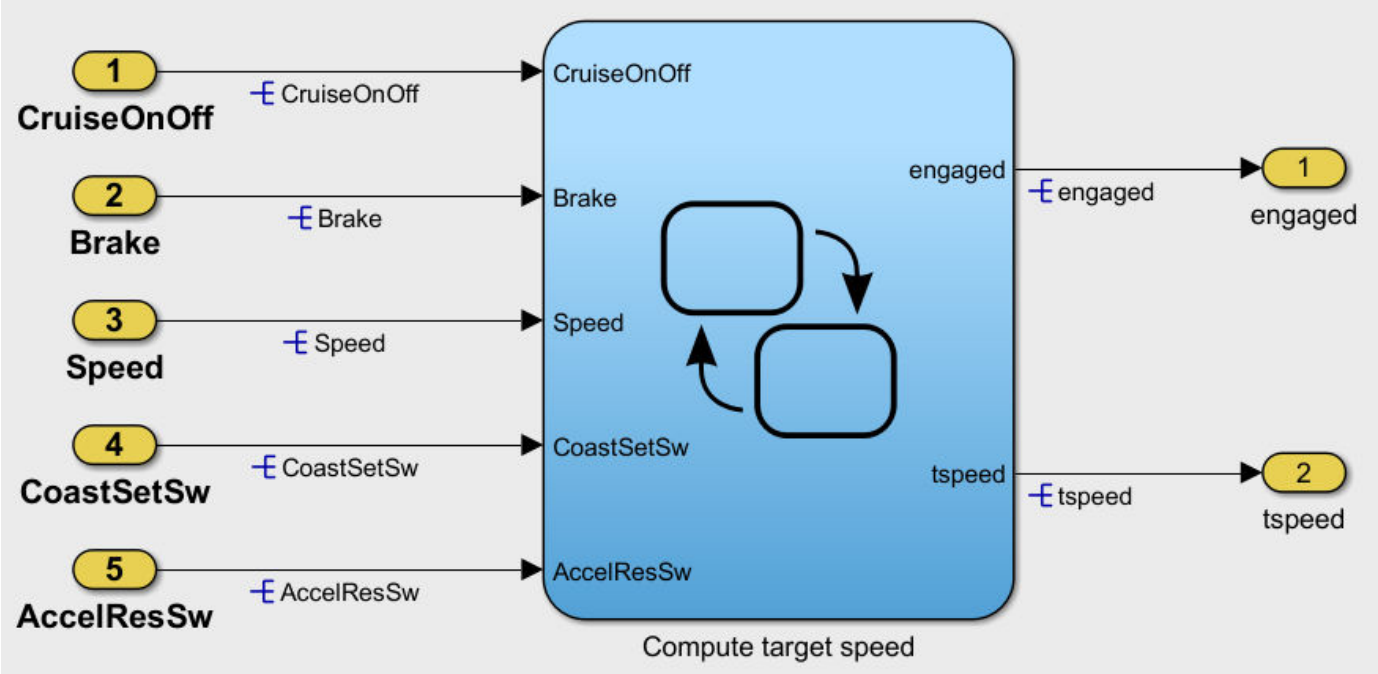
- Code generation optimizations that eliminate extra logic that the model contains. These optimizations can reduce the complexity of the code.
- Error checks in the generated code that the model metric analysis does not consider. These error checks can increase the complexity of the code.
- Additional logic in the generated code for a specific target. This logic can increase the complexity of the code.

For example, consider the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

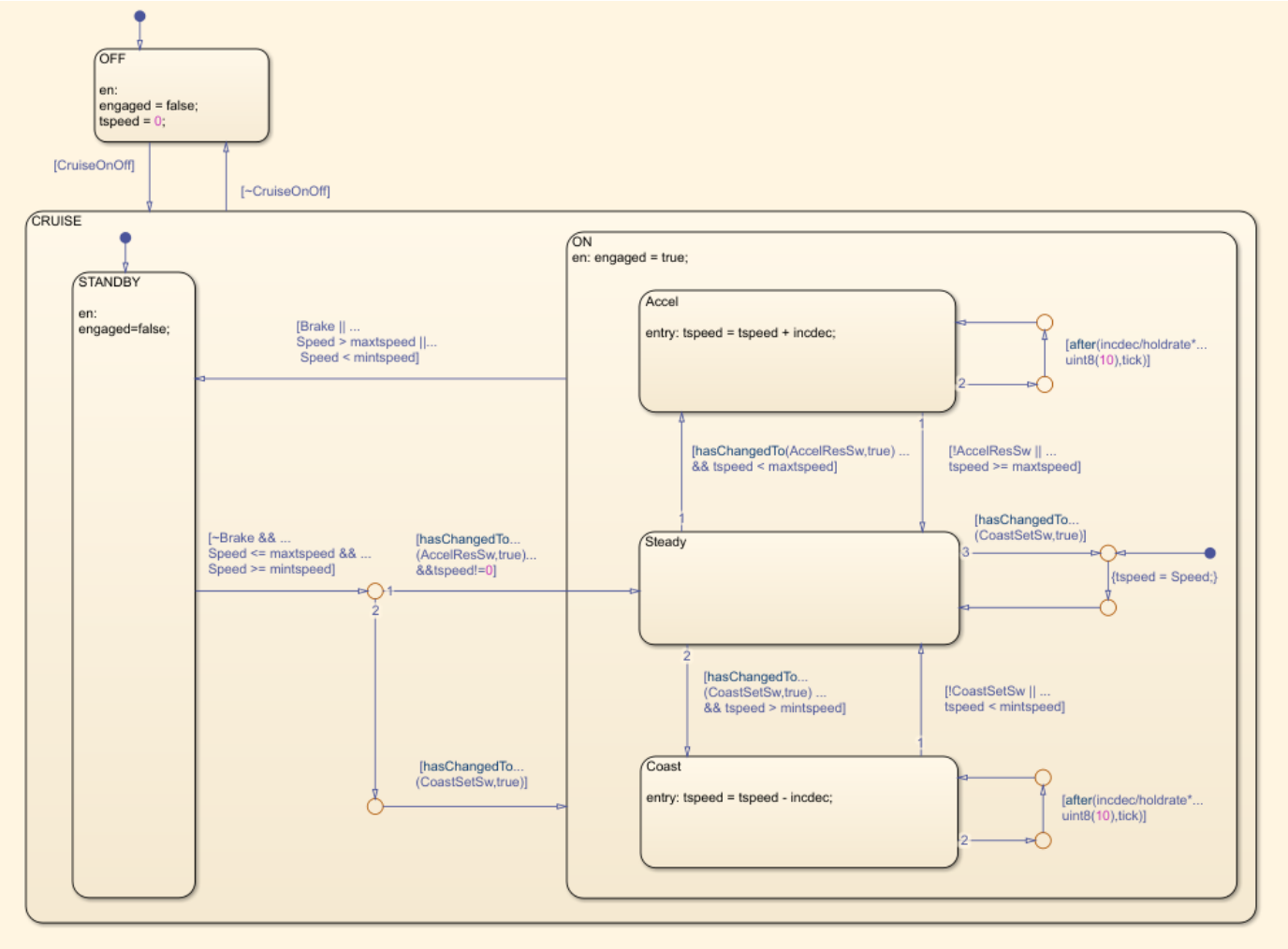
1. Open the project.

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise');
run(fullfile(path,'slVerificationCruiseStart'))
```

2 From the project, open the model folder and open `simulinkCruiseErrorAndStandardsExample`.



The model contains the chart Compute target speed. To generate and analyze code for the chart by using Polyspace, see “Analyze Code and Test Software-in-the-Loop” on page 2-12.



The reports that Polyspace generates for the code include code metrics such as cyclomatic complexity. The generated step function for the chart has a cyclomatic complexity of 20.

To measure cyclomatic complexity of the model, use the Metrics Dashboard:

1. Open the Metrics Dashboard. In the Apps gallery, click **Metrics Dashboard**.
2. Click **All Metrics**.
3. To view detailed cyclomatic complexity results, click the **Model Complexity** widget.

The chart in the model has a cyclomatic complexity of 30. For this chart, the code generator optimizes the code by consolidating logic, so the generated code has a lower cyclomatic complexity than the chart in the model. In other cases, a model may have lower cyclomatic complexity than its generated code. When you maintain the model for code generation, use the cyclomatic complexity of the model to measure your system's complexity.

See Also

“Cyclomatic complexity metric”

More About

- “Analyze Code and Test Software-in-the-Loop” on page 2-12

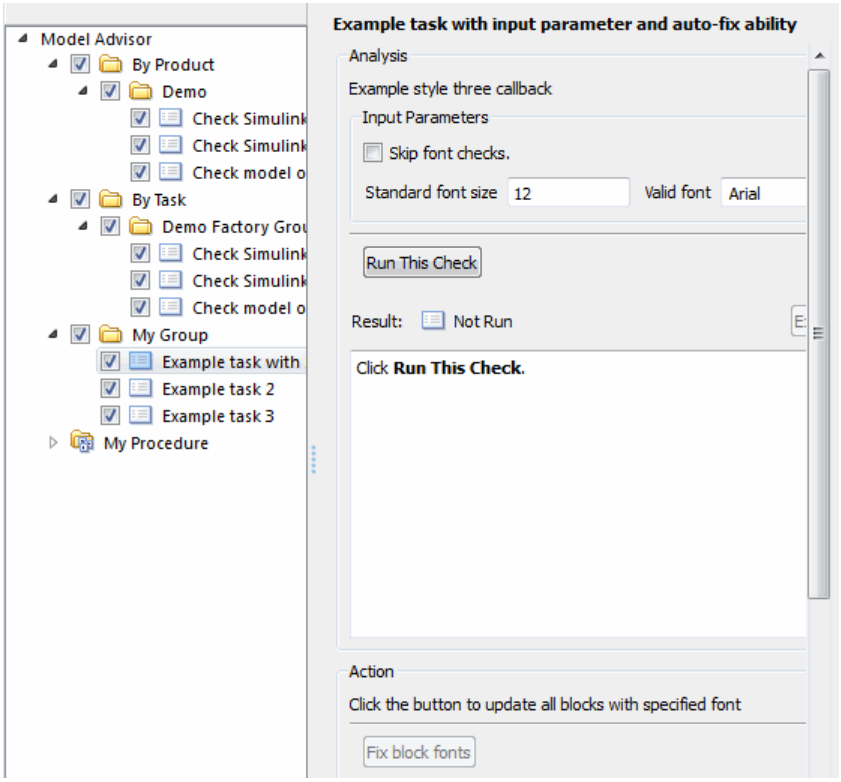
Create Model Advisor Checks

Overview of the Customization File for Custom Checks

A customization file is a MATLAB file that you create and name `sl_customization.m`. The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

Function	Description	When Required
`sl_customization()`	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at start-up. See “Defining Custom Model Advisor Checks Workflow” on page 6-34.	Required for programmatic customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Defining Custom Model Advisor Checks Workflow” on page 6-34.	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 “Defining Custom Model Advisor Checks Workflow” on page 6-34.	Required for custom checks. You must write one callback function for each custom check
One or more calls to check input parameters	Specifies input parameters to custom checks. See “Define Check Input Parameters” on page 6-36.	Optional
One or more calls to checklist views	Specifies calls to the Model Advisor Result Explorer for custom checks.	Optional
One or more calls to check actions	Specifies actions the software performs for custom checks. See “Defining Custom Model Advisor Checks Workflow” on page 6-34.	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.


Model Advisor
By Product
Demo
Check Simulink
Check Simulink
Check model o
By Task
Demo Factory Grou
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
12
Valid font
Arial
Run This Check
Result:
Not Run
Click Run This Check.
Action
Click the button to update all blocks with specified font
Fix block fonts

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: • Blocks • Lines • Annotations When getting the object, you can: • Specify a search depth • Search under masks and libraries
`get_param` / `set_param`	Getting and setting system and block parameter values.
Property Inspector	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” (Simulink).
Stateflow API (Stateflow)	Programmatic access to Stateflow objects.

Create Pass/Fail and Informational Model Advisor Checks

This example demonstrates how to create two simple check types: a pass/fail check with no fix action and an informational check. A basic pass/fail check finds and reports what the check is reviewing and whether the check passes or fails. An informational check finds and displays a description of what the check is reviewing and any references to applicable standards.

Create an sl_customization Function

In your working folder, create the `sl_customization.m` file. To register the custom checks, within the `sl_customization.m` file, create an `sl_customization(cm)` function as shown here. This function accepts one argument, a customization object. This customization manager object includes the `addModelAdvisorCheckFcn` method for registering the custom checks. The input to this method is a handle to the function (`defineModelAdvisorChecks`) that contains calls to two check definition functions. These functions contain the definitions of the simple pass/fail check and the informational check.

```
function sl_customization(cm)
% SL_CUSTOMIZATION - Model Advisor customization demonstration.

% Copyright 2019 The MathWorks, Inc.

% register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorCheck);

% ------------------------------
% defines Model Advisor Checks
% ------------------------------
function defineModelAdvisorCheck
definePassFailCheck
defineInformationCheck
```

Create the Check Definition Function for a Pass/Fail Check with No Fix Action

In this section, you create the check definition function that checks whether a Constant block value is a number or a letter. If the value is a number, the check produces a warning. If the value is a letter, the check passes.

This check uses the `DetailStyle` type of callback function. This style allows you to view results by block, subsystem, or recommended action. Applying this style produces default formatting, so that you do not have to use the `ModelAdvisor.FormatTemplate` class or the other Model Advisor formatting APIs to format the results that appear in the Model Advisor. You specify this style as an input to the `setCallbackFcn` method.

Create a new file, `definePassFailCheck.m`, and enter the function shown here:

```
function definePassFailCheck
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('simplePassFailCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Warn if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
rec.setCallbackFcn(@simplePassFailCheck,'None','DetailStyle')
```

```
mdladvRoot.publish(rec, 'Demo');

% --- Callback function that checks Constant blocks
function simplePassFailCheck(system,CheckObj)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
violationBlks=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify Constant blocks with a value that is a number.';
    ElementResults.Status = 'All Constant blocks have have a value that is a letter.';
    mdladvObj.setCheckResultStatus(true);
else
    ElementResults(1,numel(violationBlks))=ModelAdvisor.ResultDetail;
    for i=1:numel(ElementResults)
        ElementResults(i).setData(violationBlks{i});
        ElementResults(i).Description = 'Identify Constant blocks with a value that is a number.';
        ElementResults(i).Status = 'The following Constant blocks have values that are numbers:';
        ElementResults(i).RecAction =  'Change the Constant block value to a letter.';
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
```

This check identifies Constant block values that are numbers and produces a warning, but it does not provide a fix action. For more information on how to create a check definition function with a fix, see “Create a Pass/Fail Model Advisor Check with Fix Action” on page 6-9.

Create the Check Definition Function for an Informational Check

In this section, you create a check definition function for an informational check that finds and displays the model configuration and checksum information.

For an informational check, the Model Advisor displays the overall check status, but the status is not in the result. In addition, an informational check does not include the following items in the results:

- A description of the status.
- The recommended action to take when the check does not pass.
- Subcheck results.

This check definition function uses the `StyleOne` type of callback function, so you must use the Model Advisor formatting APIs to format results that appear in the Model Advisor. You specify `StyleOne` as an input to the `setCallbackFcn` method.

Create a new file, `defineInformationCheck.m`, and enter the function shown here:

```
function defineInformationCheck

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.infocheck');
rec.Title = 'Identify model configuration and checksum information';
rec.TitleTips = 'Display model configuration and checksum information';
rec.setCallbackFcn(@modelVersionChecksumCallbackUsingFT,'None','StyleOne');

% Publish check into Demo group.
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec, 'Demo');
```

```
end

% ----------------------------
% This callback function uses the StyleOne CallbackStyle type.
% ----------------------------

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');

% Description of check in results
setInformation(ft, 'Display model configuration and checksum information');

% Add See Also section for references to standards.
docLinkSfunction{1}     = {['IEC 61508-3, Table A.8 (5)' ...
                            ' ''Software configuration management'' ']};
setRefLink(ft,docLinkSfunction);

% If running the Model Advisor on a subsystem, add note to description.
if strcmp(system, model) == false
    setInformation(ft,{['NOTE: The Model Advisor is reviewing a' ...
        ' sub-system, but these results are based on root-level settings.']});
end

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);

% If error is encountered, 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

lbStr ='<br/>';
resultStr = ['Model Version: ' mdlver lbStr 'Author: ' mdlauthor lbStr ...
    'Date: ' mdldate lbStr 'Model Checksum: ' mdlsum];

% Display the results
setSubResultStatusText(ft,resultStr);
```

```
% The check does not have subresults and does not need a bar.
setSubBar(ft,0);
resultDescription{end+1} = ft;
end
```

Run the Custom Checks in the Model Advisor

1. In the Command Window, enter:

   ```
   Advisor.Manager.refresh_customizations
   ```

2. Open the model `sldemo_fuelsys` by typing this command in the MATLAB command prompt:

   ```
   sldemo_fuelsys
   ```

3. In the **Modeling** tab, select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Click **OK**.
4. In the left pane, select **By Product > Demo > Identify model configuration and checksum information**.
5. Click **Run This Check**.

   The check passes and displays the information.
6. In the left pane, select **By Product > Demo > Check Constant block usage**.
7. Click **Run This Check**.

   The check produces a warning because several blocks contain values that are numbers. The results contain links to these blocks. The result displays a **Recommended Action**.
8. Follow the **Recommended Action** to fix the Constant blocks.

See Also

`ModelAdvisor.FormatTemplate` | `ModelAdvisor.Check` | ModelAdvisor.Check.CallbackContext | `ModelAdvisor.FormatTemplate`

More About

- “Defining Custom Model Advisor Checks Workflow” on page 6-34
- “Create a Pass/Fail Model Advisor Check with Fix Action” on page 6-9
- “Create and Deploy a Model Advisor Custom Configuration” on page 7-25

Create a Pass/Fail Model Advisor Check with Fix Action

This example shows how to create a customized Model Advisor pass/fail check with a fix action. When a model does not contain a check violation, the results contain the check description and result status. When a model contains a check violation, the results contain the check description, result status, and the recommended action to fix the issue. This example adds a custom check to a Model Advisor **By Product > Demo** subfolder.

For this example, the custom check identifies blocks whose names do not appear below the blocks. The fix action is to make the block names appear below the blocks.

This example also shows you how to collect results into groups that violate a check (that is, detailed result collections), such as blocks in a subsystem. In the Model Advisor, you can review results by selecting:

- **View By > Recommended Action** − When a check is violated, this view shows a list of model elements that violate the check. When there is no violation, this view provides a brief description stating that the check was not violated.
- **View By > Subsystem** − This view shows a table of model elements that violate the check, organized by model or subsystem (when applicable).
- **View By > Block** − This view provides a list of check violations for each block.

When a check does not pass, the results include a hyperlink to each model element that violates the check. Use these hyperlinks to easily locate areas in your model or subsystem. The code for this example consists of an `sl_customization.m` file and a `defineDetailStyleCheck.m` file.

Create the sl_customization File

1. In your working folder, create an `sl_customization.m` file.
2. To register the custom checks, create an `sl_customization(cm)` function as shown here. This function accepts one argument, a customization manager object. The customization manager object includes the `addModelAdvisorCheckFcn` method for registering the custom check. The input to this method is a handle to the check definition function.

```
function sl_customization(cm)
% SL_CUSTOMIZATION - Model Advisor customization demonstration.

% Copyright 2019 The MathWorks, Inc.

% register custom checks
cm.addModelAdvisorCheckFcn(@defineDetailStyleCheck);

% ----------------------------
% defines Model Advisor Checks
% ----------------------------
function defineDetailStyleCheck;
```

Create the Check Definition File

The check definition function defines the check and fix actions that the Model Advisor takes when you run the check. For this example, the completed check definition function file is `defineDetailStyleCheck.m`, and it contains this code:

```
function defineDetailStyleCheck

mdladvRoot = ModelAdvisor.Root;

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
% Create ModelAdvisor.Action object for setting fix operation.
myAction = ModelAdvisor.Action;
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
rec.setAction(myAction);
myAction.setCallbackFcn(@ActionCB);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

end

% -----------------------------
% This callback function uses the DetailStyle CallbackStyle type.
% -----------------------------
function DetailStyleCallback(system, CheckObj)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

% Find all blocks whose name does not appear below blocks
violationBlks = find_system(system, 'Type','block',...
    'NamePlacement','alternate',...
    'ShowName', 'on');
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block.';
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
else
    ElementResults(1,numel(violationBlks))=ModelAdvisor.ResultDetail;
    for i=1:numel(ElementResults)
        ElementResults(i).setData(violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below the block.';
        ElementResults(i).Status = 'The following blocks have names that do not display below the blocks:';
        ElementResults(i).RecAction =  'Change the location such that the block name is below the block.';
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end

% -----------------------------
% This action callback function changes the location of block names.
% -----------------------------
function result = ActionCB(taskobj)
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
resultDetailObjs = checkObj.ResultDetails;
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block,'NamePlacement','normal');
end

result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
mdladvObj.setActionEnable(false);
end
```

The following steps explain how to create completed the `defineDetailStyleCheck.m` file.

1 Create a `ModelAdvisor.Root` object.

```
mdladvRoot = ModelAdvisor.Root;
```

2. Create a `ModelAdvisor.Check` object and define the unique check ID. For this check, the ID is `com.mathworks.sample.detailStyle`.

```
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
```

3. Specify the `ModelAdvisor.Check.Title` and `ModelAdvisor.Check.TitleTips` properties.

```
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
```

4. Use the `setCallbackFcn` method to call the callback function. The `setCallbackFcn` method arguments are a handle to the callback function and the `ModelAdvisor.Check.CallbackStyle` property value. For this example, the `CallbackStyle` property value is `DetailStyle`. This style allows you to view results by block, subsystem, or recommended action. Applying this style produces default formatting, so that you do not have to use the `ModelAdvisor.FormatTemplate` class or the other Model Advisor formatting APIs to format the results that appear in the Model Advisor.

```
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
```

5. To set the fix operation, create a `ModelAdvisor.Action` object and define its properties.

```
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
```

6. Use the `setCallback` method to call the action callback function. The input to this method is a handle to the action callback function.

```
myAction.setCallbackFcn(@ActionCB);
```

7. Use the `setAction` method to set the action for the check.

```
rec.setAction(myAction);
```

8. Use the `publish` method to publish the check to a folder within the **By Product** folder. For this example, the folder name is **Demo**.

```
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.
```

Create the Check Callback Definition Function

1. In the `defineDetailStyleCheck.m` file, create the check callback function. In this example, the function name is `DetailStyleCallback`. The inputs to this function are a `ModelAdvisor.CheckObject` and the path to the model or system that the Model Advisor analyzes.

```
function DetailStyleCallback(system, CheckObj)
```

2. To create a `Simulink.ModelAdvisor` object, use the `Simulink.ModelAdvisor.getModelAdvisor` method.

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

3. To identify blocks that violate the check, use the `find_system` function. For each model element, this function creates a `ModelAdvisor.ResultDetail` object.

```
violationBlks = find_system(system, 'Type','block',...
                            'NamePlacement','alternate',...
                            'ShowName', 'on');
```

4. Write code for the case when the `find_system` function does not identify blocks whose names do not appear below the block. In this case, `ElementResults` is one instance of a `ModelAdvisor.ResultDetail` object and provides information content only. The method specifies that there is no check violation and displays **Passed** in the Model Advisor.

```
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block.';
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
```

5 Write code for the case when the `find_system` function returns a list of blocks whose names do not appear below the block (`violationBlks`). `ElementResults` includes each `ModelAdvisor.ResultDetail` object that violates the check and provides a recommended action message for fixing the check violation.

For this case, the `setCheckResultStatus` method specifies the check violation and displays **Warning** or **Failed** in the Model Advisor. The `Simulink.ModelAdvisor.setActionEnable(true)` method enables the ability to fix the check violation issue from the Model Advisor.

```
else
    ElementResults(1,numel(violationBlks))=ModelAdvisor.ResultDetail;
    for i=1:numel(ElementResults)
        ElementResults(i).setData(violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below the block.';
        ElementResults(i).Status = 'The following blocks have names that do not display below the blocks:';
        ElementResults(i).RecAction =  'Change the location such that the block name is below the block.';
    end
mdladvObj.setCheckResultStatus(false);
mdladvObj.setActionEnable(true);
end
```

6 To associate the results with a check object, use the `setResultDetails` method.

```
CheckObj.setResultDetails(ElementResults);
end
```

Create the Action Callback Definition Function

1 In the `defineDetailStyleCheck.m` file, create the action callback function. In this example, the function name is `sampleActionCB`. The input to this function is a `ModelAdvisor.Task` object.

```
function result = ActionCB(taskobj)
```

2 Create handles to `Simulink.ModelAdvisor` and `ModelAdvisor.Check` objects.

```
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
```

3 Create an array of `ModelAdvisor.ResultDetail` objects for storing the information for blocks that violate the check.

```
resultDetailObjs = checkObj.ResultDetails;
```

4 Write code that changes the block name location to below the block.

```
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block,'NamePlacement','normal');
end
result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
```

5 Disable the **Action** box.

```
mdladvObj.setActionEnable(false);
```

Run the Check

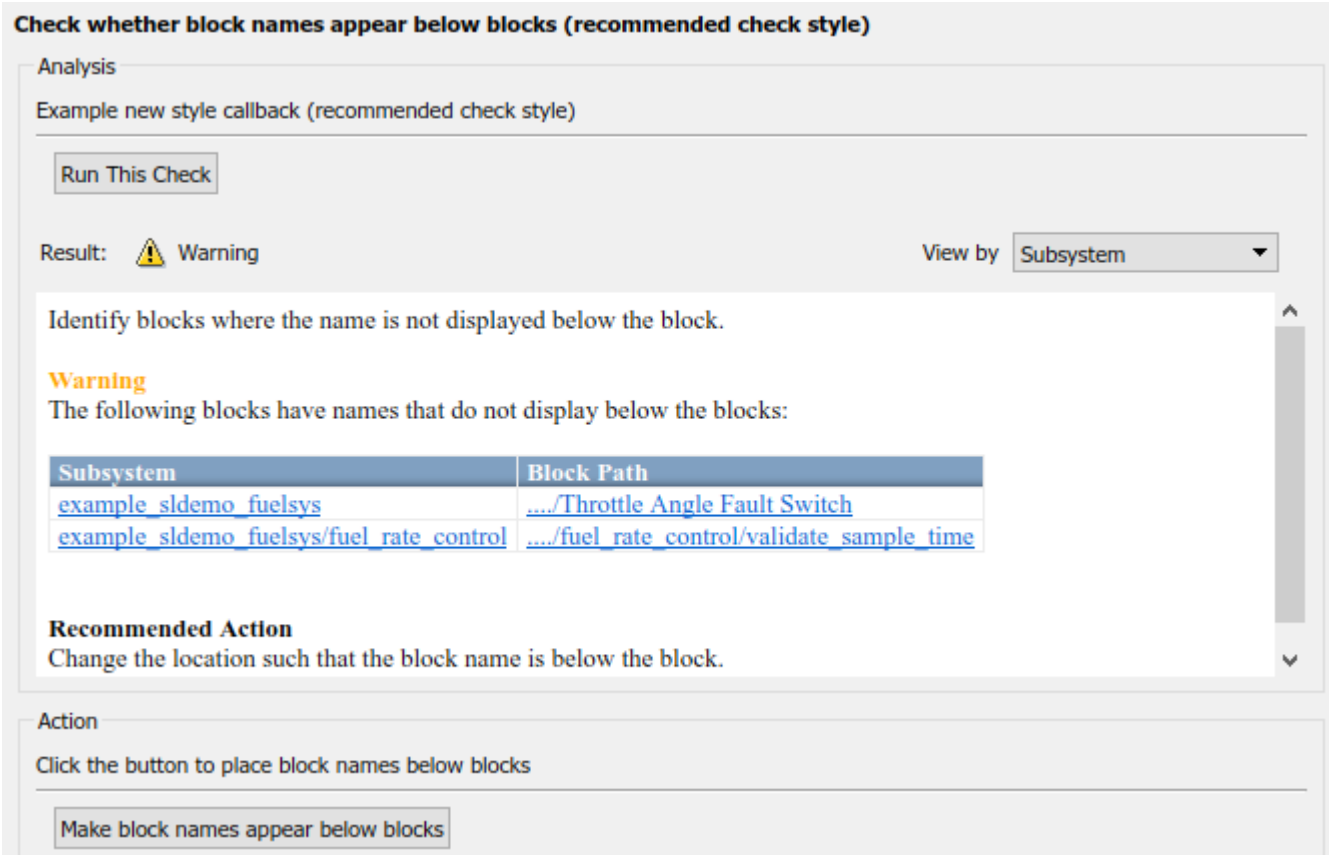
1. Save the `sl_customization.m` and `defineDetailStyleCheck.m` files.
2. In the MATLAB command window, enter:

   ```
   Advisor.Manager.refresh_customizations
   ```
3. From the MATLAB window, open the `sldemo_fuelsys` model.
4. In the top model, right-click the `Engine Speed` block and select **Rotate & Flip** > **Flip Block Name**.
5. Open the `fuel_rate_control` subsystem. Right-click the `validate_sample_time` block and select **Rotate & Flip** > **Flip Block Name**.

   Return to the top model and save as `example_sldemo_fuelsys`.
6. In the **Modeling** tab, select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Click **OK**. The Model Advisor opens.
7. In the left pane, select **By Product > Demo > Check whether block names appear below blocks**.

   Note If the **By Product** folder is not displayed in the Model Advisor window, select **Settings** > **Preferences** > **Show By Product Folder**
8. Select **Run This Check**. The Model Advisor check fails for the blocks you changed.
9. Review the results by selecting one of the **View by** options.

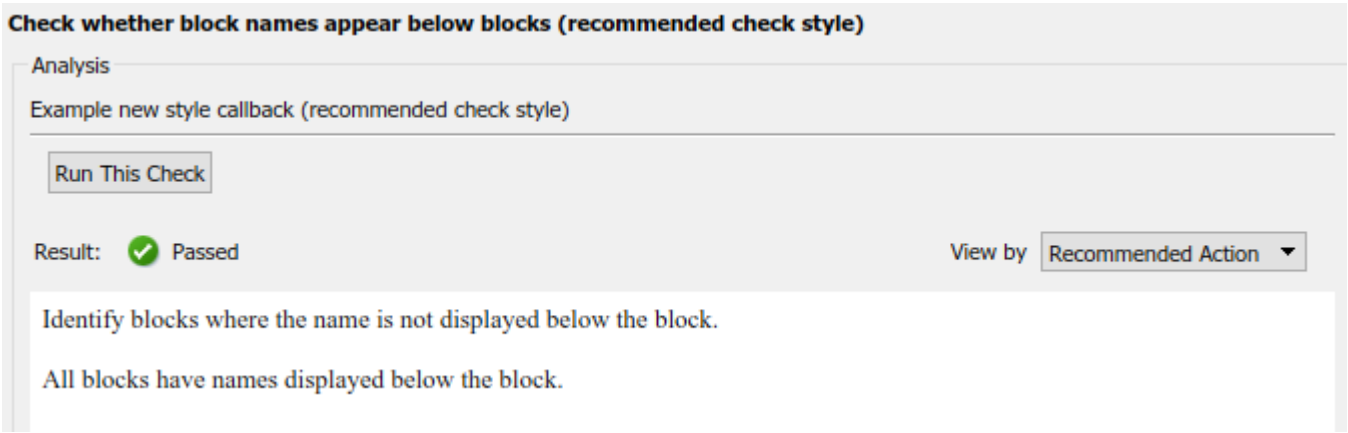
The report provides a recommended action for each check. You can click the hyperlink path to open the violating block in the model editor. For example:



10 Follow the recommended action for fixing the violating blocks by using one of these methods:

- Update each violation individually by double-clicking the hyperlink to open the block. Right-click the block and select **Rotate & Flip > Flip Block Name**.
- Select the **Make block names appear below blocks** button. The Model Advisor automatically fixes the issues in the model. Notice that the button is dimmed after the violations are fixed.

11 Save the model and rerun the Model Advisor check. The check passes.



See Also

`ModelAdvisor.FormatTemplate` | `ModelAdvisor.Check` | ModelAdvisor.Check.CallbackContext | `ModelAdvisor.FormatTemplate`

More About

- “Defining Custom Model Advisor Checks Workflow” on page 6-34
- “Create Pass/Fail and Informational Model Advisor Checks” on page 6-5
- “Create and Deploy a Model Advisor Custom Configuration” on page 7-25

Create Model Advisor 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. If desired, review the modelling guidelines:

- MathWorks Advisory Board (MAB) Modeling Guidelines (Simulink)
- High-Integrity System Modeling Guidelines (Simulink)
- Code Generation Modeling Guidelines (Simulink)

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 a Data File for a Configuration Parameter Check

This example shows how to create a data file that specifies configuration parameter values in the **Diagnostics** pane. A custom check warns when the configuration parameters values do not match the values defined in the data file.

At the command prompt, type `vdp` to open the `van der Pol Equation` model.

Right-click in the model window and select **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 the `Advisor.authoring.generateConfigurationParameterDataFile` function to create a data file specifying configuration parameter constraints in the **Diagnostics** pane. Also, 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** (`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** (`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** (`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`. Also, 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` provides tagging only for **Algebraic loop**, **Minimize algebraic loop**, and **Block Priority Violation**. For example, `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.m` file. First, you register the check using an `sl_customization.m` 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** parameter settings to `error`.

The check uses the `ex_DataFile.xml` data file created in “Create a Data File for a Configuration Parameter Check” on page 6-16.

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` to open the `van der Pol Equation` model.

Right-click in the model window and select **Model Configuration Parameters**. In the **Diagnostics** pane, set the configuration parameters as follows:

- **Algebraic loop** to `none`
- **Minimize algebraic loop** to `warning`
- **Block Priority Violation** to `warning`

In the **Modeling** tab, select **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 edit the configuration parameter. 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>` tag contains:

- `Description` - (Optional) Description of the check. Displayed in Model Advisor window.
- `PassMessage` - (Optional) Pass message displayed in Model Advisor window.
- `FailMessage` - (Optional) Fail message displayed in Model Advisor window.
- `RecommendedActions` - (Optional) Recommended actions displayed in Model Advisor window when check does not pass.

Note The `<messages>` tag is optional. `Advisor.authoring.generateConfigurationParameterDataFile` does not generate `<messages>` tagging.

In the `<checkdata>` tag, 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 tag 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:

  ```
  <PositiveModelParameterConstraint>
      <parameter>BlockPriorityViolationMsg</parameter>
  </PositiveModelParameterConstraint>
  <NegativeModelParameterConstraint>
      <parameter>AlgebraicLoopMsg</parameter>
  </NegativeModelParameterConstraint>
  ```

- `value` - Specifies the setting(s) for the configuration parameter. You can specify more than one `value` tag.

  When using `PositiveModelParameterConstraint`, `value` specifies the setting(s) that you want for the configuration parameter. For `NegativeModelParameterConstraint`, `value` specifies the setting(s) you that do not want for the configuration parameter.

  You can specify the `value` using a format in this table.

Type	Format	Example

Scalar value	`<value>xyz</value>`	In this example, constraint `NegativeModelParameterConstraint` warns when the configuration parameter settings for configuration parameter is not `error` or `none` .
		
		`<NegattiveModelParameterConstraint>`
		`  <value>error</value>`
		`  <value>none</value>`
		`</NegativeModelParameterConstraint>`

Structure or object	`<value>`
	`  <param1>xyz</param1>`
	`  <param2>yza</param2>`
	`</value>`
		In this example, constraints `PositiveModelParameterConstraint` warns when the configuration parameter settings are not a valid structure:
		
		`<PositiveModelParameterConstraint>`
		`  <value>`
		`  <double>a</double>`
		`  <single>b</single>`
		`  </value>`
		`</PositiveModelParameterConstraint>`

Array	`<value>`
	`  <element>value</element>`
	`  <element>value</element>`
	`</value>`
		In this example, constraint `NegativeModelParameterConstraint` warns when the configuration parameter settings are an invalid array:
		
		`<NegativeModelParameterConstraint>`
		`  <value>`
		`  <element>A</element>`
		`  <element>B</element>`
		`  </value>`
		`</NegativeModelParameterConstraint>`

Type	Format	Example

Structure Array	`<value>`
	`  <element>`
	`  <param1>xyz</param1>`
	`  <param2>yza</param2>`
	`  </element>`
	`  <element>`
	`  <param1>xyz</param1>`
	`  <param2>yza</param2>`
	`  </element>`
	`</value>`
		In this example, constraint `NegativeModelParameterConstraint` warns when the configuration parameter settings are an invalid structure array:
		
		`<NegativeModelParameterConstraint>`
		`  <value>`
		`  <element>`
		`  <double>a</double>`
		`  <single>b</single>`
		`  </element>`
		`  <element>`
		`  <double>a</double>`
		`  <single>b</single>`
		`  </element>`
		`  </value>`
		`</NegativeModelParameterConstraint>`

- `fixvalue` - (Optional) Specifies the setting to use when applying the Model Advisor fix action.

  You can specify the `fixvalue` using a format in this table.

Type	Format	Example

Scalar value	`<fixvalue>xyz</fixvalue>`	In this example, the fix action tag specifies the new configuration parameter setting as `warning`.
		
		`<PositiveModelParameterConstraint>`
		`  <value>error</value>`
		`  <fixaction>warning</fixaction>`
		`</PositiveModelParameterConstraint>`

Structure or object	`<fixvalue>`
	`  <param1>xyz</param1>`
	`  <param2>yza</param2>`
	`</fixvalue>`
		In this example, the fix action tag specifies the new configuration parameter setting for a structure.
		
		`<PositiveModelParameterConstraint>`
		`  <value>`
		`  <double>a</double>`
		`  <single>b</single>`
		`  </value>`
		`  <fixvalue>`
		`  <double>c</double>`
		`  <single>d</single>`
		`  </fixvalue>`
		`</PositiveModelParameterConstraint>`

Type	Format	Example
Array	<fixvalue> <element>value</element> <element>value</element> </fixvalue>	In this example, the fix action tag specifies the new configuration parameter setting for an array. <NegativeModelParameterConstraint> <value> <element>A</element> <element>B</element> </value> <fixvalue> <element>C</element> <element>D</element> </fixvalue> </NegativeModelParameterConstraint>
Structure Array	<fixvalue> <element> <param1>xyz</param1> <param2>yza</param2> </element> <element> <param1>xyz</param1> <param2>yza</param2> </element> </fixvalue>	In this example, the fix action tag specifies the new configuration parameter settings for a structure array. <NegativeModelParameterConstraint> <value> <element> <double>a</double> <single>b</single> </element> <element> <double>a</double> <single>b</single> </element> </value> <fixvalue> <element> <double>c</double> <single>d</single> </element> <element> <double>c</double> <single>d</single> </element> </fixvalue> </NegativeModelParameterConstraint>

- `dependson` - (Optional) Specifies a prerequisite subcheck.

  In this example, `dependson` specifies that configuration parameter subcheck `ID_B` must pass before configuration parameter subcheck `ID_A` runs.

```
<PositiveModelParameterConstraint id="ID_A">
    <dependson>ID_B</value>
</PostitiveModelParameterConstraint>
```

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>
</PostitiveModelParameterConstraint>
```

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>
</PostitiveModelParameterConstraint>
```

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>
</PostitiveModelParameterConstraint>
```

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”

Define Model Advisor Checks for Supported or Unsupported Blocks and Parameters

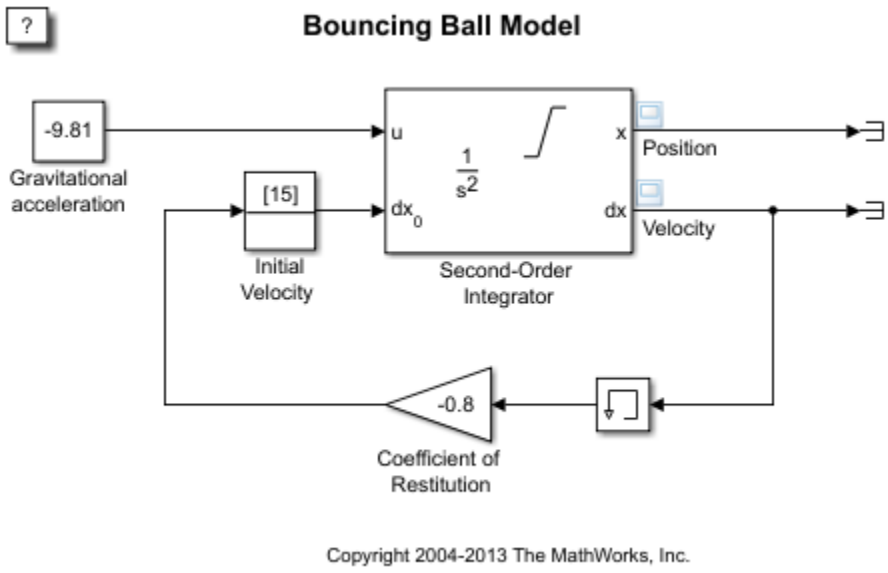
For modeling guidelines that require you to use a subset of block or parameter values, you can create Model Advisor checks in which you specify these constraints:

- Supported or unsupported block parameter values
- Supported or unsupported model parameter values
- Supported or unsupported blocks
- Check for whether blocks or parameters meet a combination of constraints

You can also create constraints that check for prerequisite constraints before checking the actual constraint. You can check your model against these constraints as you edit or run the checks interactively after you complete your model design.

Example

The `sldemo_bounce` model simulates a ball bouncing on Earth. In this example, you create two Model Advisor checks consisting of constraints. Then, check your model against those constraints.



Create Block Parameter Constraints

1 Create these block parameter constraints:

```
c1=Advisor.authoring.PositiveBlockParameterConstraint;
c1.ID='ID_1';
c1.BlockType='Gain';
c1.ParameterName='Gain';
c1.SupportedParameterValues={'-.7'};
c1.ValueOperator='eq';
```

```
c2=Advisor.authoring.NegativeBlockParameterConstraint;
c2.ID='ID_2';
c2.BlockType='InitialCondition';
c2.ParameterName='Value';
c2.UnsupportedParameterValues={'0'};
c2.ValueOperator='le';
```

Constraint `c1` specifies that a Gain block must have a value equal to `-.7`. Constraint `c2` specifies that the Initial Condition block must have a value less than or equal to zero.

2 Create this positive model parameter constraint.

```
c3=Advisor.authoring.PositiveModelParameterConstraint;
c3.ID='ID_3';
c3.ParameterName='SolverType';
c3.SupportedParameterValues={'Variable-step'};
```

Constraint `c3` specifies that the **Solver** parameter must be equal to `Variable-step`.

3 Create this positive block type constraint:

```
c4=Advisor.authoring.PositiveBlockTypeConstraint;
c4.ID='ID_5';
s1=struct('BlockType','Constant','MaskType','');
s2=struct('BlockType','Subsystem','MaskType','');
s3=struct('BlockType','InitialCondition','MaskType','');
s4=struct('BlockType','Gain','MaskType','');
s5=struct('BlockType','Memory','MaskType','');
s6=struct('BlockType','SecondOrderIntegrator','MaskType','');
s7=struct('BlockType','Terminator','MaskType','');
c4.SupportedBlockTypes={s1;s2;s3;s4;s5;s6;s7;};
c4.addPreRequisiteConstraintID('ID_3');
```

Constraint `c4` specifies the supported blocks. Constraint `c3` is a prerequisite to `c4` meaning that the Model Advisor only checks `c4` if `c3` passes.

4 Create a data file that contains these constraints. This data file corresponds to one Model Advisor check.

```
Advisor.authoring.generateBlockConstraintsDataFile( ...
              'sldemo_constraints_1.xml','constraints',{c1,c2,c3,c4});
```

The data file contains tagging specifically for constraints.

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck>
   <checkdata>
      <PositiveBlockParameterConstraint BlockType="Gain" id="ID_1">
         <parameter type="string">Gain</parameter>
         <value>-.7</value>
         <operator>eq</operator>
      </PositiveBlockParameterConstraint>
      <NegativeBlockParameterConstraint BlockType="InitialCondition" id="ID_2">
         <parameter type="string">Value</parameter>
         <value>0</value>
         <operator>le</operator>
      </NegativeBlockParameterConstraint>
      <PositiveModelParameterConstraint id="ID_3">
         <parameter type="enum">SolverType</parameter>
         <value>Variable-step</value>
      </PositiveModelParameterConstraint>
      <PositiveBlockTypeConstraint id="ID_5">
         <BlockType MaskType="">Constant</BlockType>
         <BlockType MaskType="">Subsystem</BlockType>
         <BlockType MaskType="">InitialCondition</BlockType>
         <BlockType MaskType="">Gain</BlockType>
         <BlockType MaskType="">Memory</BlockType>
         <BlockType MaskType="">SecondOrderIntegrator</BlockType>
```

```
        <BlockType MaskType="">Terminator</BlockType>
        <dependson>ID_3</dependson>
      </PositiveBlockTypeConstraint>
      <CompositeConstraint>
        <ID>ID_1</ID>
        <ID>ID_2</ID>
        <ID>ID_5</ID>
        <operator>and</operator>
      </CompositeConstraint>
   </checkdata>
</customcheck>
```

Note For model configuration parameter constraints, use the `Advisor.authoring.generateBlockConstraintsDataFile` method only when specifying model configuration parameter constraints as prerequisites to block constraints or as part of a composite constraint consisting of block and model configuration parameter constraints. For other cases, use the `Advisor_authoring.generateConfigurationParameterDatafile` method.

5 Create two block parameter constraints and a composite constraint.

```
cc1=Advisor.authoring.PositiveBlockParameterConstraint;
cc1.ID='ID_cc1';
cc1.BlockType='SecondOrderIntegrator';
cc1.ParameterName='UpperLimitX';
cc1.SupportedParameterValues={'inf'};
cc1.ValueOperator='eq';

cc2=Advisor.authoring.PositiveBlockParameterConstraint;
cc2.ID='ID_cc2';
cc2.BlockType='SecondOrderIntegrator';
cc2.ParameterName='LowerLimitX';
cc2.SupportedParameterValues={'0.0'};
cc2.ValueOperator='eq';

cc=Advisor.authoring.CompositeConstraint;
cc.addConstraintID('ID_cc1');
cc.addConstraintID('ID_cc2');
cc.CompositeOperator='and';
```

Constraint `cc1` specifies that for a Second-Order Integrator block, the **Upper limit x** parameter must have a value equal to `inf`. Constraint `cc2` specifies that for a Second-Order Integrator block, the **Lower limit x** parameter must have a value equal to zero. Constraint `cc` specifies that for this check to pass, both `cc1` and `cc2` have to pass.

6 Create a data file that contains these constraints. This data file corresponds to a second Model Advisor check.

```
Advisor.authoring.generateBlockConstraintsDataFile( ...
                'sldemo_constraints_2.xml','constraints',{cc1,cc2,cc});
```

Create Model Advisor Checks from Constraints

1 To specify and register these checks, use this `sl_customization.m` file.

```
function sl_customization(cm)

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

% register items to factory group.
cm.addModelAdvisorTaskFcn(@defineModelAdvisorGroups);
```

```
% defineModelAdvisorChecks
function defineModelAdvisorChecks

% check1
rec = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0001');
rec.Title = 'Example1: Check block parameter constraints';
rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec.TitleTips = 'Example check block parameter constraints';

% --- data file input parameters
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Data File';
inputParam1.Value = 'sldemo_constraints_1.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});
rec.SupportExclusion = false;
rec.SupportLibrary = true;

% check2
rec1 = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0002');
rec1.Title = 'Example2: Check block parameter constraints';
rec1.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec1.TitleTips = 'Example check block parameter constraints';

% --- data file input parameters
rec1.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Data File';
inputParam1.Value = 'sldemo_constraints_2.xml';
inputParam1.Type = 'String';
inputParam1.Description = 'Name or full path of XML data file.';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec1.setInputParameters({inputParam1});
rec1.SupportExclusion = false;
rec1.SupportLibrary = true;
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);
mdladvRoot.register(rec1);

%% defineModelAdvisorGroups
function defineModelAdvisorGroups
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group 1
rec = ModelAdvisor.FactoryGroup('com.mathworks.Test.factoryGroup');
rec.DisplayName='Example: My Group';
rec.addCheck('mathworks.check_0001');
rec.addCheck('mathworks.check_0002');

mdladvRoot.publish(rec);
```

You must use the `Advisor.authoring.createBlockConstraintCheck` function to create the `ModelAdvisor.Check` object and specify the constraint data file as an input parameter to this object.

2. At the command prompt, type create the **Example1: Check block parameter constraints** and **Example2: Check block parameter constraints** checks by typing this command:

   ```
   Advisor.Manager.refresh_customizations
   ```

3. At the command prompt, type `sldemo_bounce`.
4. In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.
5. In the left pane, select **By Task > Example: My Group**. For each check, in the right pane, the **Data File** parameters are set to the data files that you previously created.
6. Click **Run Selected Checks**.

7 The **Example1: Check block parameter constraints** check produces a warning because the Gain block has a value of `-0.8` not `-0.7`. The **Example2: Check block parameter constraints** check passes because the Second-Order Integrator block meets both constraints.

You can use edit-time checking for custom checks that define block and parameter constraints. To enable edit-time checking, in the Model Advisor Configuration Editor, select the checks that contain the constraints. For more information on edit-time checking, see “Check Your Model by Using Edit Time Checks” on page 3-6.

See Also

`Advisor.authoring.generateBlockConstraintsDataFile` | `NegativeBlockParameterConstraint` | `NegativeBlockTypeConstraint` | `NegativeModelParameterConstraint` | `PositiveBlockParameterConstraint` | `PositiveBlockTypeConstraint` | `PositiveModelParameterConstraint`

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 `configure` and `process_results`.
`system`	Input	Path	Model or subsystem that the Model Advisor analyzes.
`checkCellArray`	Input/Output	Cell array	As input, the array of checks constructed in the check definition function. As output, the array of checks modified by actions in the configure stage.
`taskCellArray`	Input/Output	Cell array	As input, the array of tasks constructed in the task definition function. As output, the array of tasks modified by actions in the configure stage.

Process Callback Function

This example shows a process callback function that specifies actions in the `configure` stage that makes 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
    % 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
```

Tips for Using the Process Callback Function in a sl_customization File

Observe the following tips when using process callback function in a `sl_customization` file:

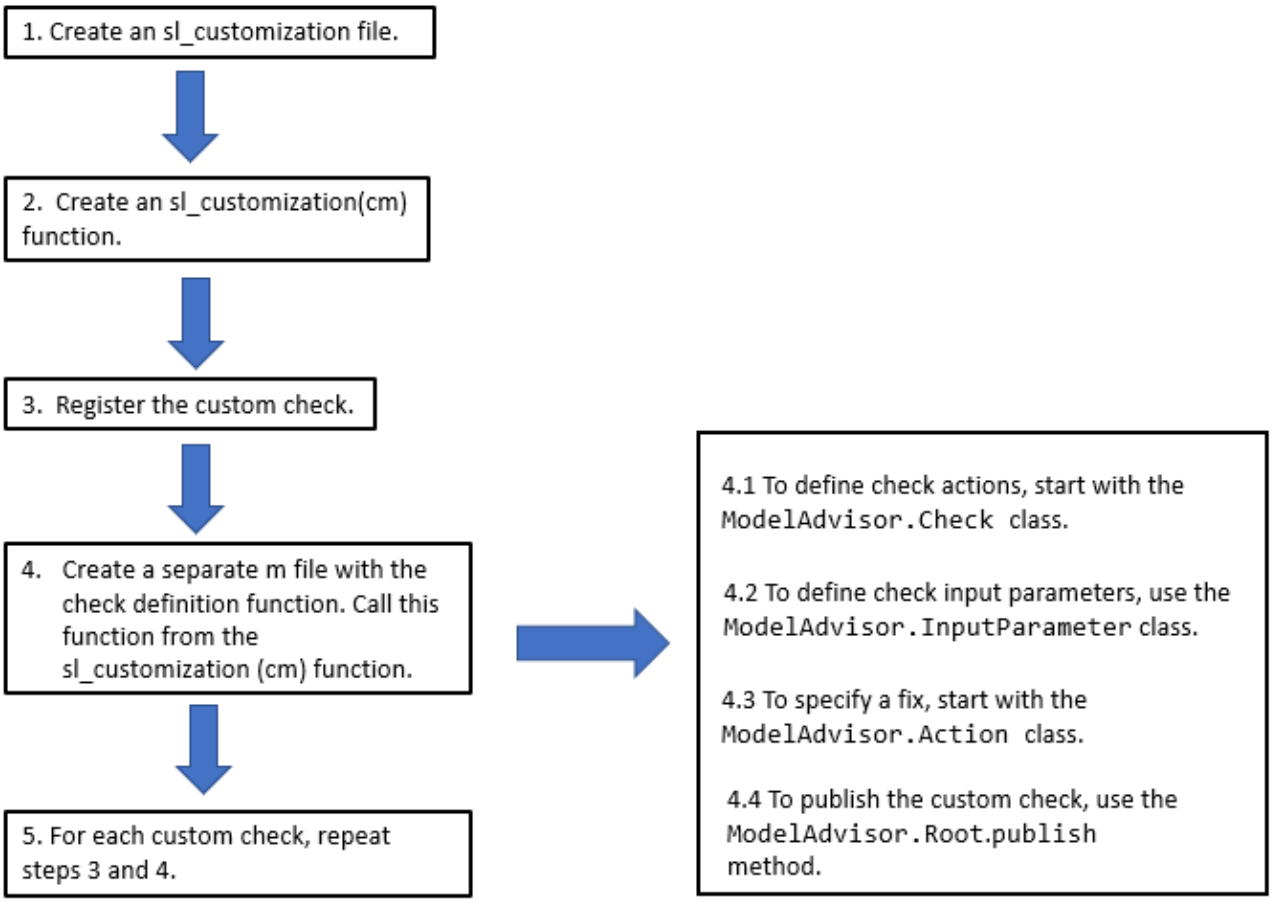
- If you delete a check in the Model Advisor Configuration Editor, you can retrieve a copy of it from the **Library** tab in the Model Advisor Configuration Editor. However, if you use a process callback function in a `sl_customization` file to hide checks and folders, the Model Advisor Configuration Editor does not display the hidden checks and folders. For a complete list of checks and folders, remove process callback functions and update the Simulink environment.
- 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.
- If you add process callbacks within the `sl_customization.m` file, include methods for registering the process callbacks in the `sl_customization` function.

See Also

“Defining Custom Model Advisor Checks Workflow” on page 6-34 | “Programmatically Customize Tasks and Folders for the Model Advisor” on page 7-15 | “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3

Defining Custom Model Advisor Checks Workflow

To specify your own set of conditions and model configuration settings for the Model Advisor to review, create custom checks. This graphic provides an overview of the workflow for creating and defining custom checks. The sections below provide details on each workflow step.



Create the sl_customization File and Function

To define a custom check, begin by creating an `sl_customization.m` file on the MATLAB path. In the `sl_customization.m` file, create an `sl_customization` function. The `sl_customization` function accepts one argument, a customization manager object:

```
function sl_customization(cm)
```

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 Model Advisor checks and folders 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.

Register Custom Checks

To register custom checks, use the `addModelAdvisorCheckFcn` method, which is part of the customization manager object that you input to the `sl_customization` function. Add this code to your `sl_customization.m` file:

```
function sl_customization(cm)
% register custom checks
cm.addModelAdvisorCheckFcn(@checkDefinitionFcn)
```

The `addModelAdvisorCheckFcn` method registers the check that you define in the `checkDefinitionFcn` to the **By Product** folder of the Model Advisor. The `checkDefinitionFcn` argument is a handle to the function that defines the custom check that you want to add to the Model Advisor as an instance of the `ModelAdvisor.Check` class.

Create a Check Definition Function

The check definition function defines the actions that the Model Advisor takes when you run the check. For each check that you create, you should define a check definition function. The following sections describe the key components of the check definition function.

Create an Instance of the ModelAdvisor.Check Class

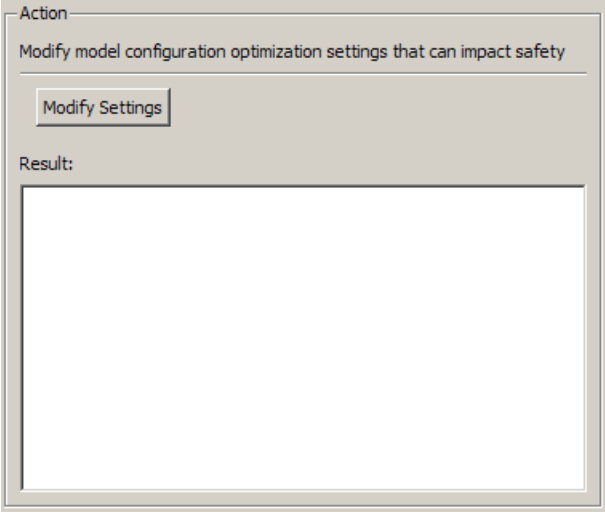
For each custom check, create one instance of the `ModelAdvisor.Check` class. Use the `ModelAdvisor.Check` properties and methods to define the check user interface and actions. This table describes some key check components.

Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this ID to access the check.
Handle to the 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.
Model compiling (optional)	Specifies whether the model is compiled for check analysis.
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 a fixing action.

Define Check Actions and a Fix

The check definition function contains a check callback function that specifies the actions that you want the Model Advisor to perform on a model or subsystem. Define the check callback function and pass a handle to it to the `setCallbackFcn` method. The Model Advisor executes the callback function when you run the check. Callback functions provide one or more return arguments that display the results after executing the check.

If you are specifying a custom check fix, the check definition function should also contain an action callback function. In the check definition function, create an instance of the `ModelAdvisor.Action` class. Define the action callback function and pass a handle to it to the `setCallbackFcn` method. When you define an action, the Model Advisor includes an **Action** box below the **Analysis** box. The **Action** box contains a button for fixing the model or subsystem. In the example below, the button name is **Modify Settings**.



Callback and action callback functions provide one or more return arguments for displaying the results after executing the check. See “Create the Check Callback Definition Function” on page 6-11 and “Create the Action Callback Definition Function” on page 6-12.

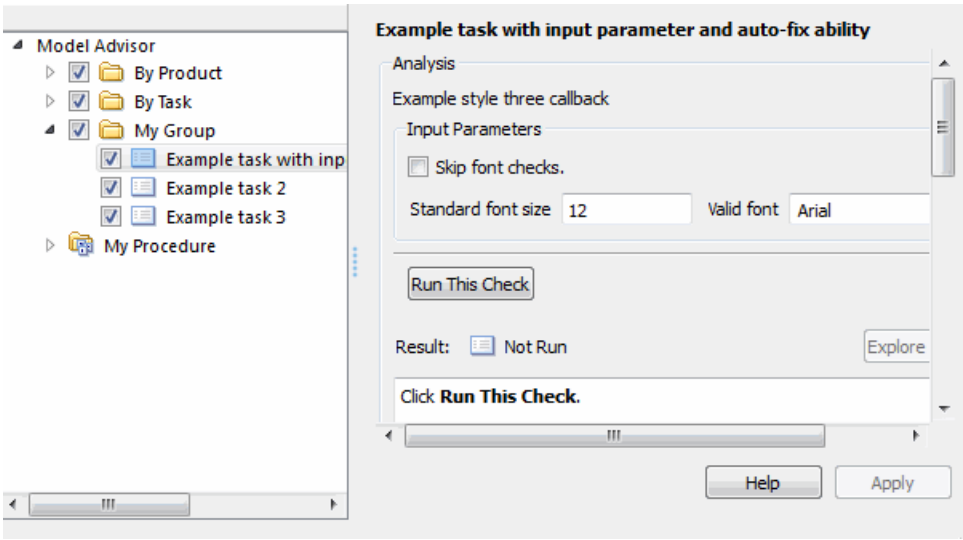
Define Check Input Parameters

With input parameters, you can request input before running the check. Define input parameters using the `ModelAdvisor.InputParameter` class. You must include input parameter definitions inside a custom check definition 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
`setInputParametersLayoutGrid`	Specifies the size of the input parameter grid.
`setRowSpan`	Specifies the number of rows the parameter occupies in the input parameter layout grid.
`setColSpan`	Specifies the number of columns the parameter occupies in the input parameter layout grid.

The Model Advisor displays input parameters in the **Input Parameters** box.



Define Where Custom Checks Appear

Create a folder for custom checks in the **By Product** folder by using the `publish` method. Then use the Model Advisor Configuration Editor to customize the folders within the Model Advisor tree. For more information, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

You can customize the Model Advisor by using the `ModelAdvisor.Group` and `ModelAdvisor.FactoryGroup` classes instead of the Model Advisor Configuration Editor. However, these APIs are a less flexible and more time-consuming way of customizing the Model Advisor. To place customized checks in custom folders at the top-level of the Model Advisor tree (the Model Advisor root), use the `ModelAdvisor.Group` class. To place customized checks in new folders in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. You must include methods that register these tasks and folders in the `sl_customization` function.

Format Check Results

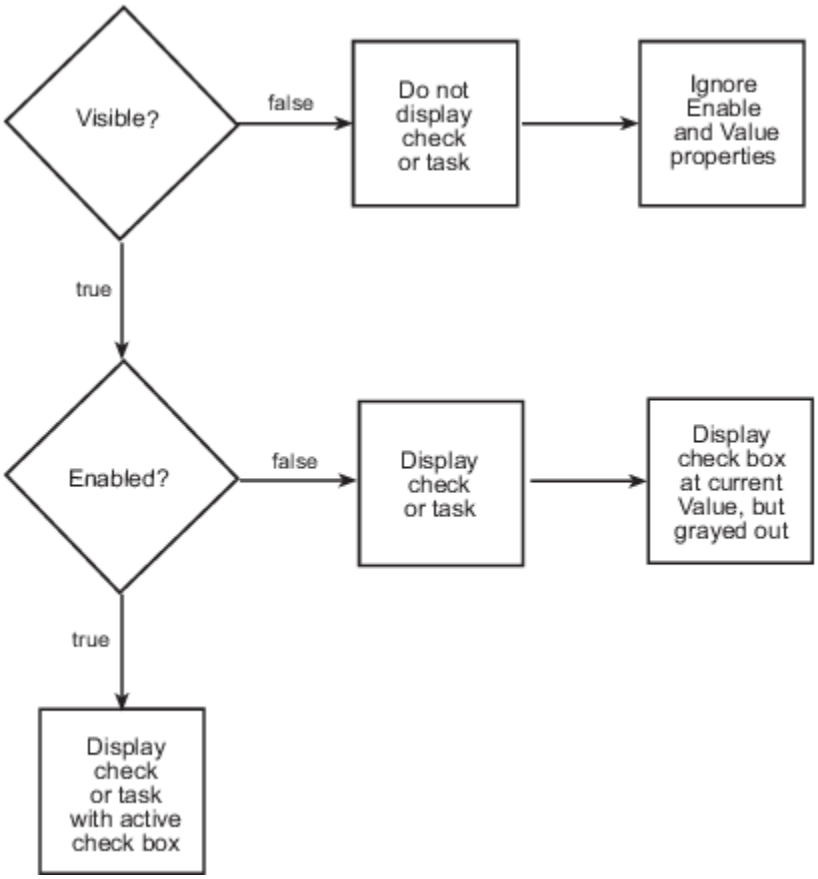
To use default formatting for Model Advisor results, specify the callback function type as `DetailStyle` in the `setCallbackFcn` method. If the default formatting does not meet your needs, use one of the other callback function styles and either the `ModelAdvisor.FormatTemplate` class or these other Model Advisor formatting APIs:

Class	Description
`ModelAdvisor.Text`	Create a Model Advisor text output.
`ModelAdvisor.List`	Create a list.
`ModelAdvisor.Table`	Create a table.
`ModelAdvisor.Paragraph`	Create and format a paragraph.
`ModelAdvisor.LineBreak`	Insert a line break.
`ModelAdvisor.Image`	Include an image in the Model Advisor output.

Display and Enable Checks

You can specify how a custom check 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.

The following chart illustrates how the `Visible`, `Enable`, and `Value` properties interact.



If you add checks to the Model Advisor as tasks, specify these properties in the `ModelAdvisor.Task` class. If you specify the properties in both the `ModelAdvisor.Check` and `ModelAdvisor.Task` classes, the `ModelAdvisor.Task` properties take precedence, except for the `Visible` and `LicenseName` properties.

See Also

`ModelAdvisor.Action` | `ModelAdvisor.Check` | `ModelAdvisor.FactoryGroup` | `ModelAdvisor.Group` | `ModelAdvisor.InputParameter` | `publish`

Related Examples

- “Create and Deploy a Model Advisor Custom Configuration” on page 7-25
- “Define the Compile Option for Custom Model Advisor Checks” on page 6-39

Define the Compile Option for Custom Model Advisor Checks

Depending on the implementation of your model and what you want your custom check to achieve, it is important that you specify the appropriate compile option so the intended information is evaluated by your custom check.

You use the ModelAdvisor.Check.CallbackContext property to define the compile option:

- `None` specifies that the Model Advisor does not have to compile your model before analysis by your custom check.
- `PostCompile` specifies that the Model Advisor must compile the model to update the model diagram and then simulate the model to execute your custom check.
- `PostCompileForCodegen` specifies that the Model Advisor must compile and update the model diagram specifically for code generation, but does not simulate the model. Use this option for Model Advisor checks that analyze code generation readiness of the model.

Checks for Models That Are Not Compiled by the Model Advisor

For custom checks that do not require the Model Advisor to compile the model before execution of the check, in the check definition you specify the ModelAdvisor.Check.CallbackContext property as:

```
rec.CallbackContext = 'None';
```

Note By default, the Model Advisor does not compile the model for custom checks. You do not have to include the ModelAdvisor.Check.CallbackContext property in the check definition.

```
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
% ------------------------------
% Sample Check: Check whose model does not need to be compiled
% ------------------------------

rec = ModelAdvisor.Check('exampleCheck2');
rec.Title = 'Non-compile check example';
rec.TitleID = 'custom.dtcCheck.NonCompile1';
rec.TitleTips = 'A custom check for a model that does not need to be compiled ';
rec.setCallbackFcn(@CheckNoCompile,'None','StyleOne');
rec.CallbackContext = 'None'; % Not compiled

mdladvRoot.publish(rec, 'Demo');
```

Checks That Require the Model to be Compiled and Simulated by the Model Advisor

For custom checks that require model compilation and simulation to properly check the implementation of the model, in the check definition you specify the ModelAdvisor.Check.CallbackContext property as:

```
rec.CallbackContext = 'PostCompile';
```

In this situation, the Model Advisor updates the model diagram and simulates the model. The Model Advisor does not flag modeling issues that fail during code generation because these issues do not affect the simulated model.

This example shows a check definition that requires a model to be compiled and simulated.

```
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
% ----------------------------
% Sample Check: Check whose model must be compiled and simulated.
% ----------------------------

rec = ModelAdvisor.Check('exampleCheck3');
rec.Title = 'PostCompile check example';
rec.TitleID = 'custom.dtcCheck.Compile1';
rec.TitleTips = 'A custom check for a model that is compiled and simulated';
rec.setCallbackFcn(@CheckCompileSimulate,'None','StyleOne');
rec.CallbackContext = 'PostCompile'; % Compiled and simulated

mdladvRoot.publish(rec, 'Demo');
```

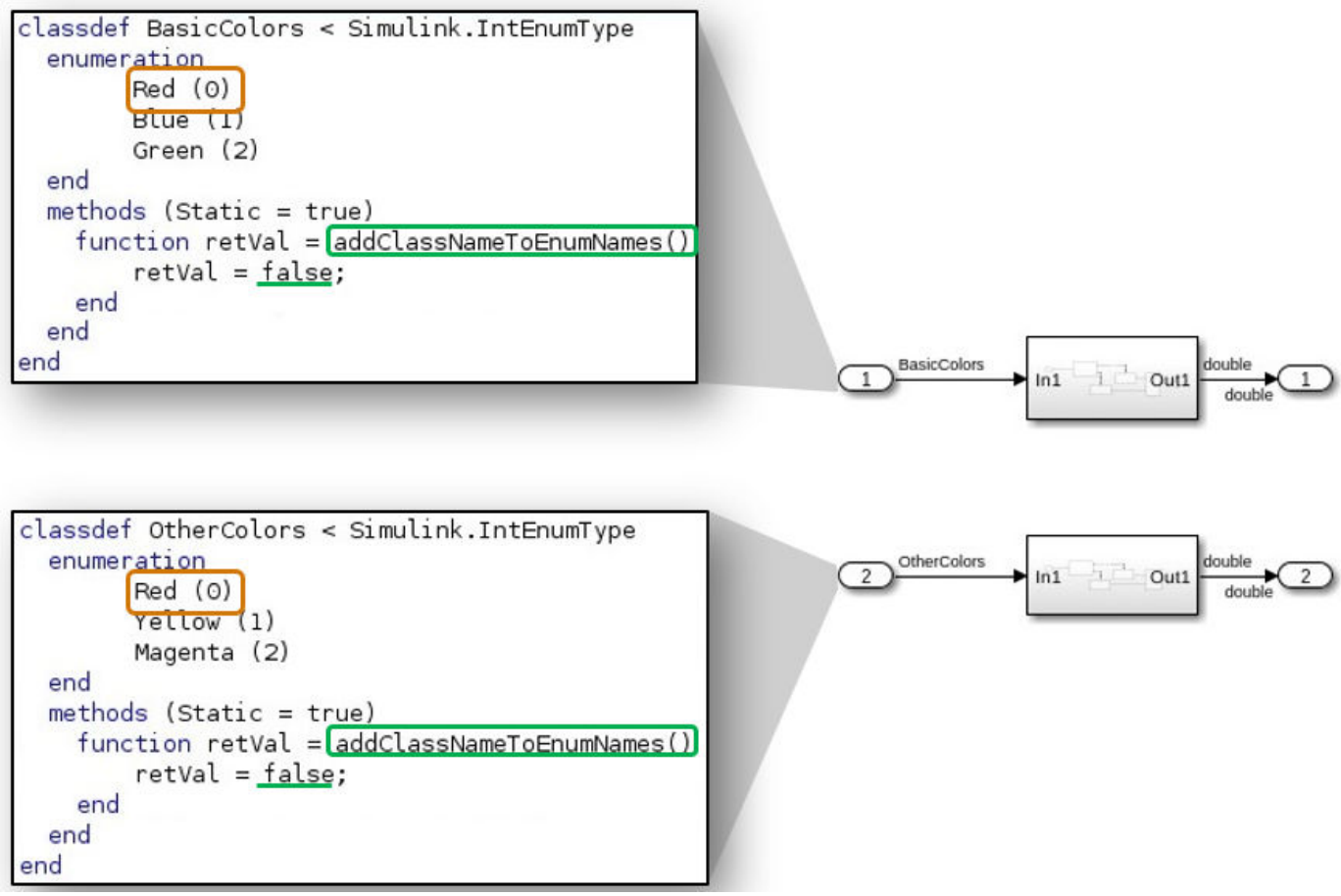
Checks That Evaluate Code Generation Readiness of the Model

For custom checks that evaluate code generation readiness, you must develop the model to generate code. In the check definition you specify the ModelAdvisor.Check.CallbackContext property as:

```
rec.CallbackContext = 'PostCompileForCodegen';
```

In this situation, the Model Advisor compiles the model and updates the model diagram specifically for code generation. The Model Advisor does not assume that the model is being simulated.

You can create custom Model Advisor checks that identify code generation setup issues in a model at an earlier stage, avoiding unexpected errors during code generation. For example, in this model, the `Red` enumeration in `BasicColors` and `OtherColors` are OK for use in a simulated model. In the generated code, however, these `Red` enumerations result in an enumeration clash. By using the `'PostCompileForCodegen'` option, your custom Model Advisor check can identify this type of code generation setup issue.



The `'PostCompileForCodegen'` option compiles the model for all variant choices. This compilation enables you to analyze possible issues present in the generated code for active and inactive variant paths in the model. An example is provided in “Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model” on page 6-42.

This example shows a check definition that requires a model to be compiled for code generation

```
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
% -----------------------------
% Sample Check: Check whose model is compiled for generated code.
% Model is not simulated.
% -----------------------------

rec = ModelAdvisor.Check('exampleCheck1');
rec.Title = 'PostCompileForCodegen check example';
rec.TitleID = 'custom.dtcCheck.CompileForCodegen1';
rec.TitleTips = 'A custom check for evaluating the generated code';
rec.setCallbackFcn(@CheckSingleToBoolConversion,'None','StyleOne');
rec.CallbackContext = 'PostCompileForCodegen'; % Compile for generated code

mdladvRoot.publish(rec, 'Demo');
```

Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model

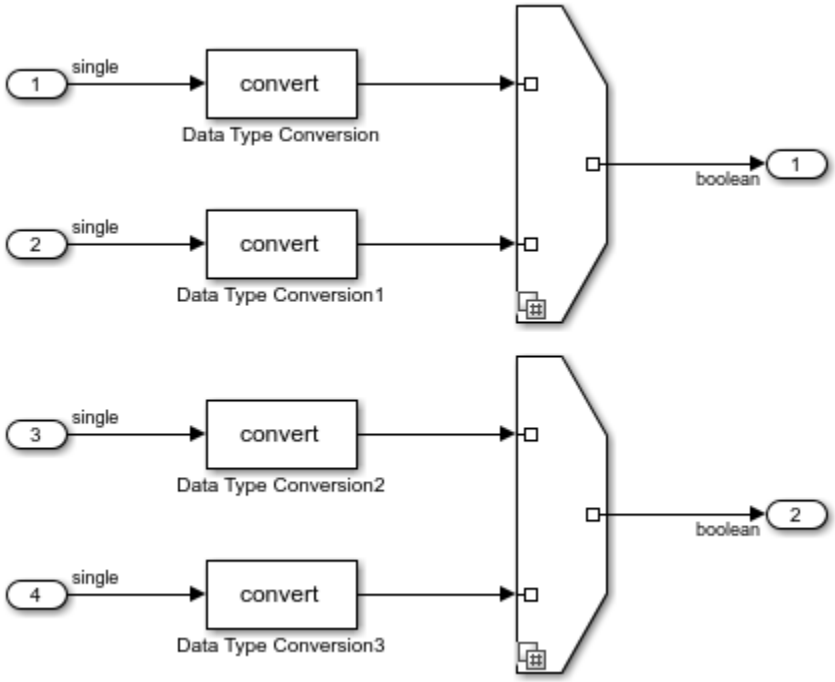
This example shows the creation of a custom Model Advisor check that evaluates active and inactive variant paths from a variant system model. The example provides Model Advisor results that demonstrate why you use `PostCompileForCodegen` versus `PostCompile` as the value for the `ModelAdvisor.Check.CallbackContext` property when generating code from the model is your final objective. See “Define the Compile Option for Custom Model Advisor Checks” on page 6-39.

Update Model to Analyze All Variant Choices

For the Model Advisor to evaluate active and inactive paths in a variant system, you must enable the **Analyze all choices during update diagram and generate preprocessor conditionals** option for the variant blocks (Variant Sink, Variant Source, and Variant Subsystem, Variant Model).

Note: Selecting this option can affect the execution time, thereby increasing the time it takes for the Model Advisor to evaluate the model.

1. Open the example model `ex_check_compile_code_gen`.
2. For each Variant Source block, open the block parameters and select the **Analyze all choices during update diagram and generate preprocessor conditionals** option.
3. Save the model to your local working folder.



Update `sl_customization.m` File

In your working folder, update the `sl_customization.m` file. Save your changes. If you are asked if it is ok to overwrite the file, click **OK**.

```
function sl_customization(cm)
```

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

end

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;

rec = ModelAdvisor.Check('exampleCheck1');
rec.Title = 'Check to identify SINGLE to BOOL conversions';
rec.TitleID = 'custom.dtcCheck.CompileForCodegen1';
rec.TitleTips = 'Custom check to identify SINGLE to BOOL conversions';
rec.setCallbackFcn(@CheckSingleToBoolConversion,'None','StyleOne');
rec.CallbackContext = 'PostCompileForCodegen'; % Compile for Code Generation

mdladvRoot.publish(rec, 'Demo');

end

% --- creates SimpleCallback function
function result = CheckSingleToBoolConversion(system)

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
result={};
dtcBlks = find_system(system, 'BlockType', 'DataTypeConversion');
for ii = numel(dtcBlks):-1:1
    dtcBlk = dtcBlks{ii};
    compDataTypes = get_param(dtcBlk, 'CompiledPortDataTypes');
    if isempty(compDataTypes)
        dtcBlks(ii) = [];
        continue;
    end
    if ~(strcmp(compDataTypes.Inport, 'single') && strcmp(compDataTypes.Outport, 'boolean'))
        dtcBlks(ii) = [];
        continue;
    end
end

ft = ModelAdvisor.FormatTemplate('ListTemplate');
ft.setInformation(['This check looks for data type conversion blocks that'...
    ' convert single data to boolean data']);
if ~isempty(dtcBlks)
    ft.setSubResultStatusText(['Check has failed. The following '...
        'data type conversion blocks convert single data to boolean:']);
    ft.setListObj(dtcBlks);
    ft.setSubResultStatus('warn');
    ft.setRecAction('Modify the model to avoid converting data type from single to boolean');
    mdladvObj.setCheckResultStatus(false);
else
    ft.setSubResultStatusText(['Check has passed. No data type conversion blocks '...
        'that convert single data to boolean were found.']);
    ft.setSubResultStatus('pass');
    mdladvObj.setCheckResultStatus(true);
end
ft.setSubBar(0);
result{end+1} = ft;
```

```
end

function result = dummy(~)
result={};
end
```

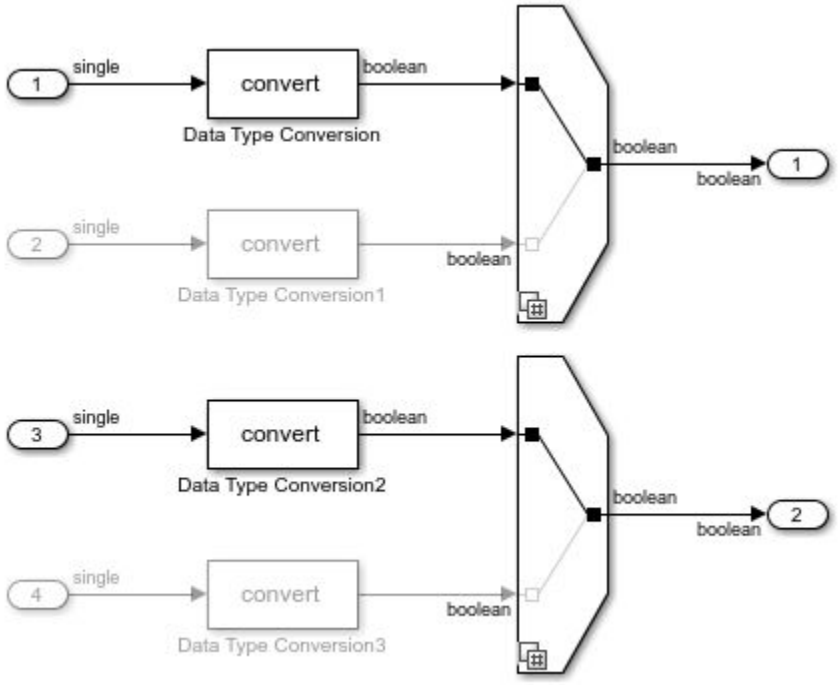
Open Model Advisor and Execute Custom Check

Prior to opening the Model Advisor and running the custom check, you must refresh the Model Advisor check information cache. In the MATLAB Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

To open the Model Advisor and execute the custom check:

1. Open your saved model.
2. In the **Modeling** tab, select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Click **OK**. The Model Advisor opens.
3. In the left pane, select **By Product > Demo > Check to identify SINGLE to BOOL conversion**. If the **By Product** folder is not displayed in the Model Advisor window, select **Settings > Preferences > Show By Product Folder**.
4. Right-click the check and select **Run This Check**. The Model Advisor compiles the model and executes the check. The Model Advisor updates the model diagram, with the inactive variant paths appearing as dimmed.



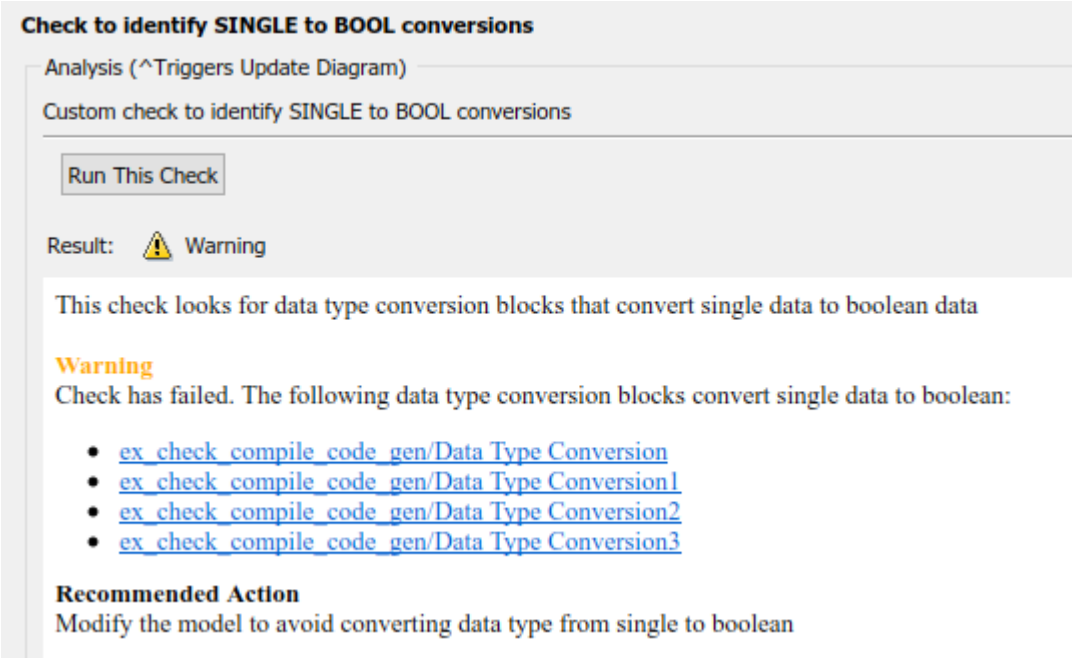
Review the Model Advisor Results

Review the check analysis results in the Model Advisor. Click the hyperlink path to open the violating block in the model editor.

In this example, because you defined the compile option in the `sl_customization.m` file as

```
rec.CallbackContext = 'PostCompileForCodegen';
```

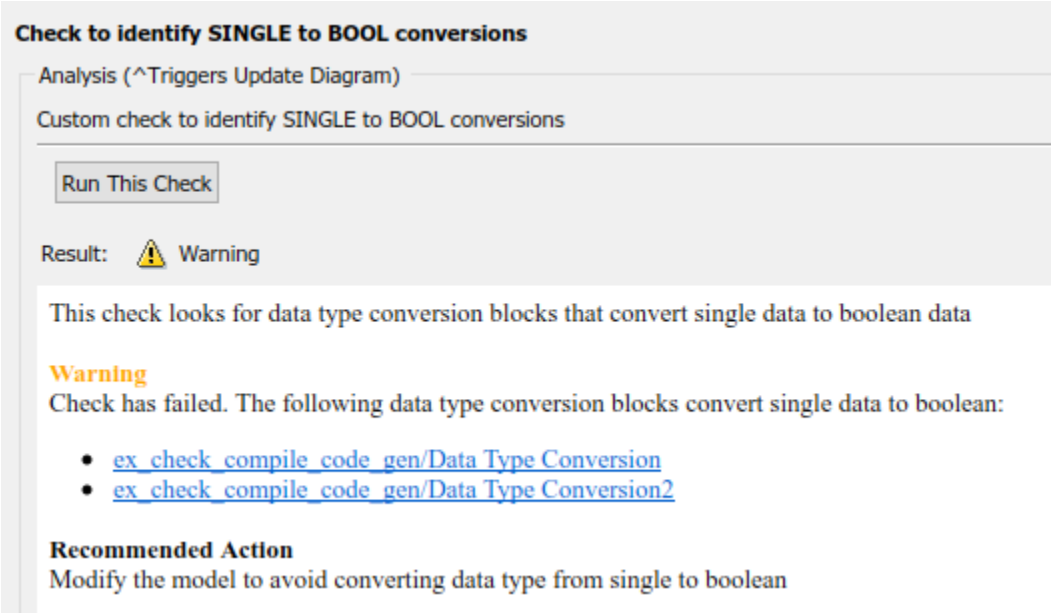
the Model Advisor generates warnings for the Data Type Conversion blocks in the active paths and the inactive paths of the Variant system.



If you defined the compile option in the `sl_customization.m` file as

```
rec.CallbackContext = 'PostCompile';
```

the results include only the Data Type Conversion blocks in the active path.



See Also

`ModelAdvisor.Check` | ModelAdvisor.Check.CallbackContext

More About

- “Defining Custom Model Advisor Checks Workflow” on page 6-34
- “Variant Systems” (Simulink)

Exclude Blocks From Custom Checks

This example shows how to exclude blocks from custom checks. To save time during model development and verification, you can exclude individual blocks from custom checks during a Model Advisor analysis. To exclude custom checks from Simulink blocks and Stateflow charts, use the `ModelAdvisor.Check.supportExclusion` and `Simulink.ModelAdvisor.filterResultWithExclusion` functions in the check definition file.

Update the Check Definition File

1. Set your folder to a writeable directory.
2. Copy the script `prepare_cust_chk_code` to your current folder and run the script. The script copies the files necessary for this example to your current folder.

   ```
   copyfile(fullfile(matlabroot,'examples','slcheck','main','prepare_cust_chk_code.m'),...
       'prepare_cust_chk_code.m','f');
   run('prepare_cust_chk_code.m');
   ```
3. Open the `defineDetailStyleCheck` file.
4. To update the **Check position of block names** check to exclude blocks during Model Advisor analysis, make two modifications to the `defineDetailStyleCheck` file.

   a. Enable the **Check position of block names** check to support check exclusions by using the `ModelAdvisor.Check.supportExclusion` property. After `rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');`, add `rec.supportExclusion = true;`. The first section of the `function defineDetailStyleCheck` now looks like:

   ```
   % Create ModelAdvisor.Check object and set properties.
   rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
   rec.Title = 'Check whether block names appear below blocks';
   rec.TitleTips = 'Check position of block names';
   rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
   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, modify the `DetailStyleCallback(system, CheckObj)` function as follows:

   ```
   % Find all blocks whose name does not appear below blocks
   violationBlks = find_system(system, 'Type','block',...
       'NamePlacement','alternate',...
       'ShowName', 'on');
   violationBlks = mdladvObj.filterResultWithExclusion(violationBlks);
   ```
5. Save the `DefineDetailStyleCheck` file. If you are asked if it is OK to overwrite the file, click **OK**.

Create and Save Exclusions

1. In order for your customizations to be visible in the Model Advisor, you must refresh the Model Advisor check information cache. At the MATLAB command prompt, type this command:

   ```
   Advisor.Manager.refresh_customizations();
   ```
2. To open the model, double-click `AdvisorCustomizationExample.slx`.
3. In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.

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.

4. In the left pane of the Model Advisor window, select the **By Product > Demo > Check whether block names appear below blocks** check. In the right pane, select **Run This Check**. The check fails.
5. In the Model Advisor window, check that **HighlightingHighlight Check Results** is selected. The blocks causing the **Check whether block names appear below blocks** check failure are highlighted in yellow.
6. In the model window, right-click the X block and select **Model Advisor > Exclude block only > Check position of block names**.
7. In the Model Advisor Exclusion Editor, click **OK** to create an exclusion file.
8. In the model window, open the `Amplifier` subsystem and right-click the `GainBlock` block and select **Model Advisor > Exclude block only > Check position of block names**.
9. In the Model Advisor Exclusion Editor, click **OK** to update the exclusion file.

Review Exclusions

1. In the Model Advisor, click **Settings > Preferences > Show Exclusion Tab**. Review the exclusions in the **Exclusions** tab.
2. In the left pane of the Model Advisor window, select the **By Product > Demo > Check position of block names** 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 applies during the analysis.
3. Close the model and the Model Advisor.

See Also

`Simulink.ModelAdvisor` | supportExclusion

Related Examples

- Example of Excluding Gain and Outport Blocks From Checks on page 3-15
- Excluding Blocks From Model Advisor Checks on page 3-10

More About

- “Run Model Advisor Checks and Review Results” on page 3-4
- “Address Model Check Results with Highlighting” (Simulink)

Model Advisor Customization

Customize the Configuration of the Model Advisor Overview

You can use Model Advisor API and the Model Advisor Configuration Editor to customize the configuration of the Model Advisor, including:

- Define which built-in (shipped) and custom Model Advisor checks are available in the Model Advisor and their order of execution
- Create custom folders and organize checks
- Designate the default configuration file for the Model Advisor
- Suppress the warning about missing checks when loading the Model Advisor configuration

To customize the Model Advisor to include custom checks and a custom configuration, perform the following tasks:

1. (Optional) Author custom checks in a customization file. See “Create Model Advisor Checks”.
2. Use the Model Advisor Configuration Editor to specify the folders and checks that you want to include in your custom Model Advisor configuration. See “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.
3. Update your Simulink environment so that the Model Advisor uses your configuration files. See “Update the Environment to Include Your Custom Configuration” on page 7-22
4. Open the Model Advisor, Load the configuration and verify that the appropriate configuration is being used and checks are available. See described in “Load and Verify a Custom Configuration” on page 7-23. . on page 7-23
5. (Optional) Deploy the custom configurations to your users. See “Deploy Custom Configurations” on page 7-24.
6. Verify that models comply with modeling guidelines. See “Run Model Advisor Checks and Review Results” on page 3-4.

Use the Model Advisor Configuration Editor to Customize the Model Advisor

Overview of the Model Advisor Configuration Editor

The Model Advisor Configuration Editor provides a way for you to specify the checks that you want to use for edit-time checking, as well as the checks included in the Model Advisor. This organizational hierarchy is saved as a configuration file, which is loaded when you initiate the Model Advisor. You can use the Model Advisor Configuration Editor to modify existing configurations, create new Model Advisor configurations, and specify the default configuration.

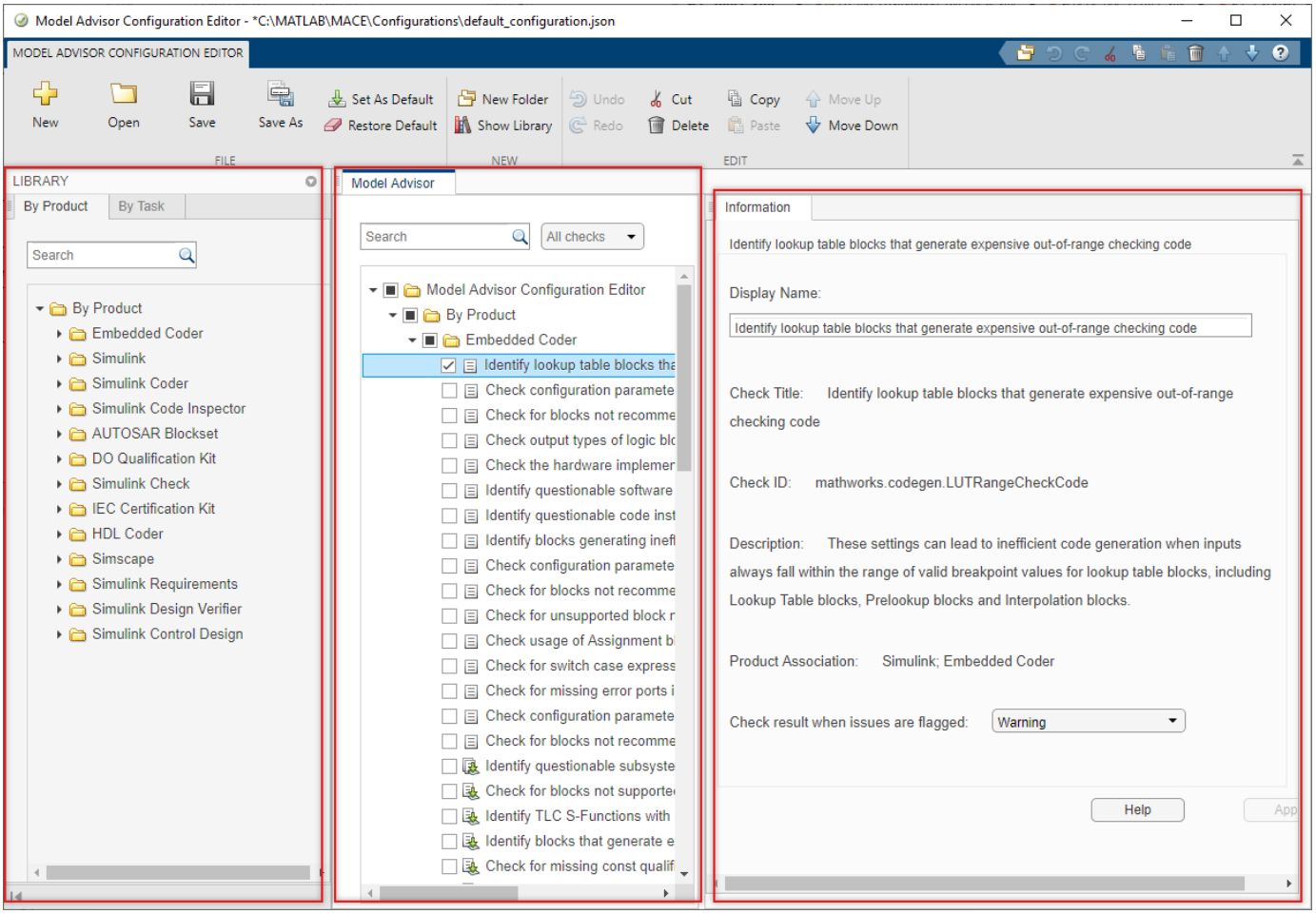
The Model Advisor Configuration Editor gives you the flexibility to customize the Model Advisor analysis to meet the needs of your organization by allowing you to:

- Review all available Model Advisor checks.
- Add, remove, and organize built-in checks and folders in the Model Advisor tree.
- Integrate custom Model Advisor checks in your verification and validation workflow.
- Disable and enable checks and folders.
- Rename checks and folders.
- Specify whether a check is marked as a warning or failure when it is flagged during a Model Advisor analysis.
- Suppress the warning about missing checks when loading the Model Advisor configuration.

The Model Advisor Configuration Editor includes:

- The **Library** pane — A read-only pane that lists all checks and folders that are available for use in the configuration, delineated by the **By Product** and **By Task** tabs. To permanently display the Library tab, click **Show Library** on the toolstrip.
- The **Model Advisor** pane — This pane lists the checks and folders in the current Model Advisor configuration, filtered by:
  - `All checks` — Lists all Model Advisor checks included in the current configuration
  - `Edit Time supported checks` — Lists only the Model Advisor checks that are supported as edit-time checks
- **Information** tab — This tab provides details about the check and folder, such as display name, check title and ID, description, license requirements, subchecks, and input parameters.

Use the search functionality in the **Library** and **Model Advisor** panes to locate specific checks and folders.



Model Advisor Configuration Editor

Open the Model Advisor Configuration Editor

Before opening 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.

When implementing custom checks or Model Advisor customizations by using the Model Advisor API, you must first update the Simulink environment to include your `sl_customization.m` file. At the MATLAB command line, enter

```
Advisor.Manager.refresh_customizations
```

Use one of these methods to open the Model Advisor Configuration Editor:

- Programmatically — At the MATLAB command line, enter `Simulink.ModelAdvisor.openConfigUI`.
- From the Simulink editor — In the **Modeling** tab, select **Model Advisor** > **Model Advisor Configuration Editor** .
- From the Model Advisor — Select **Settings** > **Open Configuration Editor**.

The configuration file that is currently being used by the Model Advisor displays when you open the Model Advisor Configuration Editor. The file name for the configuration is displayed at the top of the window. Verify that you are evaluating the correct configuration file. To open a different configuration file, click **Open** and browse to the file you would like to review.

To create a new configuration, click the **New** button on the toolstrip. Use **Save As** to rename the configuration file. Model Advisor configuration files are saved in `.json` format.

Specify a Default Configuration File

You can use the Model Advisor Configuration Editor to specify a default configuration that loads automatically when the Model Advisor opens. To set the default configuration, open the configuration file in the Model Advisor Configuration Editor and click the **Set As Default** button on the toolstrip.

Note If you have previously designated a default configuration, you can use **Clear default configuration setting** to clear the setting that designates the current default configuration file. Clicking the button does not modify the configuration that is currently displayed in the Model Advisor Configuration Editor. When you do not specify a default configuration, the Model Advisor uses the standard configuration that is defined by your system administrator.

If you do not specify the configuration file as the default, when you save the file, you are prompted as to whether to make the file the default configuration. To make this file the default configuration, click **Yes**.

Customize the Model Advisor Configuration

You can use the Model Advisor Configuration Editor to customize the Model Advisor configuration tree, including adding and removing checks and folders and specifying the order in which checks are executed. You can also disable the ability for users to select whether to include or exclude a check from an analysis. You can also use the Model Advisor Configuration Editor to define the input parameters for a check.

Note Checks that are copied from the **Library** tab retain their default parameter settings. When they are pasted into your custom configuration folder, the box beside each check is not selected.

Checks that are copied or cut from a folder in the **Model Advisor** tab retain their user-define parameter settings. When a check is included in multiple folders, you can specify different parameters for each check individually.

Organize the Hierarchy

You can customize the layout of the checks and folders in the Model Advisor configuration tree by using:

- **New Folder** to create a folder.
- **Copy**, **Cut**, and **Paste** to add, copy, and move checks and folders.
- **Delete** to remove checks and folders.
- **Move Up** or **Move Down** to shift the position of the check or folder in the configuration tree. The folders and checks that are higher in the configuration tree are executed first in the analysis.

Enable or Disable Checks

You can use the Model Advisor Configuration Editor to disable the check box control for checks and folders in the Model Advisor. By doing so, the check is still listed in the Model Advisor configuration tree, but it is dimmed and you do not have the ability to add or remove the check from the analysis.

In the **Model Advisor** pane, right-click on a folder or check and select **Disable**. Depending on the check box selection in the Model Advisor Configuration Editor, the following results occur in the Model Advisor:

- If the box beside check is selected in the Model Advisor Configuration Editor, then in the Model Advisor, the check is automatically selected. Because you selected **Disable**, the check is dimmed and you cannot choose to remove the check from the analysis.

  If the box beside the check is *not* selected and the **Disable** option is applied in the Model Advisor Configuration Editor, then in the Model Advisor, the check is not selected and you cannot included it in the analysis.
- If the box beside folder is selected in the Model Advisor Configuration Editor, then in the Model Advisor, the checks within the folder are automatically selected. Because you selected **Disable**, the folder and its checks are dimmed and you cannot choose to remove the checks from the analysis.

  If the box beside the folder is *not* selected and the **Disable** option is applied in the Model Advisor Configuration Editor, then in the Model Advisor, none of the checks within the folder are selected and you cannot include it in the analysis.

When a check or folder is disabled, you can use the **Enable** option to allow users to determine whether to include the check(s) in an analysis.

Note **Enable** and **Disable** affects the execution of checks in the analysis for both the Model Advisor user interface and edit-time checking.

Specify Parameters for Check Customization

You can use the Model Advisor Configuration Editor to customize a Model Advisor check, such as the display name and input parameters for the check.

In the **Information** tab, review the content that you can customize for the check:

- **Display Name** — Provide a new name for the check, which is displayed in the Model Advisor. Note that changing the display name does not change the check title.
- **Check result when issues are flagged** — Specify whether you want the check to be marked as a warning or failure in the results when the check flags an issue in your model. The default value is `Warning`. Select `Fail` to mark a flagged check as failed in the results.
- **Input Parameters** — Specify additional characteristics and functionality for the check. The Model Advisor uses these parameters to further define the emphasis of the analysis. For example, you can choose to include only subcheck **jc_0736_b** and specify the acceptable number of single-byte spaces in the analysis for Model Advisor check “Check indentation of code in Stateflow states”.

Suppress Warning Message for Missing Checks

The Model Advisor automatically warns you of checks that are missing when loading a Model Advisor configuration. You can use the Model Advisor Configuration Editor to suppress this message. Select the **Model Advisor Configuration Editor** root node and, in the **Information** tab, select **Suppress warning message for missing checks when loading configuration**.

Alternatively, you can programmatically suppress the Model Advisor warning by entering this command at the MATLAB command line:

```
warning('off','Simulink:tools:MALoadConfigMissCorrespondCheck')
```

Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration

You can use the Model Advisor Configuration Editor to organize the hierarchy of the Model Advisor and specify checks that are included in check analyses. This example shows how to create a new configuration file, specify checks for the Model Advisor and edit-time checking, define check parameters, and load the configuration to the Model Advisor.

Copy the Model to Your Current Directory

Copy model `AdvisorCustomizationExample.slx` and its supporting files, including the custom Model Advisor configuration file `demoConfiguration.json`, to your writeable directory.

1. In MATLAB, set your current folder to a writeable directory.

2. Enter this command in the MATLAB command prompt. MATLAB copies the example files to your writeable directory.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main','prepare_cust_chk_code.m'),...
    'prepare_cust_chk_code.m','f');
run('prepare_cust_chk_code.m');
```

Open the Model and Review the Model Advisor Configuration

In this example, you will refresh the Model Advisor check information cache, open the model, and review the Model Advisor configuration. Specifically, you will review the effect that the configuration has on the Model Advisor user interface and edit-time checking.

1. To refresh the Model Advisor check information cache, at the MATLAB command prompt, type:

```
Advisor.Manager.refresh_customizations();
```

2. Open the example model.

```
open_system('AdvisorCustomizationExample.slx');
```

3. On the **Modeling** tab, open the **Model Advisor**. You can also open the Model Advisor by typing this command at the MATLAB command prompt:

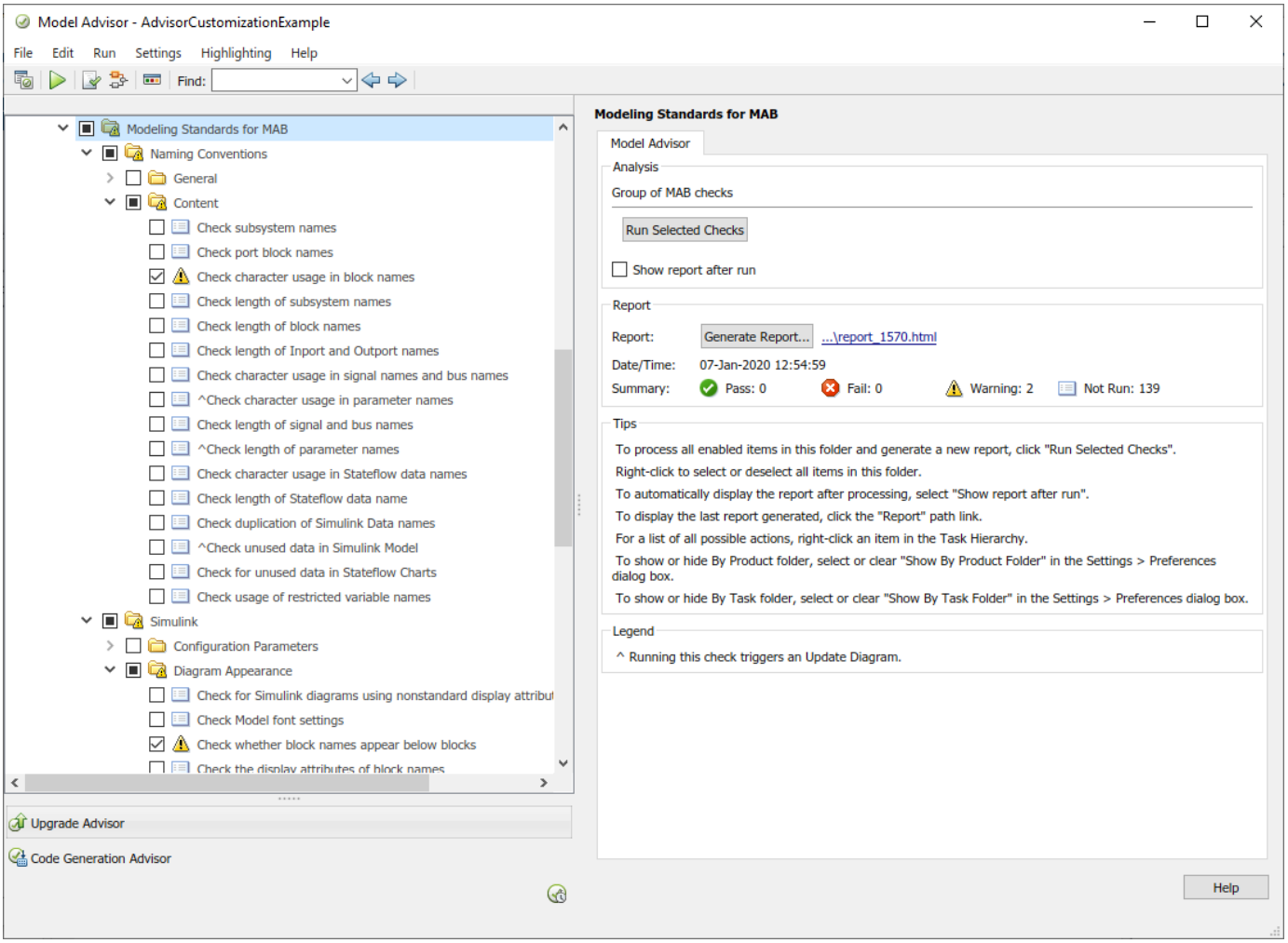
```
modeladvisor('AdvisorCustomizationExample.slx');
```

```
Updating Model Advisor cache...
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.Manage
```

4. In the **By Task** tab, make sure that the box beside these checks is selected:

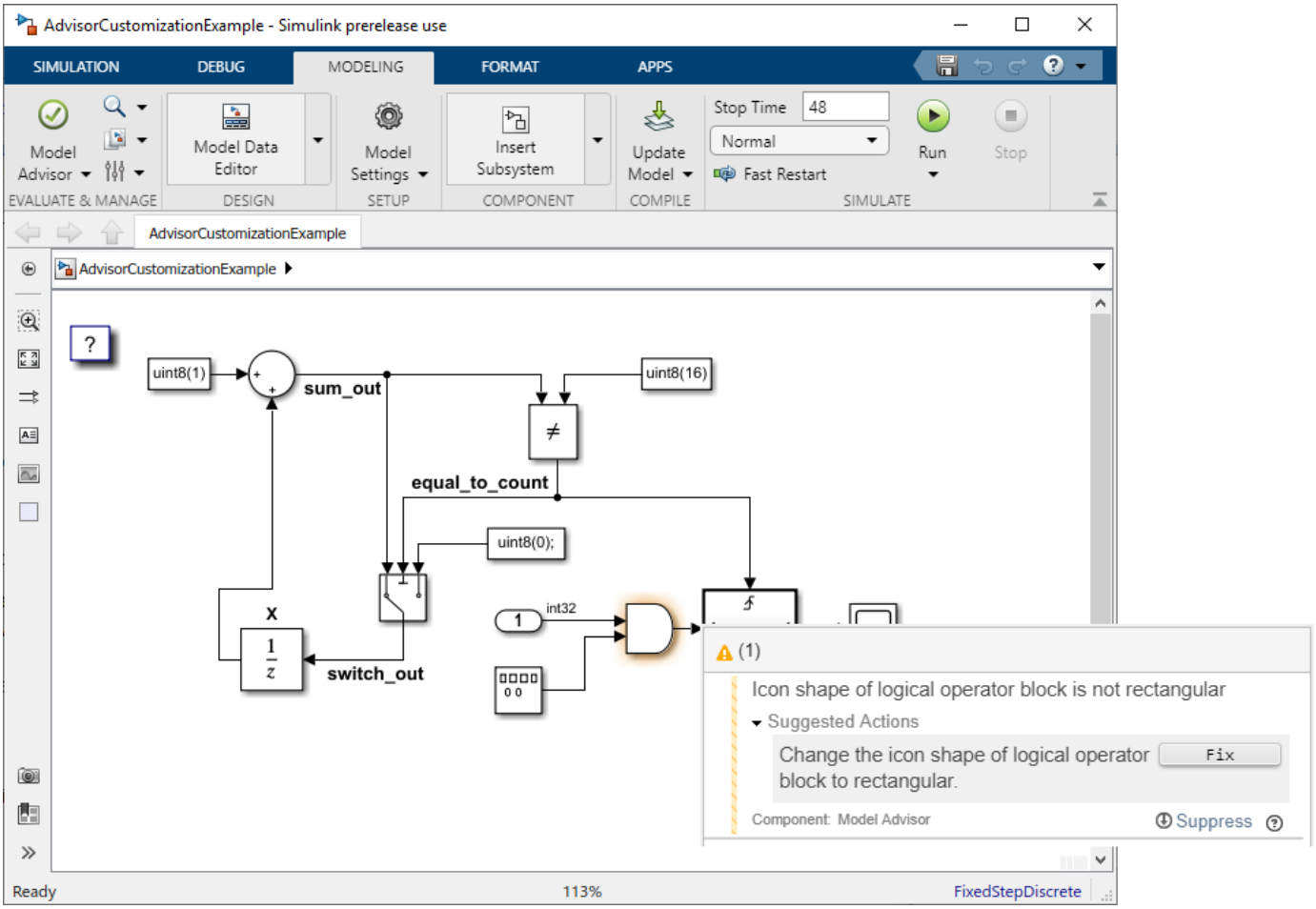
- **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names**
- **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks**

5. Right-click on the **Modeling Standards for MAB folder** and select **Run Selected Checks**. In the results, notice that both checks are marked with a warning. See “Run Model Advisor Checks and Review Results” on page 3-4 for more information about using the Model Advisor.



6. Return to the Simulink model editor.

7. On the **Modeling** tab, verify that the **Model Advisor > Edit-Time Checks** option is selected. Observe that the **Logical Operator** blocks is highlighted. Click on the warning icon above the block to view edit-time check violations. For more information, see “Check Your Model by Using Edit Time Checks” on page 3-6.

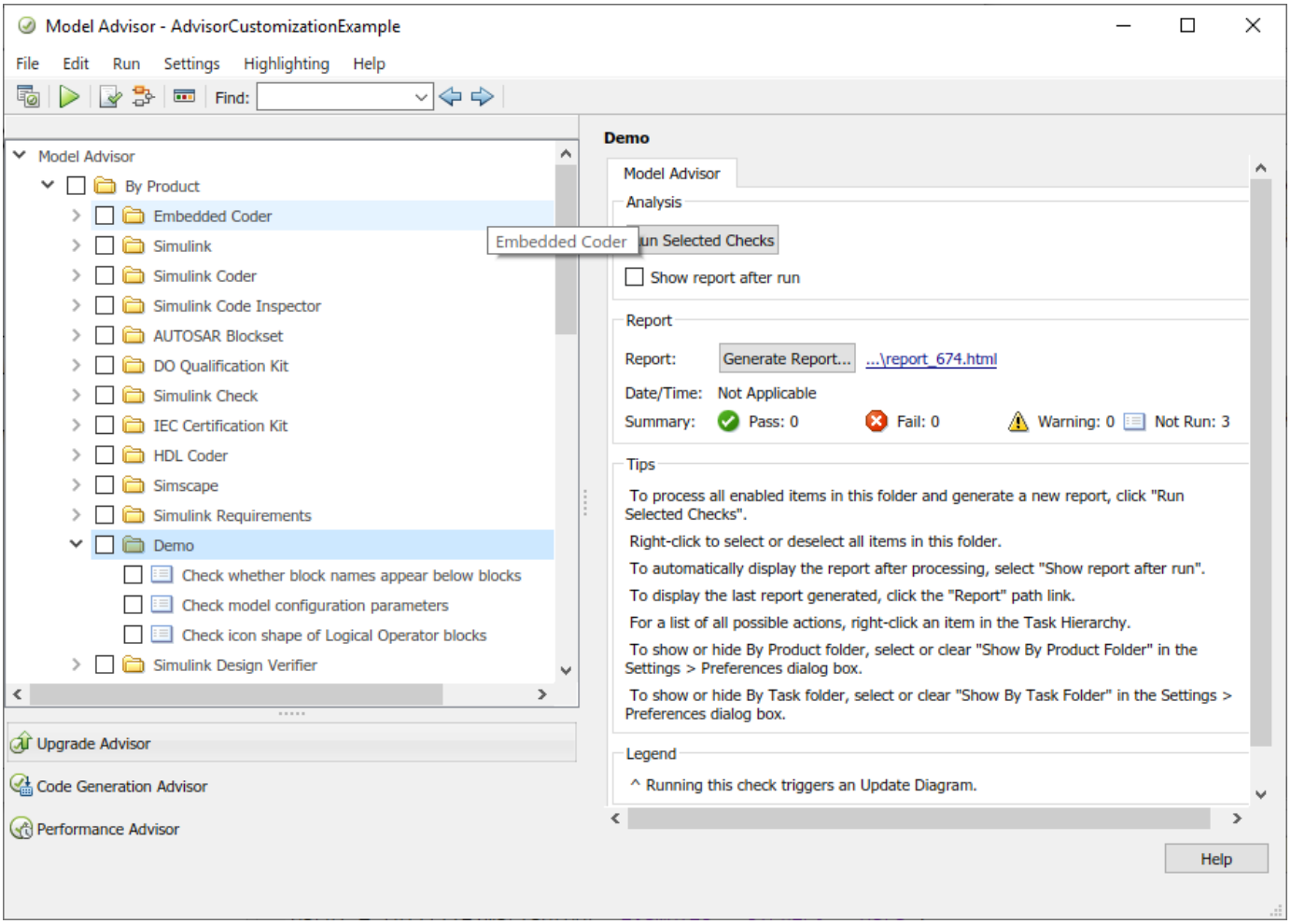


Load a Model Advisor Configuration

In this example, you will load the Model Advisor configuration file that was shipped with example `AdvisorCustomizationExample.slx`. The Model Advisor configuration file can be in `.json` (as of R2020a) or `.mat` (R2019b and earlier) format.

1. To load a custom configuration file, select **Settings > Load Configuration** and select the configuration you would like to use. For this example, select `demoConfiguration.json` from the writeable directory you specified previously. The Model Advisor updates to reflect the new configuration and displays the configuration file name at the top of the window.

2. Explore the Model Advisor and observe that that only three checks are available in this configuration.



3. To specify configuration file `demoConfiguration.json` as the default Model Advisor configuration, select **Settings > Set Current Configuration as Default**.

To clear the default configuration setting, select **Settings > Clear default configuration settings**. When you do not specify a default configuration, the Model Advisor uses the standard configuration that is defined by your system administrator.

Create a New Model Advisor Configuration

In this example, you will create a custom configuration file named `demo_Configuration_ISO26262_MAB.json`. You will use this configuration file to define the MathWorks Advisory Board (MAB) modeling guidelines checks that you want to execute by using the Model Advisor user interface.

1. In model `AdvisorCustomizationExample.slx`, open the Model Advisor and check that `demoConfiguration.json` is the current configuration.

2. Select **Settings > Open configuration editor** to open the Model Advisor Configuration Editor.

Note: The file that is currently loaded in the Model Advisor cache automatically displays when you open the Model Advisor Configuration Editor.

3. Click **New**. In the **Model Advisor** pane, select `All checks`. This specifies that the configuration you are viewing is for all of the checks in the Model Advisor user interface.

4. Highlight the Model Advisor Configuration Editor root folder and click **New Folder**. Select the new folder and in the **Information** tab, change the display name to `ISO 26262 Standard Checks`. Click **Apply**.

5. In the toolstrip, select **Show Library** to display the **Library** pane. In the **By Product** tab search field, enter **ISO 26262**.

6. Right-click on the `Simulink Check > Modeling Standards > IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks` folder and select **Copy**. Right-click on your custom ISO 26262 Standard Checks folder and click **Paste**. Verify that the folder and checks have been copied to your custom configuration folder.

Note: Checks that are copied from the **Library** pane retain their default parameter settings. When they are pasted into your custom configuration folder, the box beside each check is not selected.

7. In the **By Task** folder on the **Model Advisor** pane, right-click on the `Modeling Standards for MAB folder` and select **Cut**. Click on the Model Advisor Configuration Editor root folder and click **Paste**. The folder is removed from the **By Task** folder and is added as a new subfolder in the Model Advisor Configuration Editor root folder.

Note: Checks that are copied or cut from a folder in the **Model Advisor** pane retain their user-defined parameter settings. When a check is included in multiple folders, you can specify different parameters for each check individually.

8. Select the `ISO 26262 Standard Checks` folder and use the **Move Down** button to change the position this folder in the hierarchy. The Model Advisor will execute the checks in the `Modeling Standards for MAB` folder first.

9. Select the **By Product** and **By Task** folders and select **Delete**.

10. Click **Save As** and rename the configuration file to `demo_Configuration_ISO26262_MAB`. Select **Yes** at the prompt to save the configuration as the default configuration. The file automatically saves in `.json` format.

Note: If you have previously designated a default configuration, you can use **Clear default configuration** setting to clear the flag that specifies the current default configuration file. Clicking the button does not modify the configuration that is currently displayed in the Model Advisor Configuration Editor.

11. Close the Model Advisor Configuration Editor and the `AdvisorCustomizationExample.slx` model.

12. Refresh the Model Advisor cache and open model `AdvisorCustomizationExample.slx` by entering the following in the MATLAB command line:

```
Advisor.Manager.refresh_customizations();
open_system('AdvisorCustomizationExample.slx');
```

13. Open the Model Advisor and check that configuration file `demo_Configuration_ISO26262_MAB.json` is being used. The Model Advisor displays the folders `Modeling Standards for MAB` and `ISO 26262 Standard Checks`.

Specify Model Advisor Checks

In this example, you will customize the checks in your custom configuration file, `demo_Configuration_ISO26262_MAB.json`, and review the effect that your customizations have on the Model Advisor analysis of the `AdvisorCustomizationExample.slx` model.

1. On the **Modeling** tab, click **Model Advisor > Model Advisor Configuration Editor** and, if necessary, open `demo_Configuration_ISO26262_MAB.json`.

2. In the **Model Advisor** pane, verify that the configuration focus is set to the `All checks` option.

3. Clear the check box the box beside the **Model Advisor Configuration Editor** root node folder. (This step is optional. However, deselecting the checks allows you to more easily view the results of using the Model Advisor Configuration Editor to specify checks for display in the Model Advisor.)

4. To allow you to use the Model Advisor user interface to specify which checks to include in the Model Advisor analysis, right-click on the **Model Advisor Configuration Editor** root folder and click **Enable**. (Note: Enable is the default setting. This option is dimmed when none of the checks are disabled.)

5. Check the box beside the **Modeling Standards for MAB > Naming Conventions >Content > Check character usage in block name** check.

6. Right-click on each of these checks and select **Disable**:

- **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names**
- **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem name**

7. Click the **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check and, in the **Information** tab, select `Fail` for the **Check result when issues are flagged** option. Click **Apply**.

Note: The default for the **Check result when issues are flagged** option is `Warning`.

8. Click **Save** to save the configuration. Close the Model Advisor Configuration Editor and the model.

9. Refresh the Model Advisor cache and open model `AdvisorCustomizationExample.slx` by entering the following in the MATLAB command line:

```
Advisor.Manager.refresh_customizations();
open_system('AdvisorCustomizationExample.slx');
```

10. Open the Model Advisor and verify that configuration file `demo_Configuration_ISO26262_MAB.json` is being used.

Observe these checks, which reflect the settings that you chose in the Model Advisor Configuration Editor:

- The **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names check** is dimmed and the check box is selected. This check will always execute in a Model Advisor analysis and, because it is dimmed, you cannot choose to exclude it from the analysis.

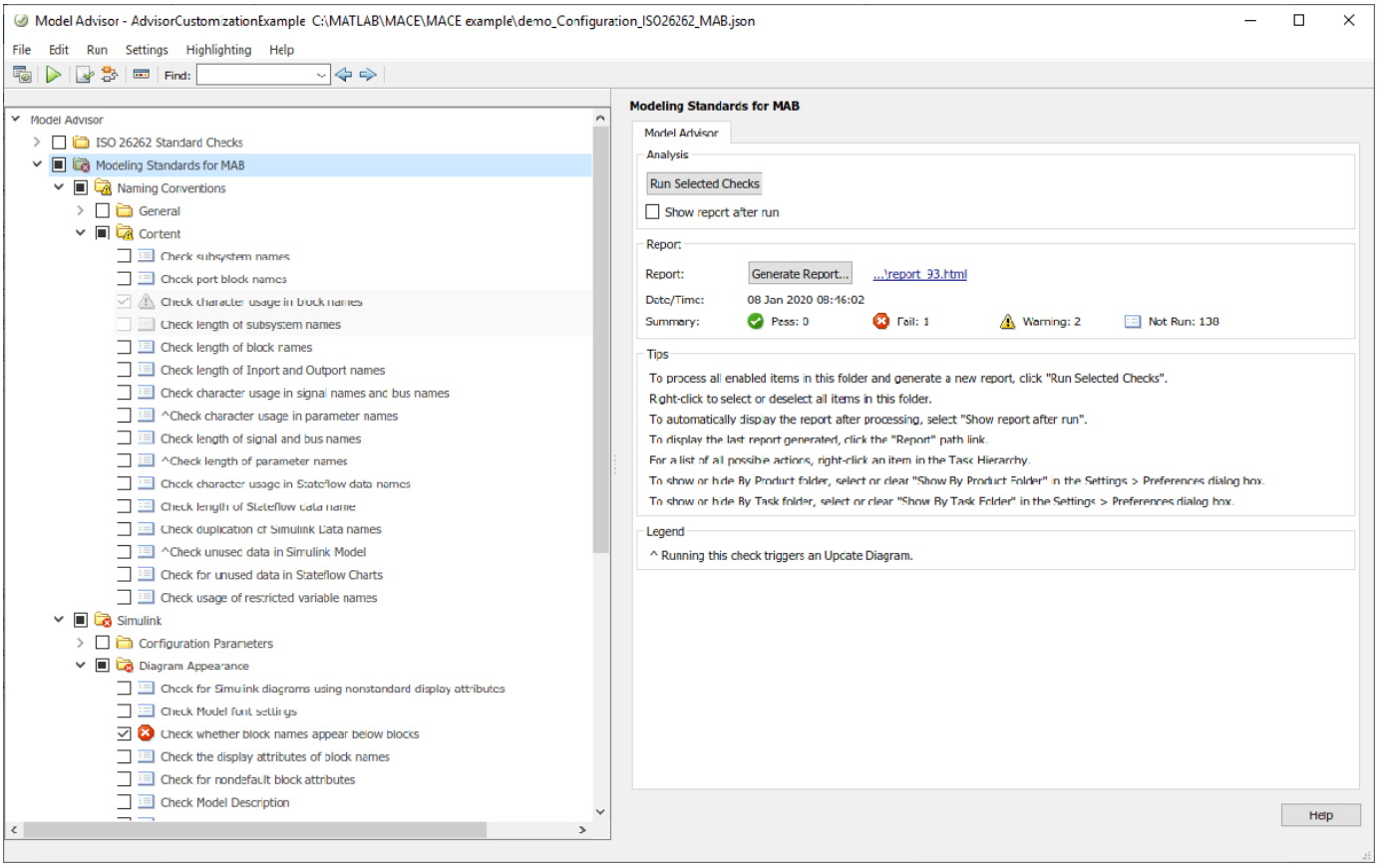
- The **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem names check** is dimmed and the check box is not selected. This check will not be included in the analysis and, because it is dimmed, you cannot select it for inclusion in the analysis.

11. Check the box beside the **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check.

12. To run the Model Advisor analysis, right-click on the **Model Advisor Standards for MAB** root node and select **Run Selected Checks**.

13. Click on the following checks and review the Model Advisor analysis results:

- The **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names** check is marked with a warning icon and the results specify the check violation is in the Gain block.
- There are no results for the **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem** check because it could not be selected for the analysis.
- The **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check is marked with a fail icon. This behavior is intended; you specified this check settings by using the Model Advisor Configuration Editor.



Specify Checks for Edit-Time Checking

In this example, you will specify the checks that are available in the `demo_Configuration_ISO26262_MAB.json` configuration file for execution during edit-time checking.

1. In model `AdvisorCustomizationExample.slx`, on the **Modeling** tab, make sure that the **Model Advisor > Edit-Time Checks** option is selected.

2. On the **Modeling** tab, click **Customize Edit-Time Checks**. The Model Advisor Configuration Editor opens with the configuration focus option set to `Edit-Time supported checks`. Only a subset of Model Advisor checks are supported for edit-time checking; the `Edit-Time supported checks` option displays only checks that support edit-time checking.

3. In the **Model Advisor** pane, check the box beside the **Modeling Standards for MAB > Simulink > Operation Blocks > Check icon shape of Logical Operator blocks** check.

4. Click **Save** to save the configuration. Close the Model Advisor Configuration Editor. Return to the `AdvisorCustomizationExample.slx` model.

5. In the Simulink editor, notice that the **Logical Operator** block is now highlighted. Open the violation and click **Fix**. The Model Advisor changes the block to a rectangular shape and the highlighted edit-time check warning clears.

See Also

`ModelAdvisor.Check` | `ModelAdvisor.setDefaultConfiguration`

More About

- “Defining Custom Model Advisor Checks Workflow” on page 6-34
- “Customize the Configuration of the Model Advisor Overview” on page 7-2
- “Update the Environment to Include Your Custom Configuration” on page 7-22
- “Load and Verify a Custom Configuration” on page 7-23

Programmatically Customize Tasks and Folders for the Model Advisor

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.

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.

Function	Description	Required or Optional
`sl_customization()`	Registers custom checks and tasks, folders with the Simulink customization manager at startup. See “Defining Custom Model Advisor Checks Workflow” on page 6-34.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Defining Custom Model Advisor Checks Workflow” on page 6-34.	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 7-16.	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 7-16.	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.

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, and folders. 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:

Note If you add custom checks within the `sl_customization.m` file, include methods for registering the checks 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 hierarcy, navigate to the folder that contains the MathWorks check.
2. In the right pane of the Model Advisor, select the **Source** tab. The Model Advisor displays the **Title**, **TitleID**, and **Source** information for each check in the folder.

   Note If the **Source** tab is not available, open **Settings** > **Preferences** and select **Show Source Tab**
3. 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.

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.

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.

See Also

`ModelAdvisor.Check` | `ModelAdvisor.FactoryGroup` | `ModelAdvisor.Group` | `ModelAdvisor.Procedure` | `ModelAdvisor.Task` | `publish`

More About

- “Defining Custom Model Advisor Checks Workflow” on page 6-34
- “Display and Enable Checks” on page 6-38
- “Customize the Configuration of the Model Advisor Overview” on page 7-2

Programmatically 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 “Customize the Configuration of the Model Advisor Overview” on page 7-2.
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. (Optional) Author custom checks in a customization file. See “Create Model Advisor Checks”.
5. Organize the checks into procedures for a procedural-based configuration:
   a. Create procedures by using the procedure API. For detailed information, see “Create Procedural-Based Configurations” on page 7-19.
   b. Create the custom configuration “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3

      .
6. (Optional) Deploy the custom configurations to your users. For detailed information, see “Deploy Custom Configurations” on page 7-24.
7. Verify that models comply with modeling guidelines. For detailed information, see “Run Model Advisor Checks and Review Results” on page 3-4.

Create Procedural-Based Configurations

Create Procedures Using the Procedures API

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.

Note When creating procedural checks, be aware of potential conflicts with the checks. Verify that it is possible to pass both checks.

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.

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.

The following code example adds subprocedures to a procedure:

```
%Create a procedure
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);
```

You can specify where the Model Advisor places a procedure using the `ModelAdvisor.Group.addProcedure` method. 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 procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAG);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

See Also

`ModelAdvisor.Check` | `ModelAdvisor.Procedure`

More About

- “Defining Custom Model Advisor Checks Workflow” on page 6-34
- “Customize the Configuration of the Model Advisor Overview” on page 7-2

Update the Environment to Include Your Custom Configuration

To make custom configuration available for use by the Model Advisor, you need to first update your Simulink environment to refresh the Model Advisor cache. This includes the creation of new or modifications to existing:

- `.json` files by using the Model Advisor Configuration Editor. See “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.
- `sl_customization.m` files for custom Model Advisor checks. See “Create Model Advisor Checks”.

To update your environment:

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 (Simulink).
2. In the MATLAB command line, enter:

   ```
   Advisor.Manager.refresh_customizations
   ```
3. Open your model
4. In the **Modeling** tab, select **Model Advisor** to open the Model Advisor. If you have customized the configuration by using the Model Advisor Configuration Editor, load and verify the configuration as described in “Load and Verify a Custom Configuration” on page 7-23.

Load and Verify a Custom Configuration

You can load a custom configuration for use in the Model Advisor. Once you have verified that the custom configuration displays the folders and checks that were specified by using the Model Advisor Configuration Editor, you can set it so that the Model Advisor use that configuration every time you open the Model Advisor.

Before you load the configuration, use `Advisor.Manager.refresh_customizations` to update your Simulink environment. See “Update the Environment to Include Your Custom Configuration” on page 7-22

1. In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.
2. Select **Settings > Load Configuration**. If you see a warning that the Model Advisor report corresponds to a different configuration, click **Load** to continue.
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; observe that the name of the configuration file is identified at the top of the window.

   Note The Model Advisor automatically warns you of checks that are missing when loading a Model Advisor configuration. To suppress this warning, see “Suppress Warning Message for Missing Checks” on page 7-7.
5. Browse the content of the Model Advisor to verify that folders and checks specified in the custom configuration file are displayed. If the correct configuration file is not being used, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3 and “Update the Environment to Include Your Custom Configuration” on page 7-22 .
6. To instruct the Model Advisor to always use this configuration, select **Settings > Set Current Configuration as Default**.

You can also use the `ModelAdvisor.setDefaultConfiguration` function to programmatically specify the default configuration used by the Model Advisor.

Deploy 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. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

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 Model Advisor checks	Customization file	• `sl_customization.m` • Files containing check and action callback functions (if separate)
Created custom Model Advisor configuration files	Model Advisor Configuration Editor	Configuration `.json` file

1. Distribute the files and tell the user to include these files on the MATLAB path.
2. Instruct the user to load the custom configuration.

Create and Deploy a Model Advisor Custom Configuration

To check that a model meets your company's standards and modeling guidelines, you can customize the Model Advisor. This example shows you how to add custom checks to the Model Advisor and remove shipping checks that you do not require. You can save a custom configuration and deploy it to others in your development group. Deploying a custom configuration allows your development group to review models using the same set of checks.

Define Custom Checks

This example defines these three types of custom checks:

- A pass/fail check that groups results by blocks and subsystems and provides a fix action.
- A check that verifies model configuration parameter settings.
- A check that specifies a constraint for a block parameter setting and provides a fix action.

In this example, you will add the three checks to the Model Advisor and remove all shipping checks.

1. Set your current folder to a writeable directory.

2. Copy the script `prepare_cust_chk_code` to your current folder and run the script. The script copies the files necessary for this example to your current folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main','prepare_cust_chk_code.m'),...
    'prepare_cust_chk_code.m','f');
run('prepare_cust_chk_code.m');
```

3. One of the files, `sl_customization.m`, includes an `sl_customization` function that defines custom checks. Open and inspect the `sl_customization.m` file.

```
function sl_customization(cm)
% SL_CUSTOMIZATION - Model Advisor customization demonstration.

% Copyright 2019 The MathWorks, Inc.

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

% ----------------------------
% defines Model Advisor Checks
% ----------------------------
function defineModelAdvisorChecks
defineDetailStyleCheck;
defineConfigurationParameterCheck;
defineBlockConstraintCheck;
```

The `sl_customization` function accepts a customization manager object. The customization manager object includes the `addModelAdvisorCheckFcn` method for registering custom checks. The input to this method is a handle to a function (`defineModelAdvisorChecks`). This function contains calls to the three check definition functions that correspond to the three custom checks.

Pass/Fail Check with Fix Action

The `defineDetailStyleCheck.m` file contains the `defineDetailStyleCheck` definition function, which defines a check that lists blocks whose names are not displayed below the blocks. This check

provides a fix that moves those names below the blocks. The name of this check is **Check whether block names appear below blocks**. Open and inspect the `defineDetailStyleCheck.m` file.

```
function defineDetailStyleCheck

mdladvRoot = ModelAdvisor.Root;

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
% Create ModelAdvisor.Action object for setting fix operation.
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
rec.setAction(myAction);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

end

% -----------------------------
% This callback function uses the DetailStyle CallbackStyle type.
% -----------------------------
function DetailStyleCallback(system, CheckObj)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

% Find all blocks whose name does not appear below blocks
violationBlks = find_system(system, 'Type','block',...
    'NamePlacement','alternate',...
    'ShowName', 'on');
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the bloc
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
else
    ElementResults(1,numel(violationBlks))=ModelAdvisor.ResultDetail;
    for i=1:numel(ElementResults)
        ElementResults(i).setData(violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below t
        ElementResults(i).Status = 'The following blocks have names that do not display below the
        ElementResults(i).RecAction =  'Change the location such that the block name is below the
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end

% -----------------------------
% This action callback function changes the location of block names.
% -----------------------------
function result = ActionCB(taskobj)
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
```

```
resultDetailObjs = checkObj.ResultDetails;
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block,'NamePlacement','normal');
end

result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
mdladvObj.setActionEnable(false);
end
```

This check uses the `setCallbackFcn` type of `DetailStyle`. Applying this style produces default formatting, so that you do not have to use the `ModelAdvisor.FormatTemplate` or the other Model Advisor formatting APIs to format the results that appear in the Model Advisor. For more information on how to create this type of check definition function, see “Create a Pass/Fail Model Advisor Check with Fix Action” on page 6-9.

Model Configuration Parameter Settings Check

The `defineConfigurationParameterCheck.m` file contains the `defineConfigurationParameterCheck` check definition function, which defines a check that identifies model configuration parameter settings that might impact MISRA C:2012 compliant code generation. The name of this check is **Check model configuration parameters**.

This check type requires a supporting XML data file that contains the model configuration parameter settings that you want to check. This file must be on the MATLAB path. For this example, that file is `configurationParameterDataFile.xml`. For more information on how to create this check type, see “Create Model Advisor Check for Model Configuration Parameters” on page 6-16.

Open and inspect the `defineConfigurationParameterCheck.m` file.

```
function defineConfigurationParameterCheck

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.configurationParameter');
rec.Title = 'Check model configuration parameters';
rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec.TitleTips = 'Identify configuration parameters that might impact MISRA C:2012 compliant code

% --- data file input parameters
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Data File';
inputParam1.Value = 'configurationParameterDataFile.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);

% publish check into Demo group.
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec, 'Demo');

end
```

Block Parameter Constraint Check

The `defineBlockConstraintCheck.m` file contains the defineBlockConstraintCheck check definition function, which defines a check that identifies Logical Operator blocks that do not have a rectangular shape. The name of this check is **Check icon shape of Logical Operator blocks**.

This check type supports edit-time checking and requires a supporting XML file that contains the block constraint information. This XML file must be on the MATLAB path. For this example, that file name is `blockConstraintDataFile.xml`. For more information on this check type, see “Define Model Advisor Checks for Supported or Unsupported Blocks and Parameters” on page 6-27.

Open and inspect the `defineBlockConstraintCheck.m` file.

```
function defineBlockConstraintCheck

rec = Advisor.authoring.createBlockConstraintCheck('com.mathworks.sample.blockConstraint');
rec.Title = 'Check icon shape of Logical Operator blocks';
rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec.TitleTips = 'Checks icon shape of Logical Operator blocks. Icon shape of Logical Operator sho

% --- data file input parameters
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Data File';
inputParam1.Value = 'blockConstraintDataFile.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});
rec.SupportExclusion = false;
rec.SupportLibrary = true;

% publish check into Demo group.
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec, 'Demo');

end
```

View Custom Checks in the Model Advisor

1. In order for your custom checks to be visible in the Model Advisor, you must refresh the Model Advisor check information cache. At the MATLAB command prompt, type this command:

```
Advisor.Manager.refresh_customizations();
```

2. Open the example model.

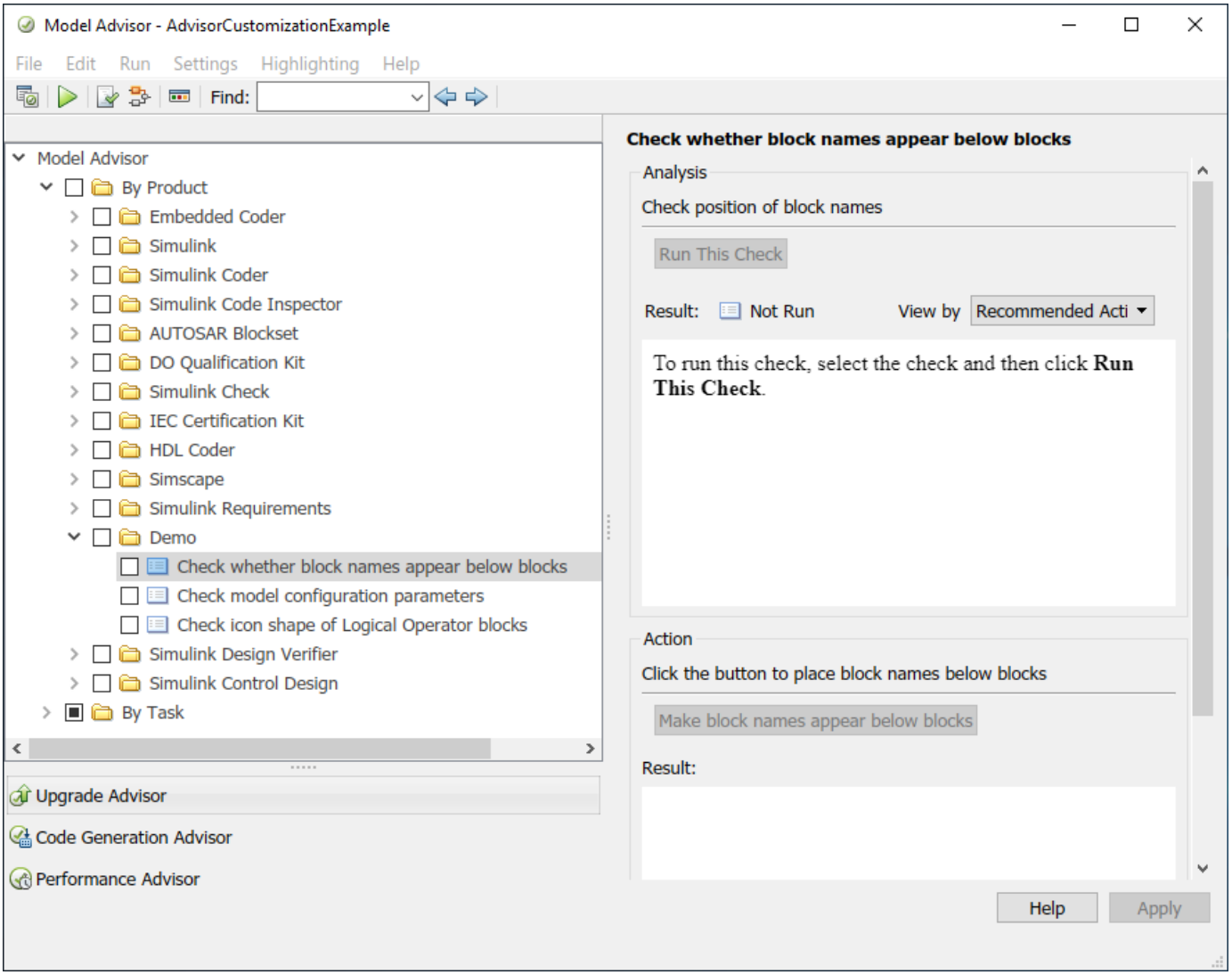
```
open_system('AdvisorCustomizationExample.slx');
```

3. On the **Modeling** tab, open the **Model Advisor**. You can also open the Model Advisor by typing this command at the MATLAB command prompt:

```
modeladvisor('AdvisorCustomizationExample.slx');
```

```
Updating Model Advisor cache...
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.Manager
```

4. Observe the custom checks in the **By Product > Demo** folder.



In the check definition functions, the `publish` command adds the checks to the **By Product > Demo** folder. Note, if you already have a default configuration set, the checks do not appear in the Model Advisor. Restore the default shipping configuration by selecting **Settings > Restore Default Configuration**.

Notice that the **Check whether block names appear below blocks** and the **Check model configuration parameters** checks contain an **Action** box with the option of fixing any check violations. In your custom check definition file, you supply the fix as part of the action callback

function. For more information on how to define custom checks, see “Defining Custom Model Advisor Checks Workflow” on page 6-34.

On the model canvas, observe that the Unit Delay and Logical Operator blocks are highlighted in yellow because edit-time checking is on. These blocks contain check violations. Edit-time checking allows you to interactively check and fix your model for compliance issues.

To fix these violations, hover over a highlighted block and click the yellow icon. A diagnostic box with a **Fix** button opens.

To turn edit-time checking on or off, on the **Modeling** tab, select **Model Advisor > Edit-Time Checks**.

Specify and Deploy a Model Advisor Custom Configuration

To specify which checks to include in the Model Advisor and which checks to use for edit-time checking, use the Model Advisor Configuration Editor.

1. To open the Configuration Editor, on the **Modeling** tab, select **Model Advisor > Model Advisor Configuration Editor**.

2. To add or remove checks and folders, select from the options in the **Edit** section of the Model Advisor Configuration Editor.

3. To save a configuration, select **Save**. A window opens and prompts you to save the configuration as a .json file. For this example, you do not have to save the configuration, as the corresponding `demoConfiguration.json` file ships with this example. You previously copied this file to your working folder.

4. Close the model and the Model Advisor Configuration Editor.

```
bdclose;
```

When you save a configuration, you get a prompt asking you whether you want to save that configuration as the default configuration. Setting a default configuration allows the Model Advisor to use that configuration every time you open the Model Advisor. For more information, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

Load and Run a Model Advisor Custom Configuration

1. Open the example model.

```
open_system('AdvisorCustomizationExample.slx');
```

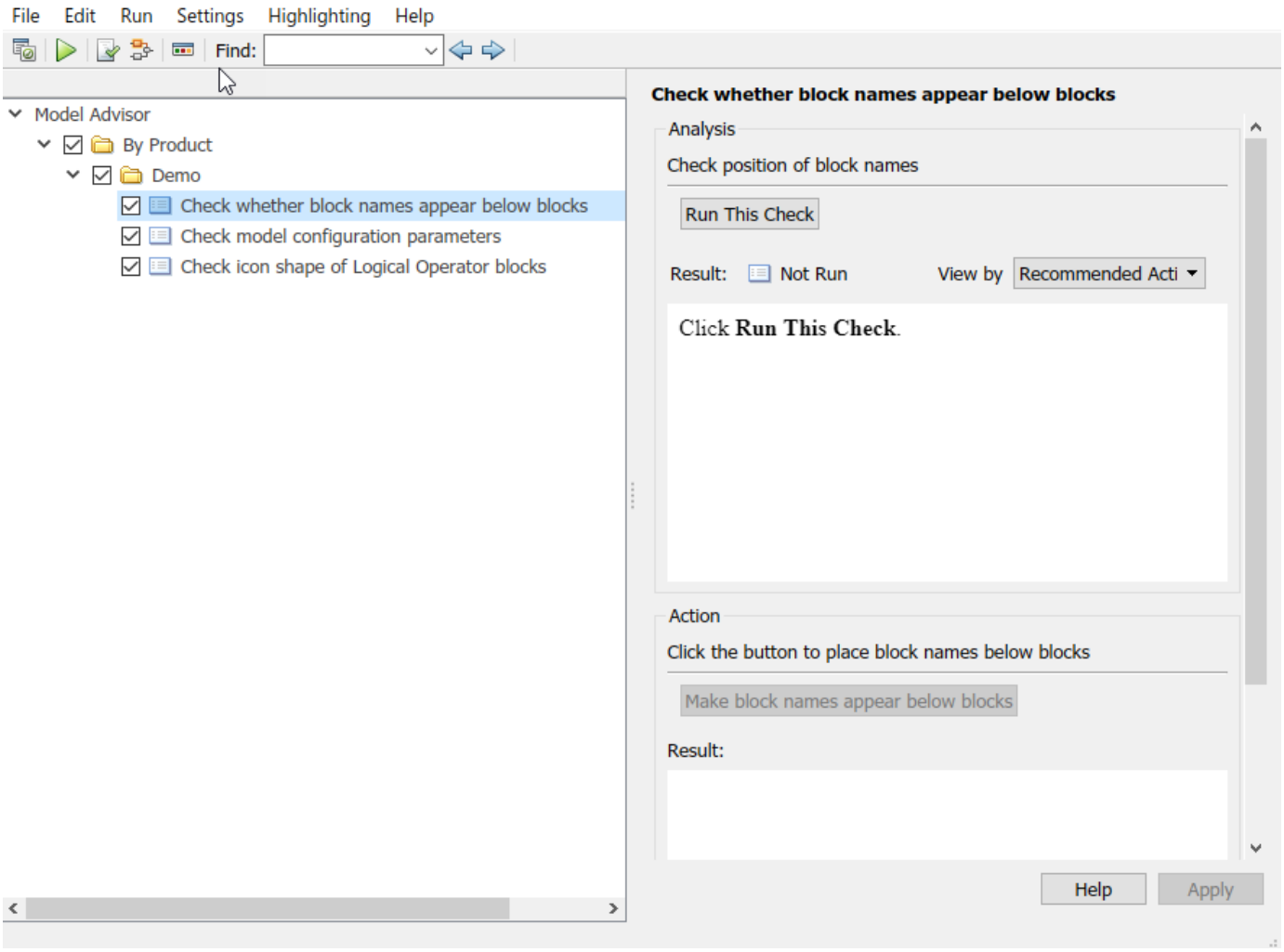
2. Open the Model Advisor. Load the custom configuration by selecting **Settings > Load Configuration**. In the **Open** dialog box, navigate to and select the `demoConfiguration.json` file. Or enter this command at the MATLAB command prompt:

```
modeladvisor('AdvisorCustomizationExample.slx', 'configuration',...
    'demoConfiguration.json');
```

```
Model Advisor is removing the existing report.
```

3. A Warning dialog box opens. Click **Remove existing report and continue**.

The Model Advisor contains only the **By Product > Demo** folder with the three custom checks.



3. Click the **Demo** folder and click **Run Selected Checks.**

4. Click the **Check whether block names appear below blocks** check. This check contains warnings. To apply a fix and resolve the warnings, on the right pane, click the **Make block names appear below blocks** button.

5. Click the **Check model configuration parameters** check. This check contains warnings. To apply a fix and resolve the warnings, click the **Modify Settings** button.

6. Click the **Check icon shape of Logical Operator blocks** check. This check contains a warning but no automatic fix. To apply a fix, follow the recommended action.

7. Close the model and the Model Advisor.

```
bdclose;
```

8. Remove the files from your working directory. Refresh the Model Advisor check information cache by typing this command at the command type:

```
Advisor.Manager.refresh_customizations
```

.

`ModelAdvisor.Check` | `ModelAdvisor.Procedure` | `Simulink.ModelAdvisor` | `addProcedure` | `addTask` | `register` | `setCheck`

More About

- “Define Custom Tasks” on page 7-16

Model Slicer

- “Highlight Functional Dependencies” on page 8-2
- “Highlight Dependencies for Multiple Instance Reference Models” on page 8-8
- “Refine Highlighted Model” on page 8-12
- “Refine Dead Logic for Dependency Analysis” on page 8-22
- “Create a Simplified Standalone Model” on page 8-28
- “Highlight Active Time Intervals by Using Activity-Based Time Slicing” on page 8-29
- “Simplify a Standalone Model by Inlining Content” on page 8-36
- “Workflow for Dependency Analysis” on page 8-38
- “Configure Model Highlight and Sliced Models” on page 8-40
- “Model Slicer Considerations and Limitations” on page 8-43
- “Using Model Slicer with Stateflow” on page 8-49
- “Isolating Dependencies of an Actuator Subsystem” on page 8-51
- “Isolate Model Components for Functional Testing” on page 8-55
- “Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results” on page 8-63
- “Simplification of Variant Systems” on page 8-65
- “Programmatically Resolve Unexpected Behavior in a Model with Model Slicer” on page 8-66
- “Programmatically Resolve Unexpected Behavior in a Model with Model Slicer” on page 8-88
- “Refine Highlighted Model Slice by Using Model Slicer Data Inspector” on page 8-99
- “Debug Slice Simulation by Using Fast Restart Mode” on page 8-106
- “Isolate Referenced Model for Functional Testing” on page 8-113
- “Analyze the Dead Logic” on page 8-117
- “Investigate Highlighted Model Slice by Using Model Slicer Data Inspector” on page 8-122

Highlight Functional Dependencies

Large models often contain many levels of hierarchy, complicated signals, and complex mode logic. You can use Model Slicer to understand which parts of your model are significant for a particular behavior. This example shows how to use Model Slicer to explore the behavior of the `sldvSliceClimateControlExample` model. You first select an area of interest, and then highlight the related blocks in the model. In this example, you trace the dependency paths upstream of `Out1` to highlight which portions of the model affect its behavior.

Open the model and highlight the functional dependencies of a signal in the system:

1. Add the example folder to the search path.

   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   ```

2. Open the `sldvSliceClimateControlExample` model.

   ```
   sldvSliceClimateControlExample
   ```

3. To open the Model Slice Manager, on the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

   When you open the Model Slice Manager, Model Slicer compiles the model. You then configure the model slice properties.

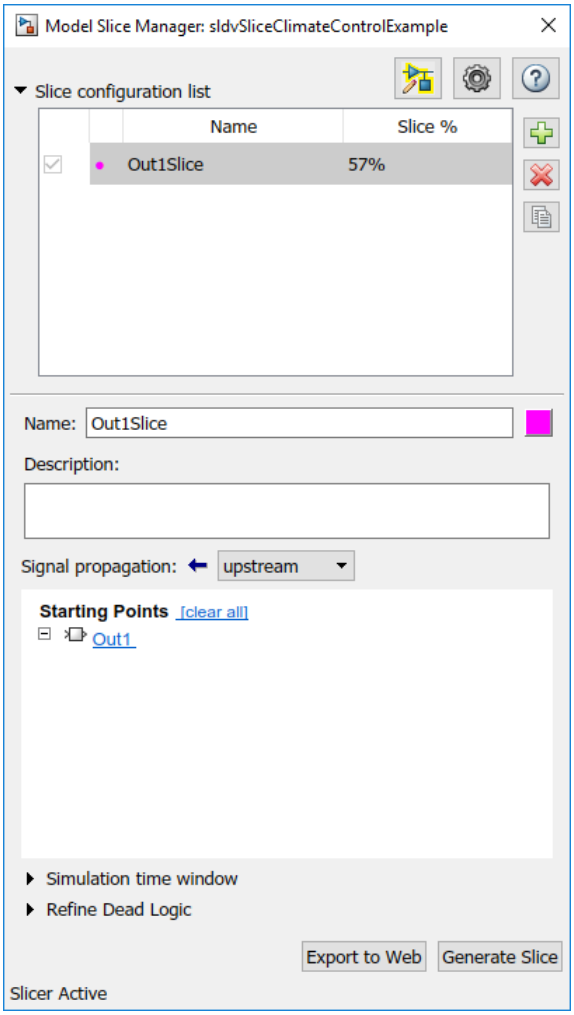
4. In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.

5. Set the slice properties:

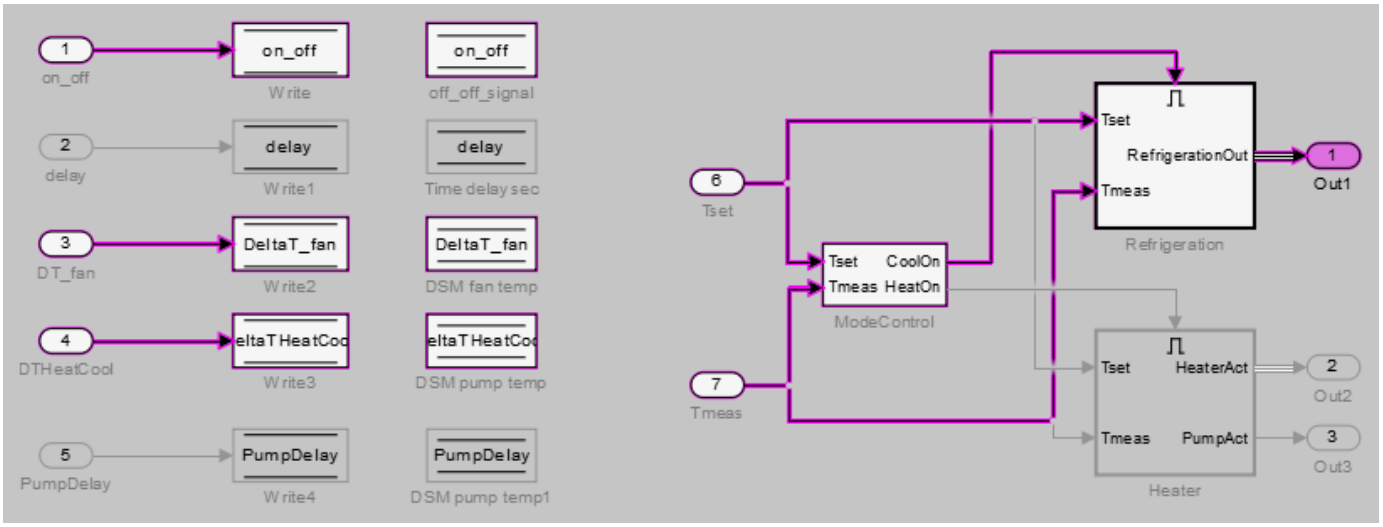
   - **Name**: `Out1Slice`
   - **Color**: (magenta)
   - **Signal Propagation**: `upstream`

   Model Slicer can also highlight the constructs downstream of or bidirectionally from a block in your model, depending on which direction you want to trace the signal propagation.

6. Add `Out1` as a starting point. In the model, right-click `Out1` and select **Model Slicer > Add as Starting Point**.



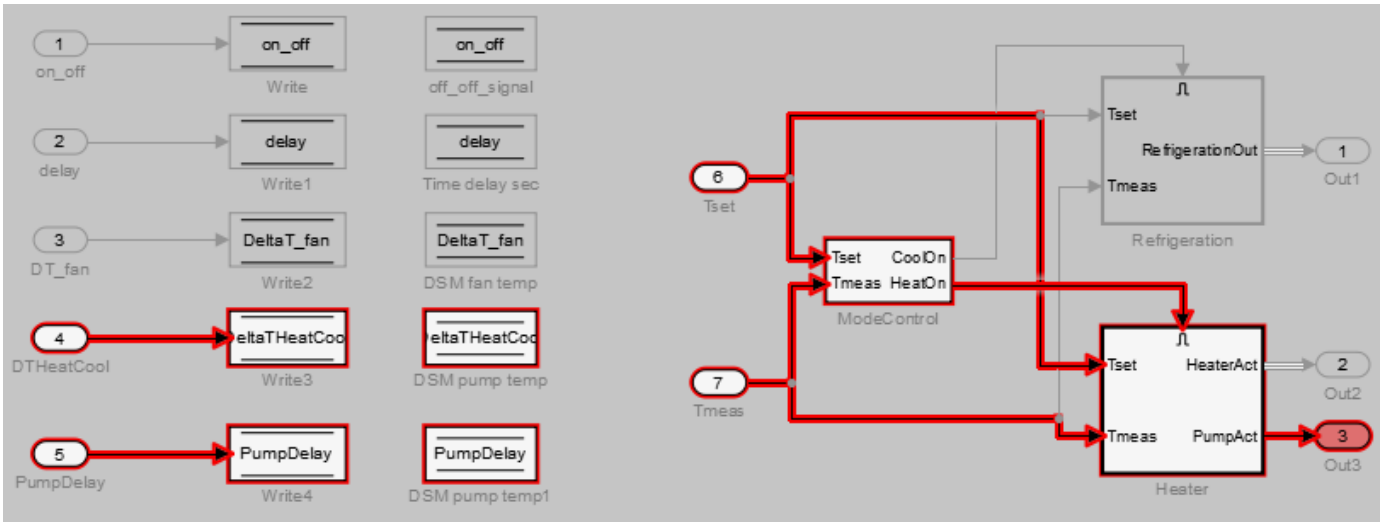
The Model Slicer now highlights the upstream constructs that affect `Out1`.



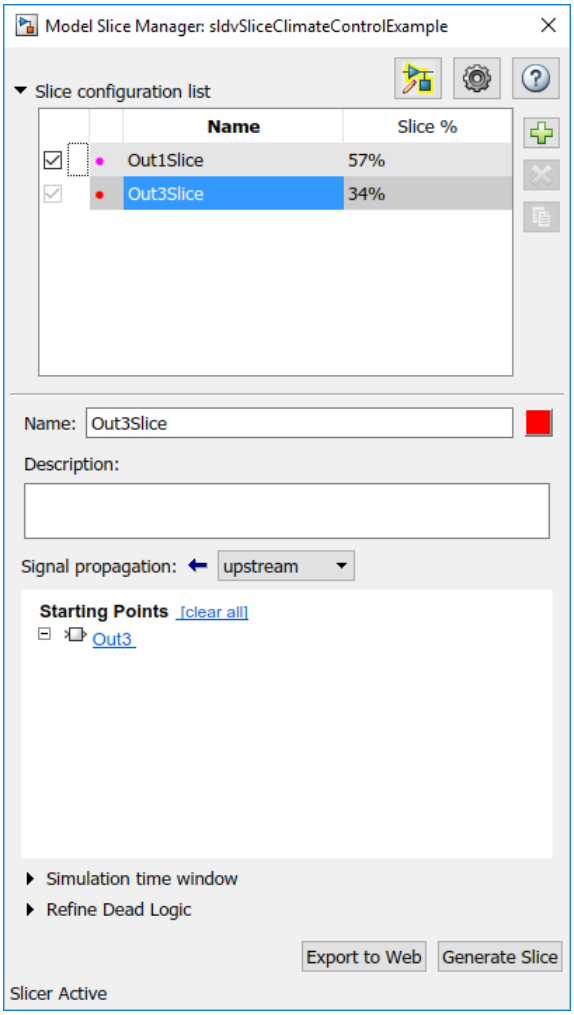
If you create two slice configurations, you can highlight the intersecting portions of their highlights. Create a new slice configuration and view the intersecting portions of the slice configuration you created above and the new slice configuration:

1 Create a new slice configuration with the following properties

- **Name**: Out3Slice
- **Color**: (red)
- **Signal Propagation**: `upstream`
- **Starting point**: `Out3`

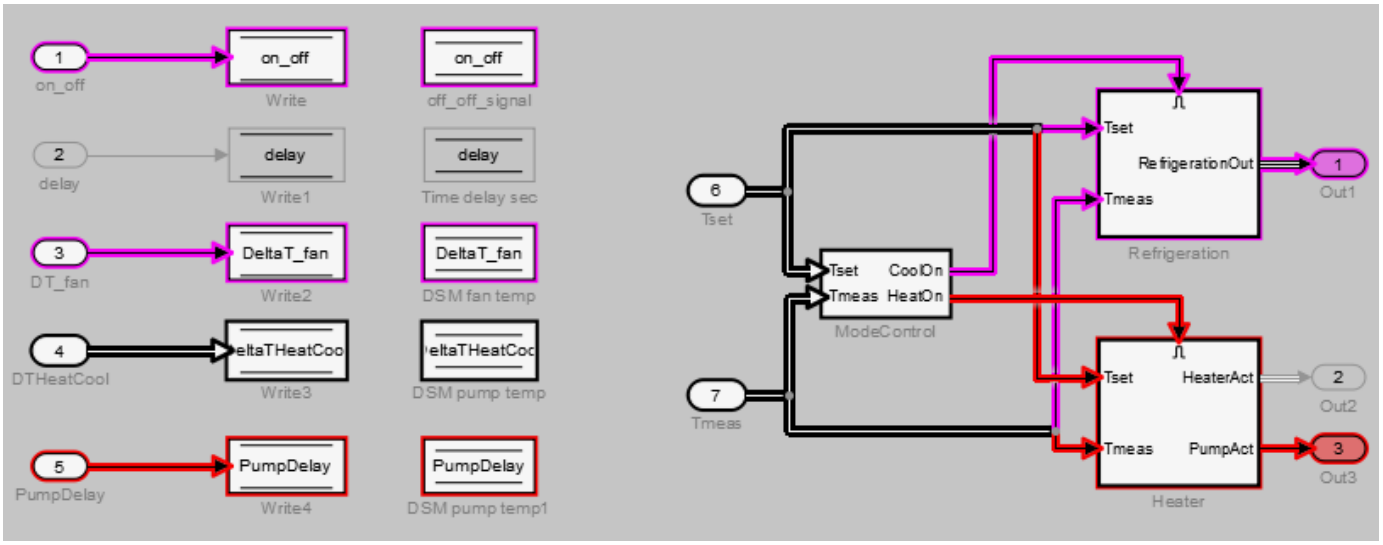


2 In the Model Slice Manager, select both the `Out1Slice` slice configuration and the `Out3Slice` slice configuration.



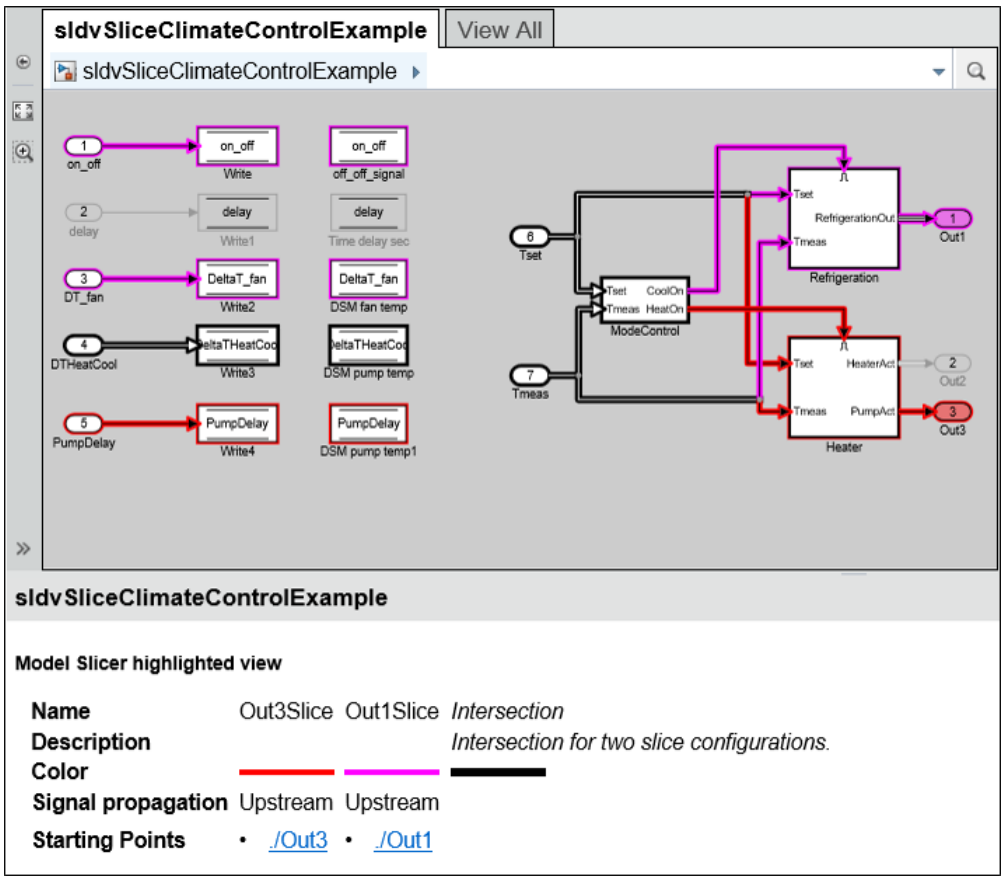
Model Slicer highlights portions of the model as follows:

- The portions of the model that are exclusively upstream of `Out1` are highlighted in cyan.
- The portions of the model that are exclusively upstream of `Out3` are highlighted in red.
- The portions of the model that are upstream of both `Out1` and `Out3` are highlighted in black.



After you highlight a portion of your model, you can then refine the highlighted model to an area of interest. Or, you can create a simplified standalone model containing only the highlighted portion of your model.

To view the details of the highlighted model in web view, click **Export to Web**. The web view HTML file is stored in `<current folder>\<model_name>\webview.html`.



See Also

More About

- “Refine Highlighted Model” on page 8-12
- “Create a Simplified Standalone Model” on page 8-28
- “Model Slicer Considerations and Limitations” on page 8-43

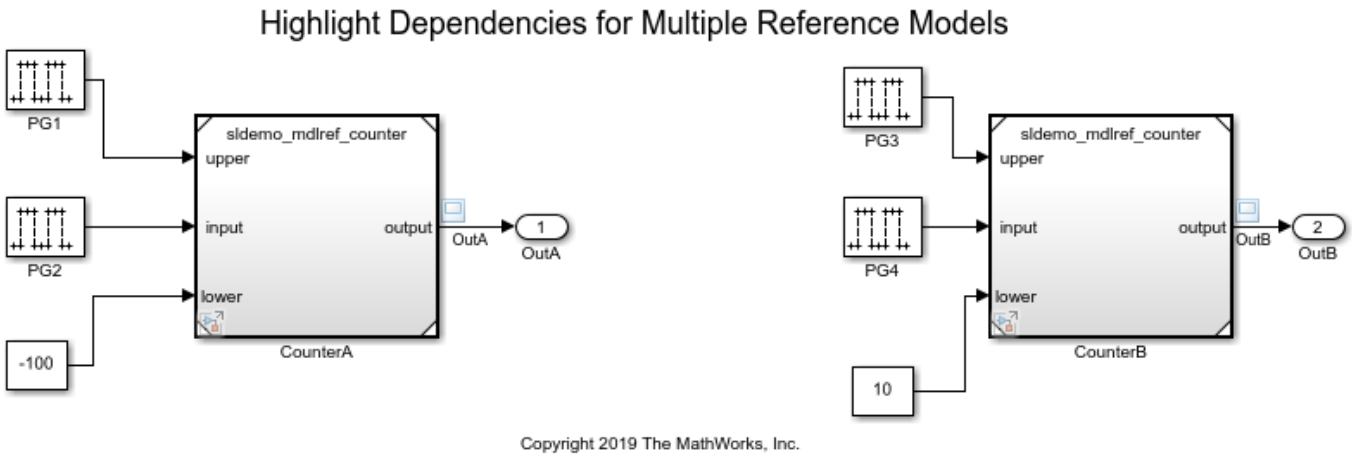
Highlight Dependencies for Multiple Instance Reference Models

To highlight the functional dependencies in a Simulink model with multiple instances of a referenced model, use Model Slicer. You can use Model Slicer on a Simulink model that contains single or multiple references to a same model in normal simulation mode.

This example shows the behaviour of Model Slicer when there are multiple instances of the referenced model. The `slslicerdemo_multi_instance` model consists of `sldemo_mdlref_counter` referenced two times with different inputs during the course of the signal flow transition.

1. Open the model **slslicerdemo_multi_instance.slx**.

```
open_system('slslicerdemo_multi_instance');
```

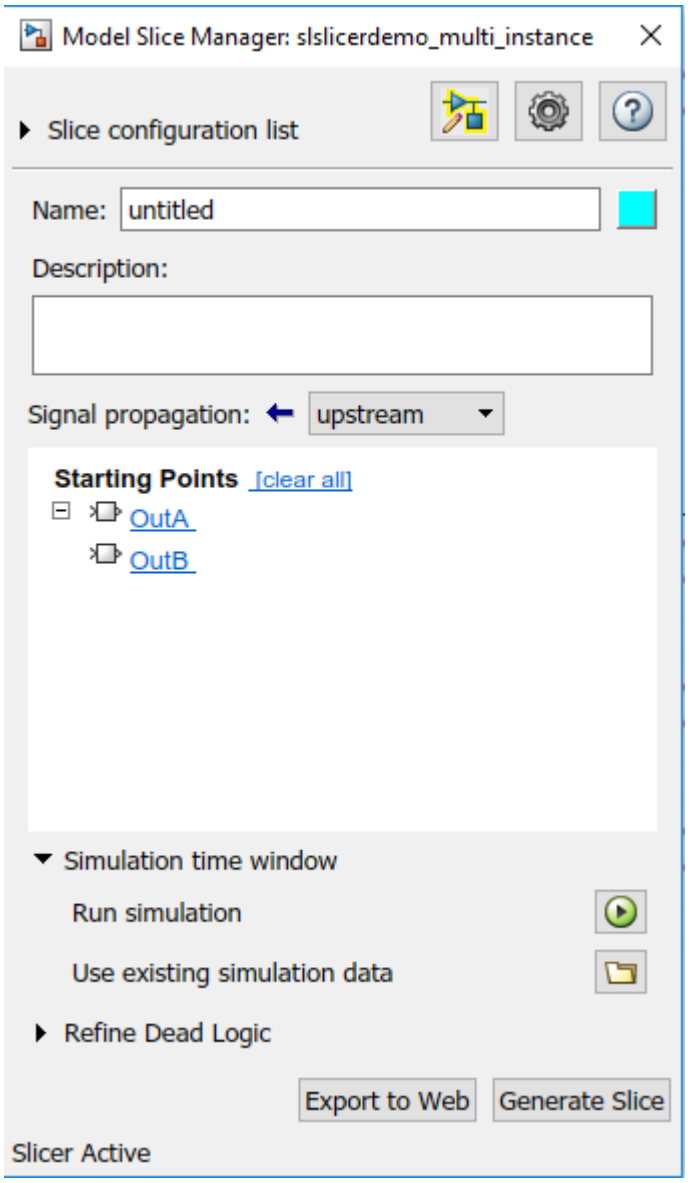


2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

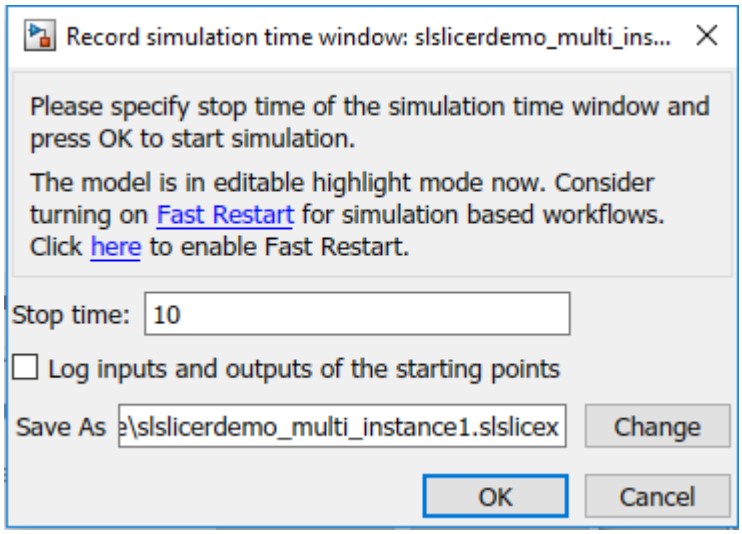
3. In the Model Slicer window, click **Add all outports**. This sets **OutA** and **OutB** as starting points.

4. Ensure that the **Signal Propagation** is set to **upstream**.

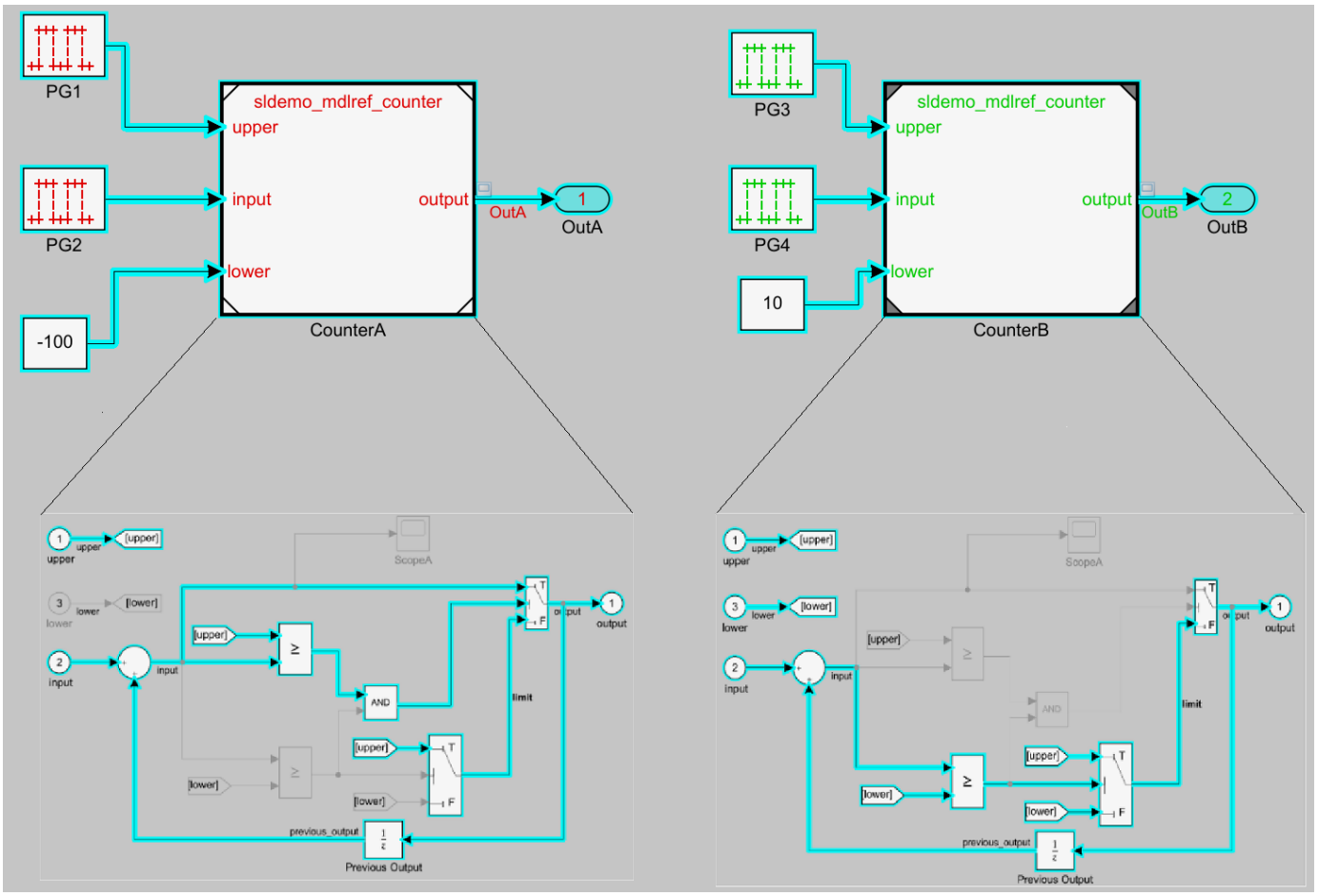
5. In the **Simulation time window** section, click **Run simulation**.



6. In the simulation time window, click **OK**. The model simulation starts.



7. The simulated model highlights the upstream dependency of the outports **OutA** and **OutB**.



You can notice that the referenced model in both the instances shows different signal propagations highlighted by Simulink Slicer for which the signal travels.

8. To generate the slice, click **Generate Slice**.

More About

- “Highlight Functional Dependencies” on page 8-2
- “Model Slicer Considerations and Limitations” on page 8-43

Refine Highlighted Model

After you highlight a model using Model Slicer, you can refine the dependency paths in the highlighted portion of the model. Using Model Slicer, you can refine a highlighted model by including only those blocks used in a portion of a simulation time window, or by excluding blocks or certain inputs of switch blocks. By refining the highlighted portion of your model, you can include only the relevant parts of your model.

In this section...
“Define a Simulation Time Window” on page 8-12
“Exclude Blocks” on page 8-16
“Exclude Inputs of a Switch Block” on page 8-19

Define a Simulation Time Window

You can refine a highlighted model to include only those blocks used in a portion of a simulation time window. Defining the simulation time window holds some switch blocks constant, and as a result removes inactive inputs.

1. Add the example folder to the search path.

   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   ```

2. Open the `sldvSliceClimateControlExample` model.

   ```
   sldvSliceClimateControlExample
   ```

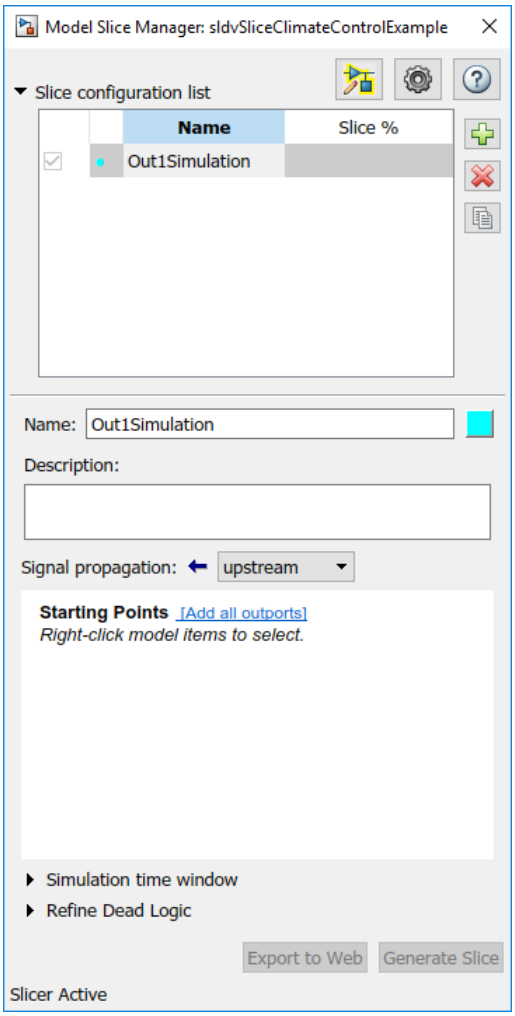
3. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

   When you open the Model Slice Manager, Model Slicer compiles the model. You then configure the model slice properties.

4. In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.

5. Set the slice properties:

   - **Name**: `Out1Simulation`
   - **Color**: (cyan)
   - **Signal propagation**: `upstream`



6 In the top level of the model, select the `Out1` block as the slice starting point. Right-click the `Out1` block and select **Model Slicer > Add as Starting Point**.

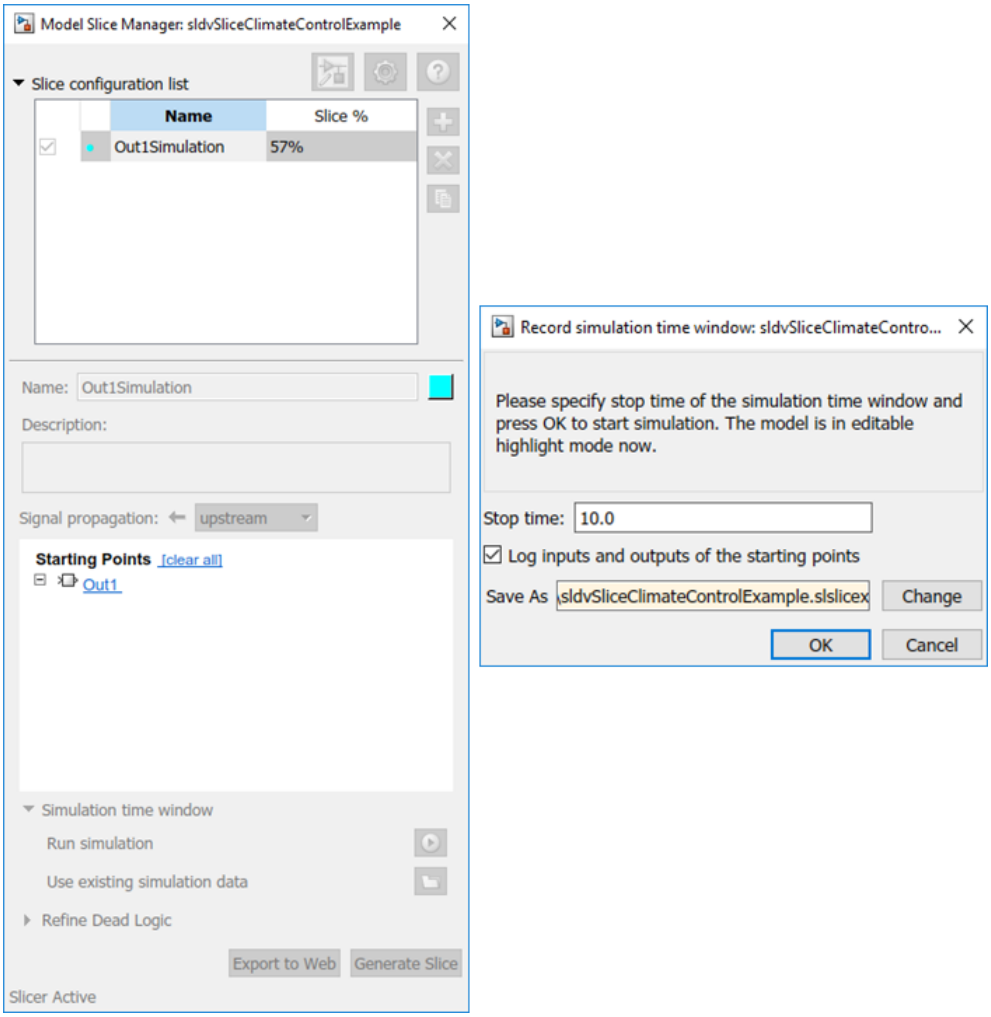
The model is highlighted.

7 In the Model Slice Manager, select **Simulation time window**.

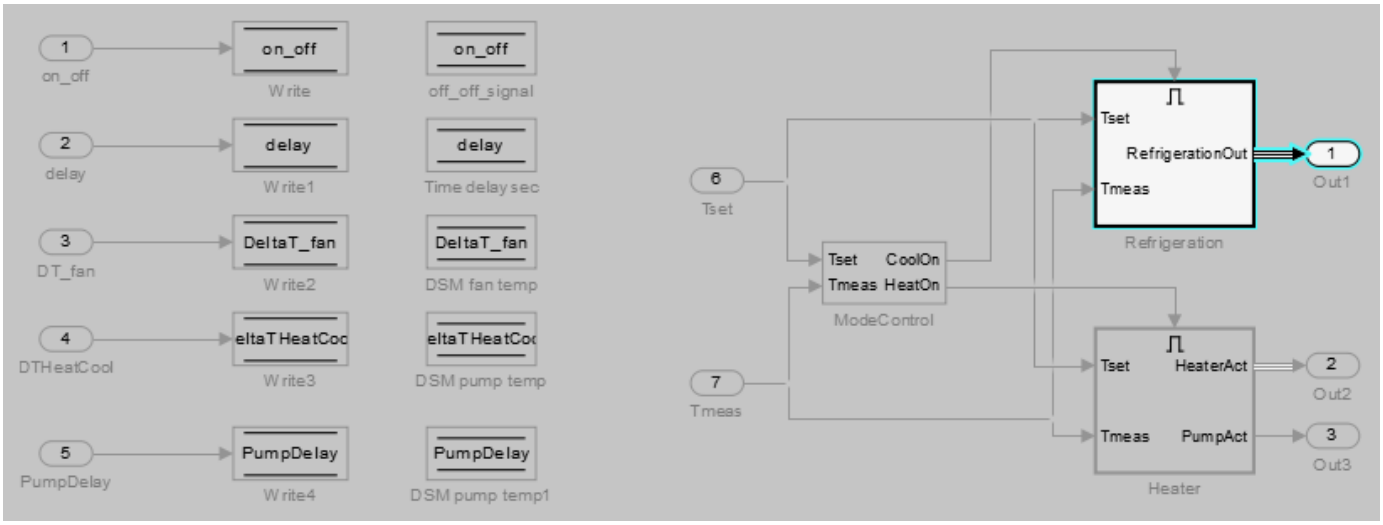
8 To specify the stop time of the simulation time window, click the run simulation button in the Model Slice Manager.

9 Set the **Stop time** to `10`.

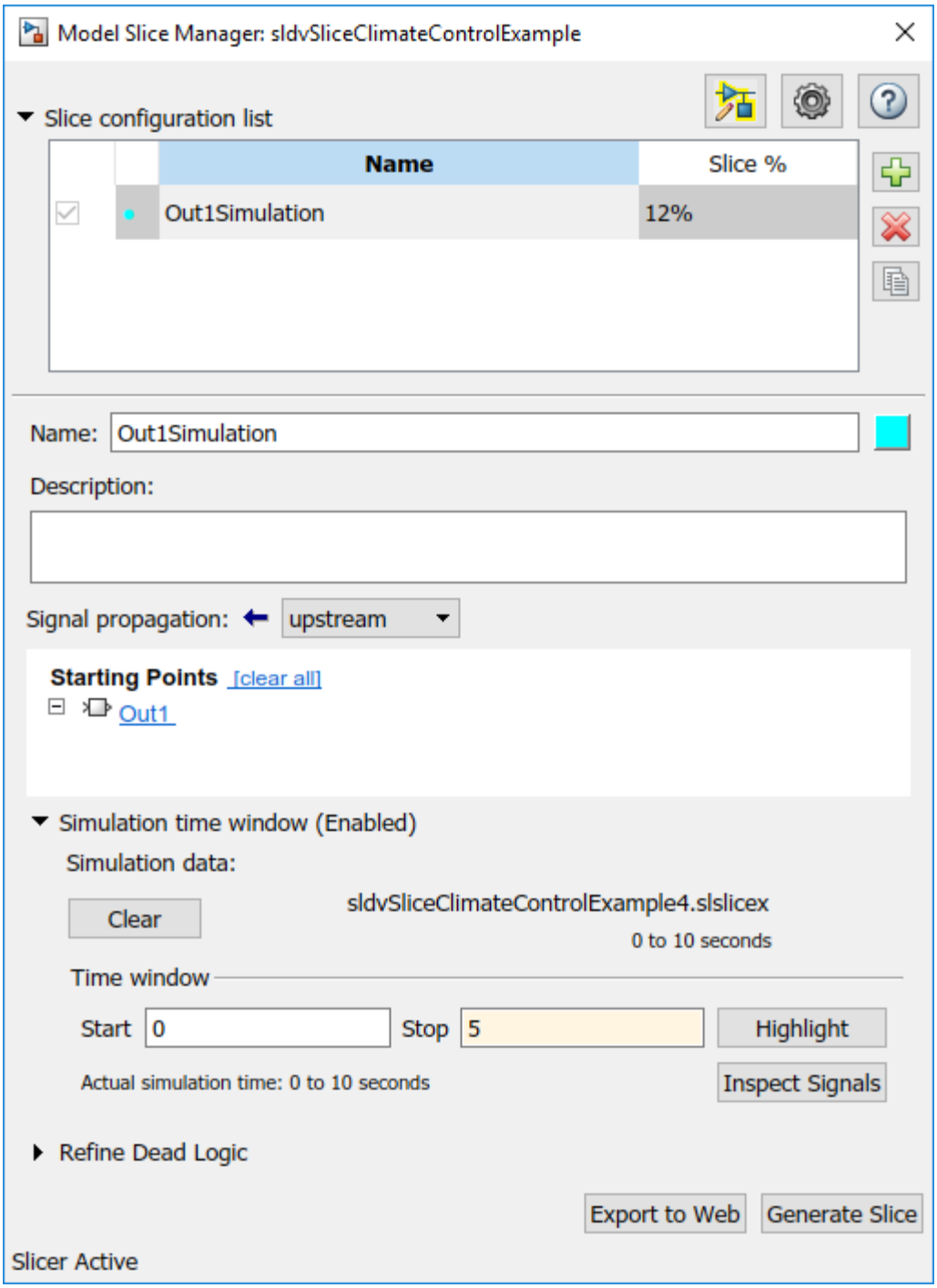
10 Click **OK** to start the simulation.



The path is restricted to only those blocks that are active until the stop time that you entered.

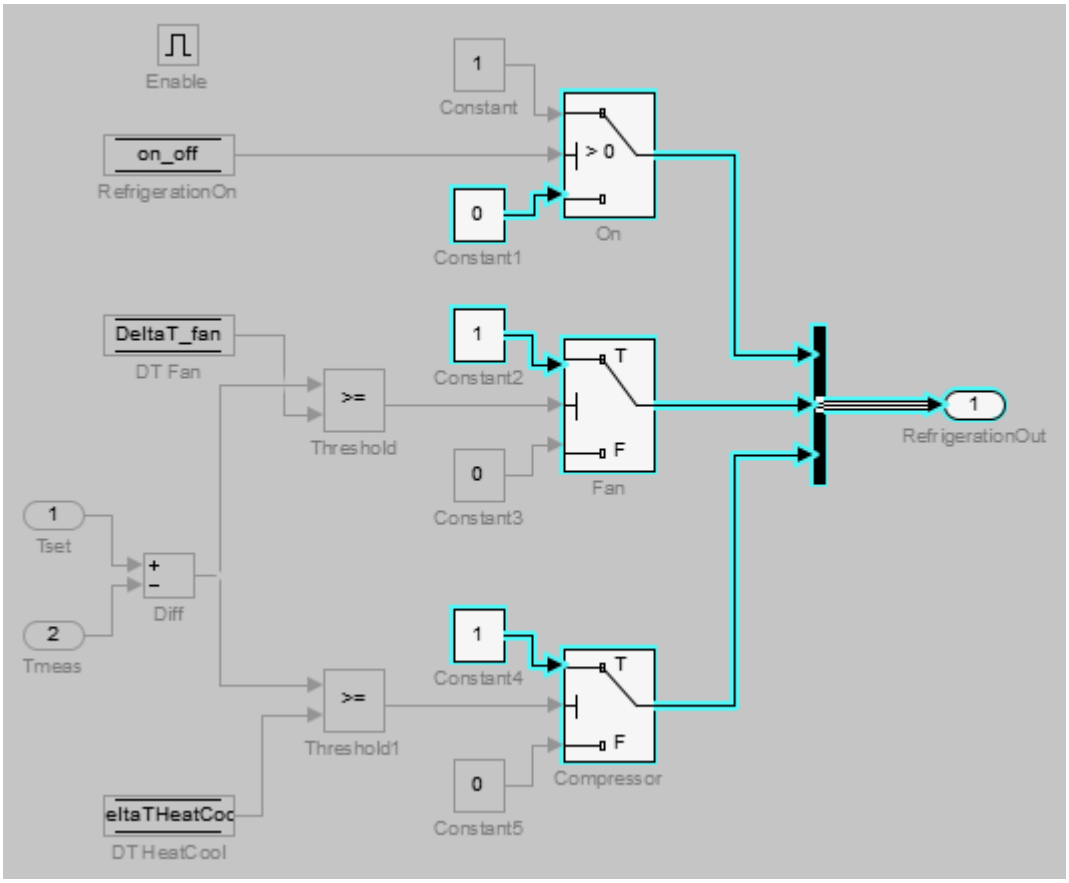


11 To highlight the model for a defined simulation time window, set the **Stop** time to 5. Click **Highlight**.



12 To see how this constraint affects the highlighted portion of the model, open the `Refrigeration` subsystem.

The highlighted portion of the model includes only the input ports of switches that are active in the simulation time window that you specified.



After you refine your highlighted model to include only those blocks used in a portion of a simulation time window, you can then “Create a Simplified Standalone Model” on page 8-28 incorporating the highlighted portion of your model.

Exclude Blocks

You can refine a highlighted model to exclude blocks from the analysis. Excluding a block halts the propagation of dependencies, so that signals and model items beyond the excluded block in the analysis direction are ignored.

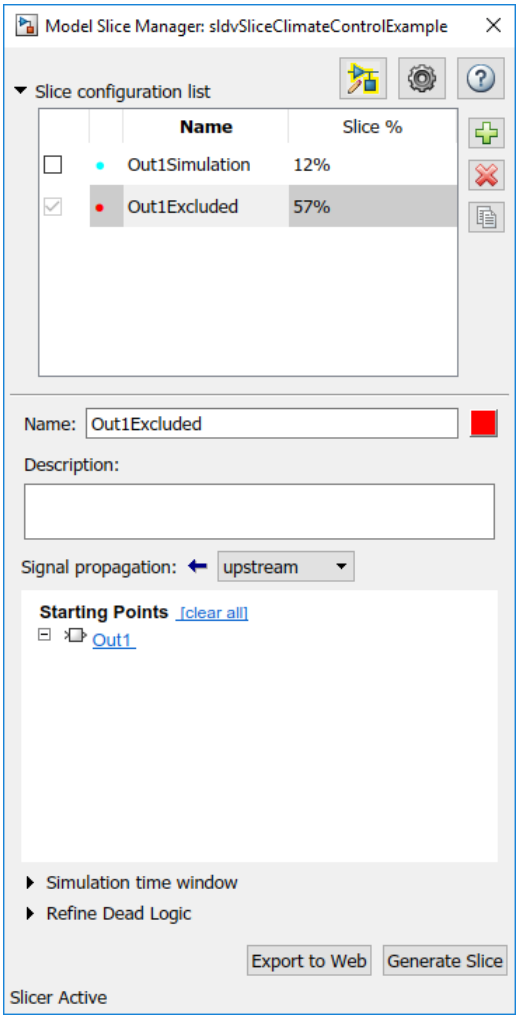
Exclusion points are useful for viewing a simplified set of model dependencies. For example, control feedback paths create wide dependencies and extensive model highlighting. You can use an exclusion point to restrict the analysis, particularly if your model has feedback paths.

Note Simplified standalone model creation is not supported for highlighted models with exclusion points.

1. In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.
2. To add a new slice configuration, click the add new button .
3. Set the slice properties:

- **Name**: `Out1Excluded`
- **Color**: (red)
- **Signal Propagation**: `upstream`

4 In the top level of the model, select the `Out1` block as the slice starting point. Right-click the `Out1` block and select **Model Slicer > Add as Starting Point**.

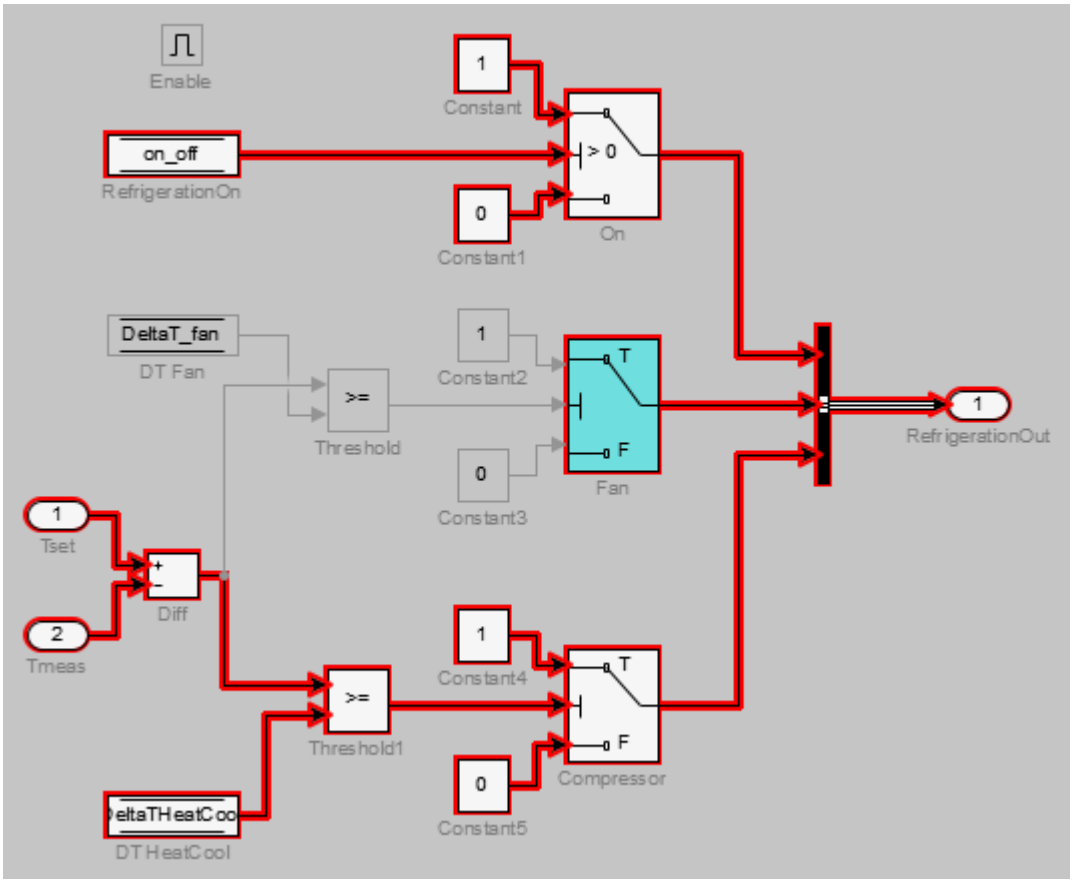


The model is highlighted.

5 To open the subsystem, double-click `Refrigeration`.

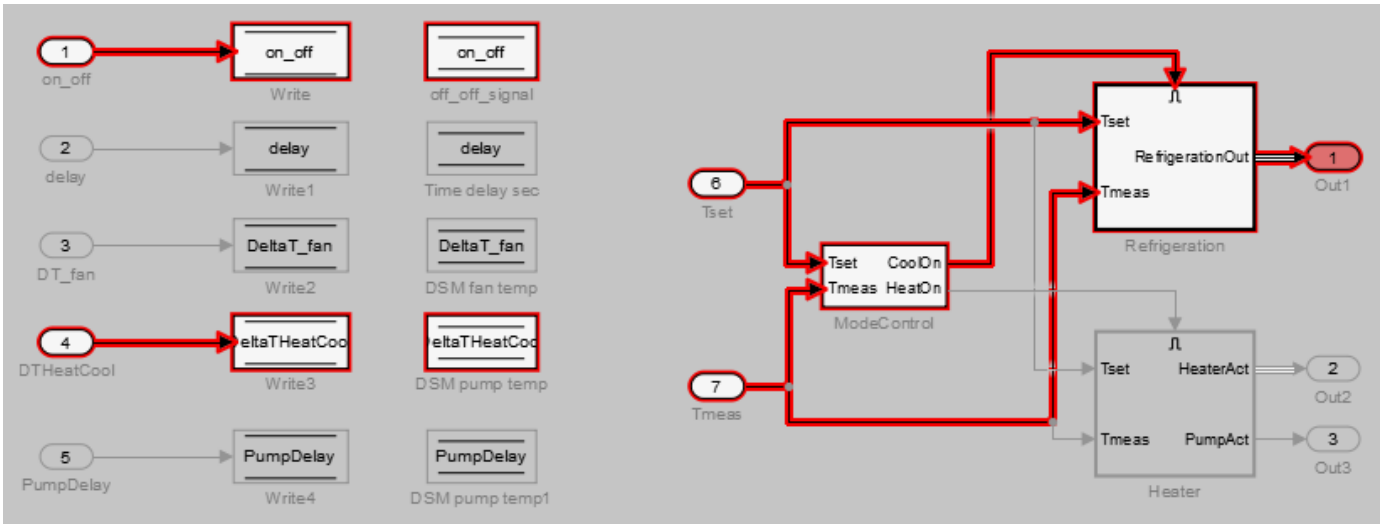
6 Right-click the `Fan` switch block, and then select **Model Slicer > Add as Exclusion Point**.

The blocks that are exclusively upstream of the `Fan` switch block are no longer highlighted. The `DT Fan` Data Store Read block is no longer highlighted.



7 To see how this constraint affects the highlighted portion of the model, view the parent system.

The `DSM fan temp` Data Store Memory block and the `Write2` Data Store Write block are no longer highlighted, because the `DT Fan` Data Store Read in the `Refrigeration` subsystem no longer accesses them.

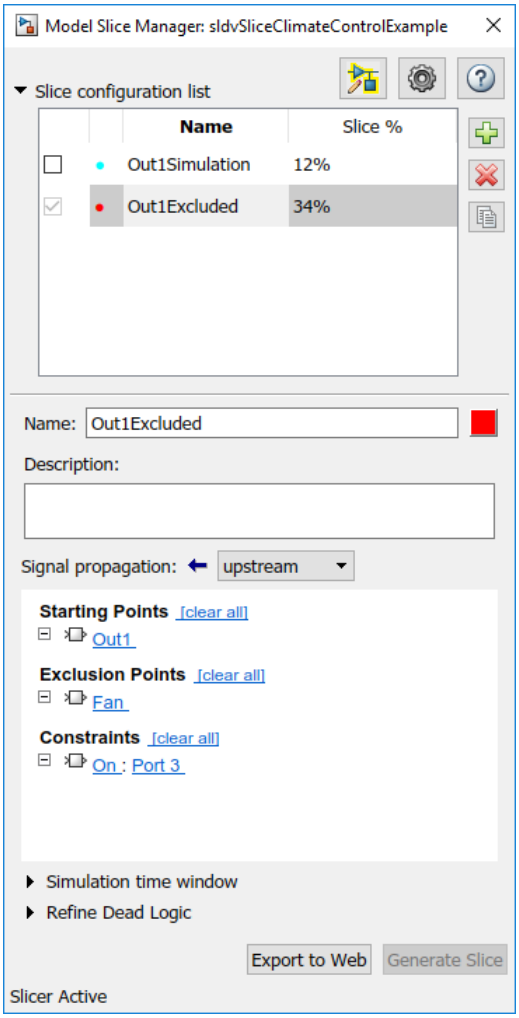


Exclude Inputs of a Switch Block

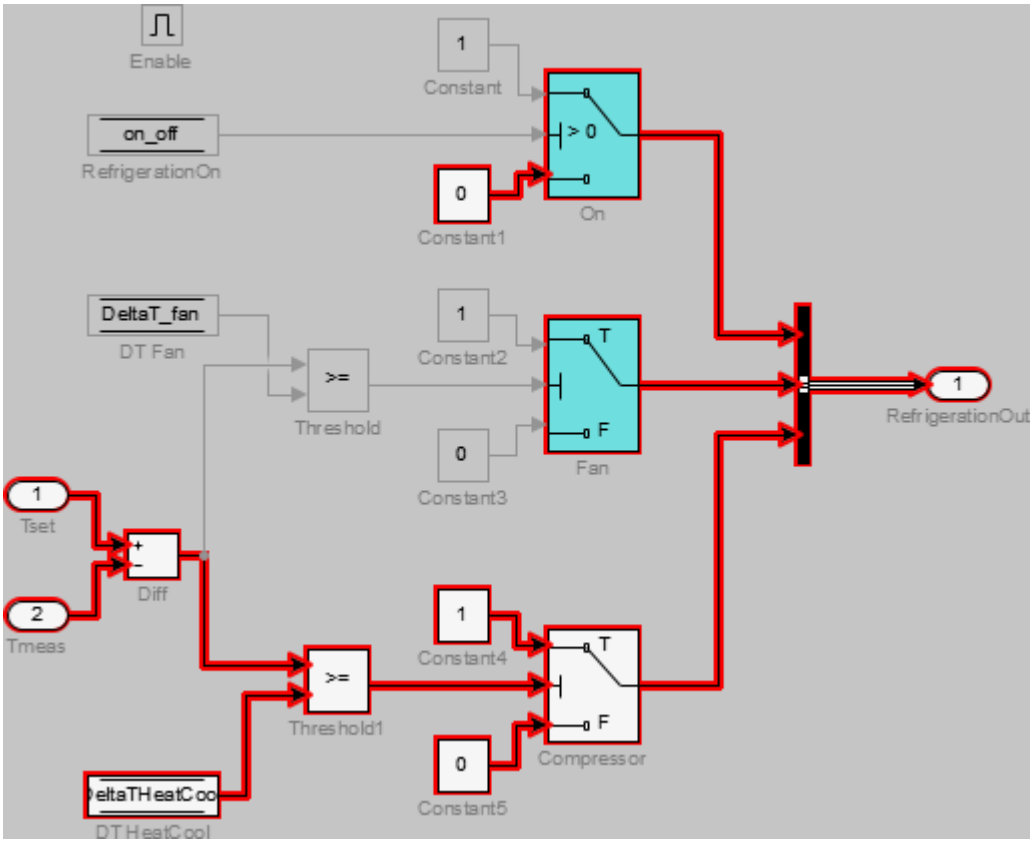
For complex signal routing, you can constrain the dependency analysis paths to a subset of the available paths through switch blocks. Constraints appear in the Model Slice Manager.

Note Simplified standalone model creation is not supported for highlighted models with constrained switch blocks.

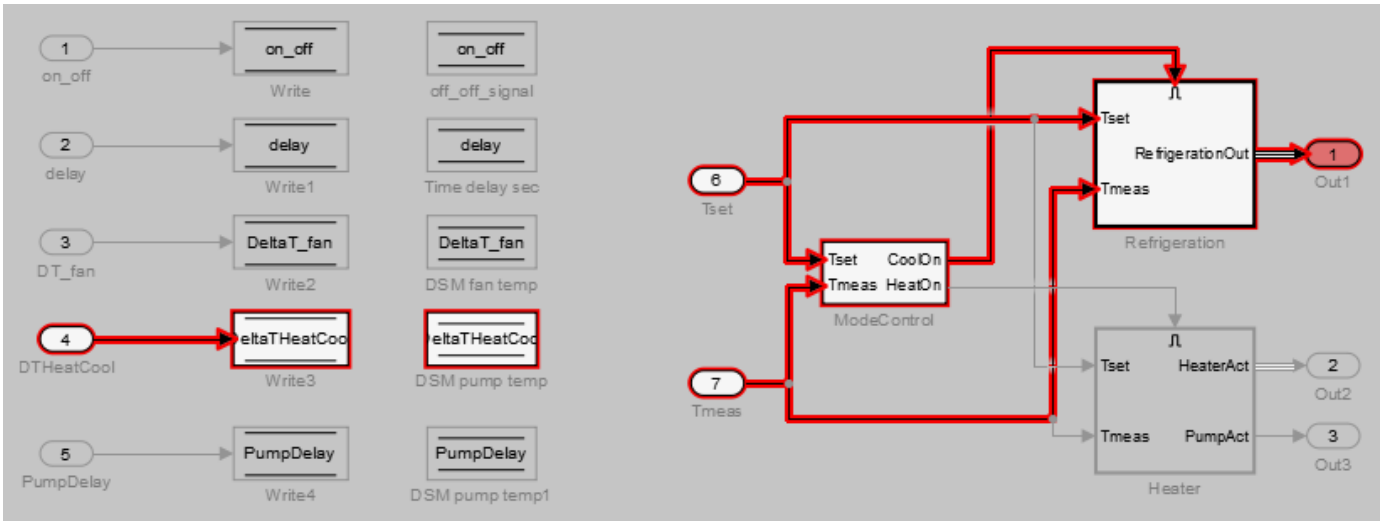
1. Double-click `Refrigeration` to open the subsystem.
2. Constrain the `On` switch block:
   - Right-click the switch block and select **Model Slicer > Add Constraint**.
   - In the Constraints dialog box, select **Port 3**.
   - Click **OK**.



The path is restricted to the `Constant1` port on the switch. The blocks that are upstream of **Port 1** and **Port 2** of the constrained switch are no longer highlighted. Only the blocks upstream of **Port 3** are highlighted.



3 To see how this constraint affects the highlighted portion of the model, view the parent system.



See Also

More About

- “Create a Simplified Standalone Model” on page 8-28

- “Model Slicer Considerations and Limitations” on page 8-43

Refine Dead Logic for Dependency Analysis

To refine the dead logic in your model for dependency analysis, use the Model Slicer. To provide an accurate slice, Model Slicer leverages Simulink Design Verifier dead logic analysis to remove the unreachable paths in the model. Model Slicer identifies the dead logic and refines the model slice for dependency analysis. For more information on Dead logic, see “Dead Logic Detection” (Simulink Design Verifier).

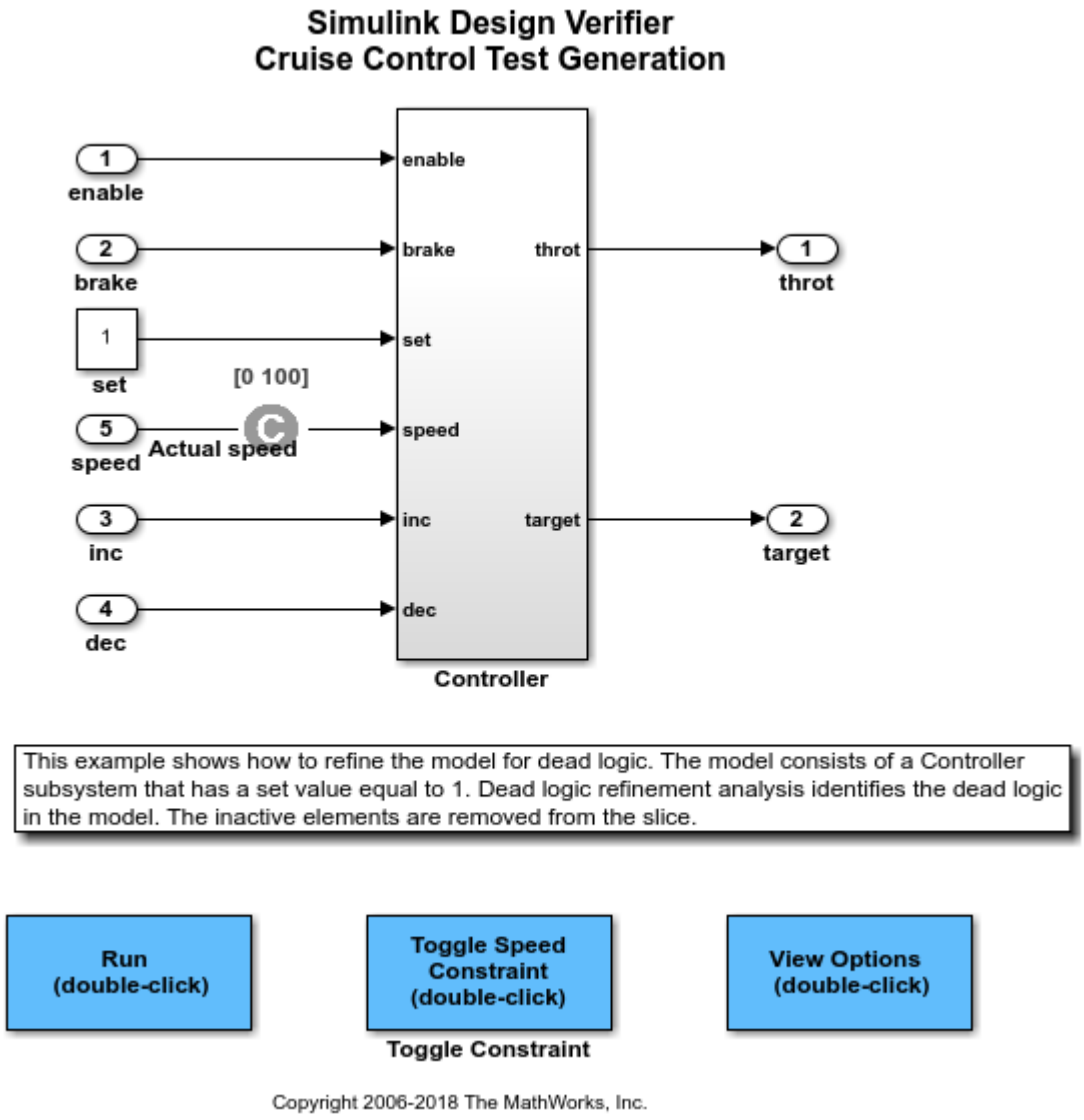
Analyze the Dead Logic

This example shows how to refine the model for dead logic. The `sldvSlicerdemo_dead_logic` model consists of dead logic paths that you refine for dependency analysis.

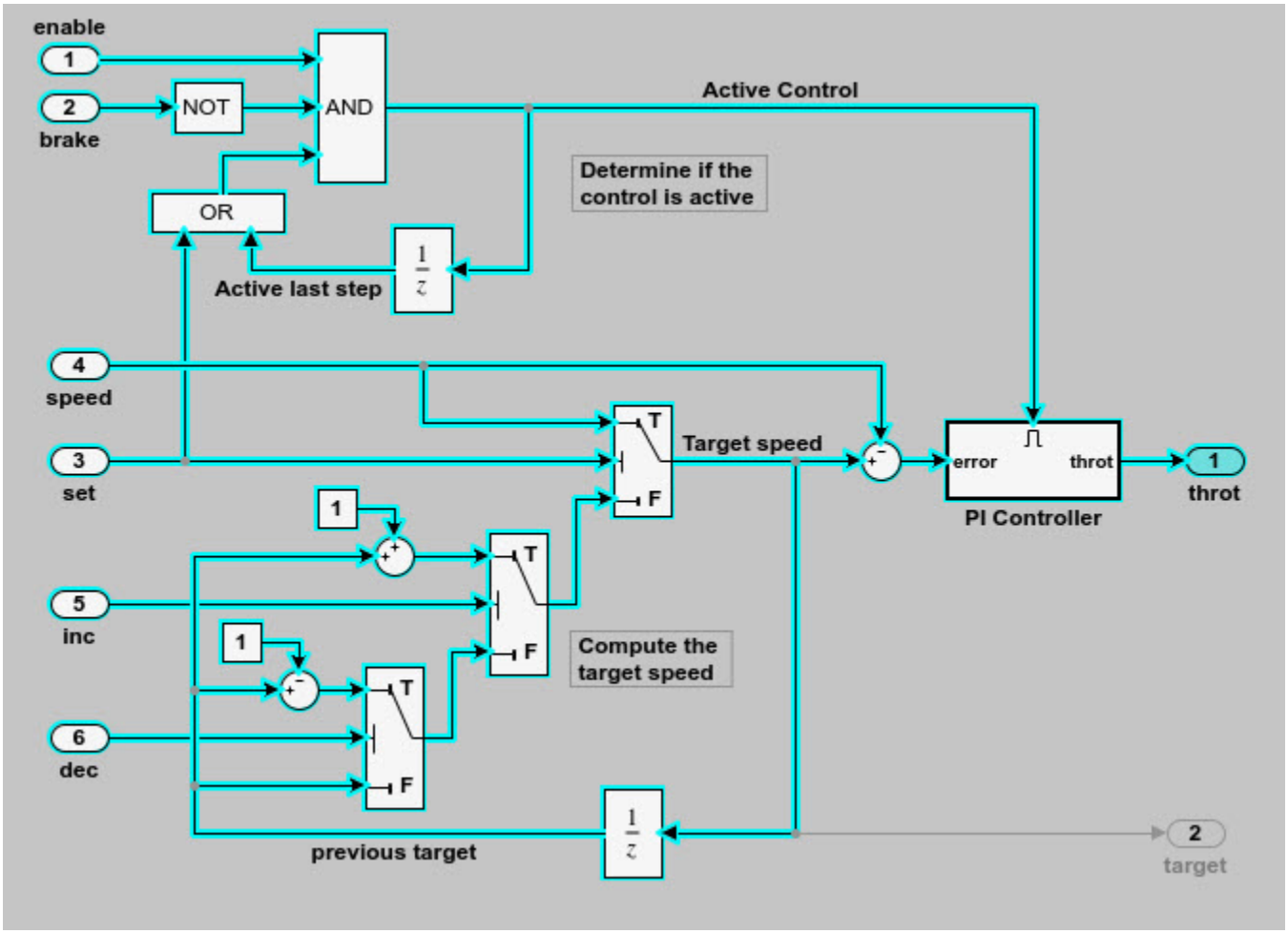
1. Open the `sldvSlicerdemo_dead_logic` model.

2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```



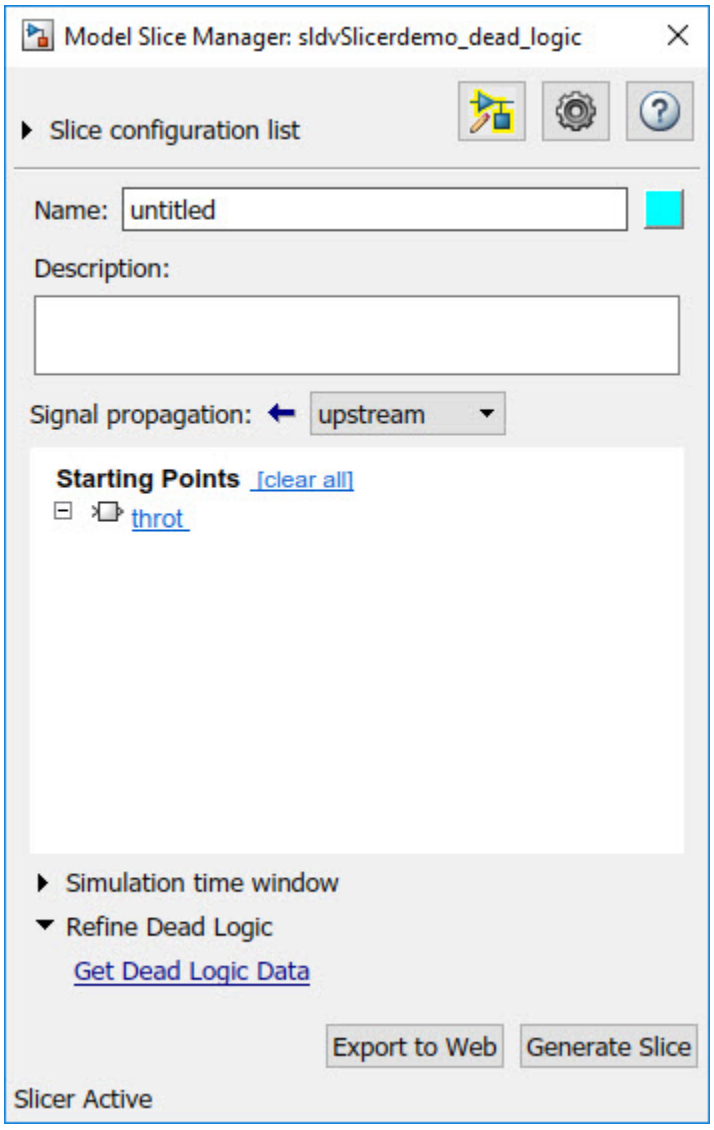
Open the `Controller` subsystem and add the outport `throt` as the starting point.



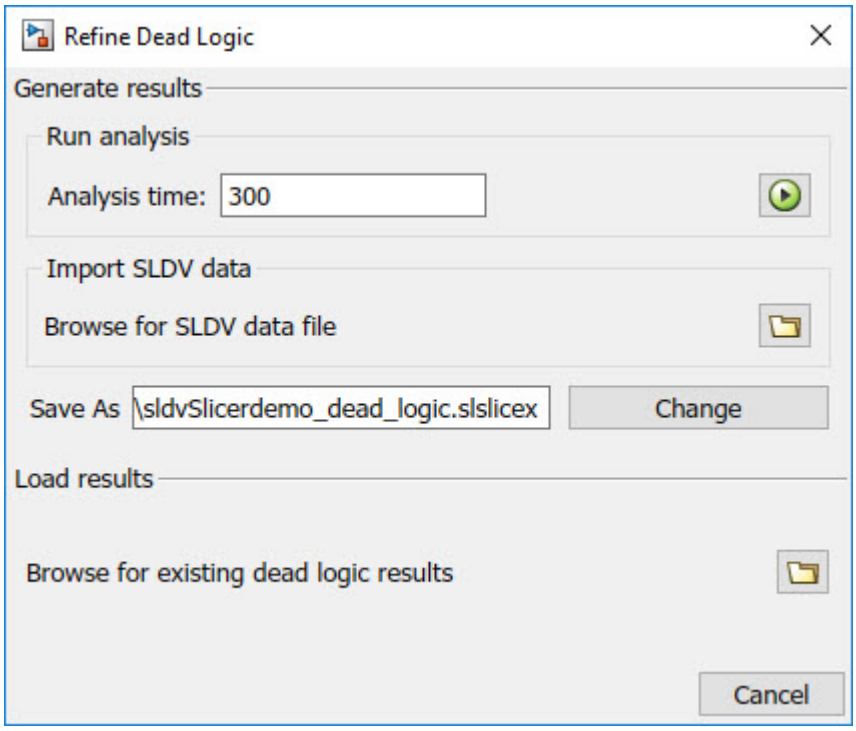
The Model Slicer highlights the upstream dependency of the `throt` outport.

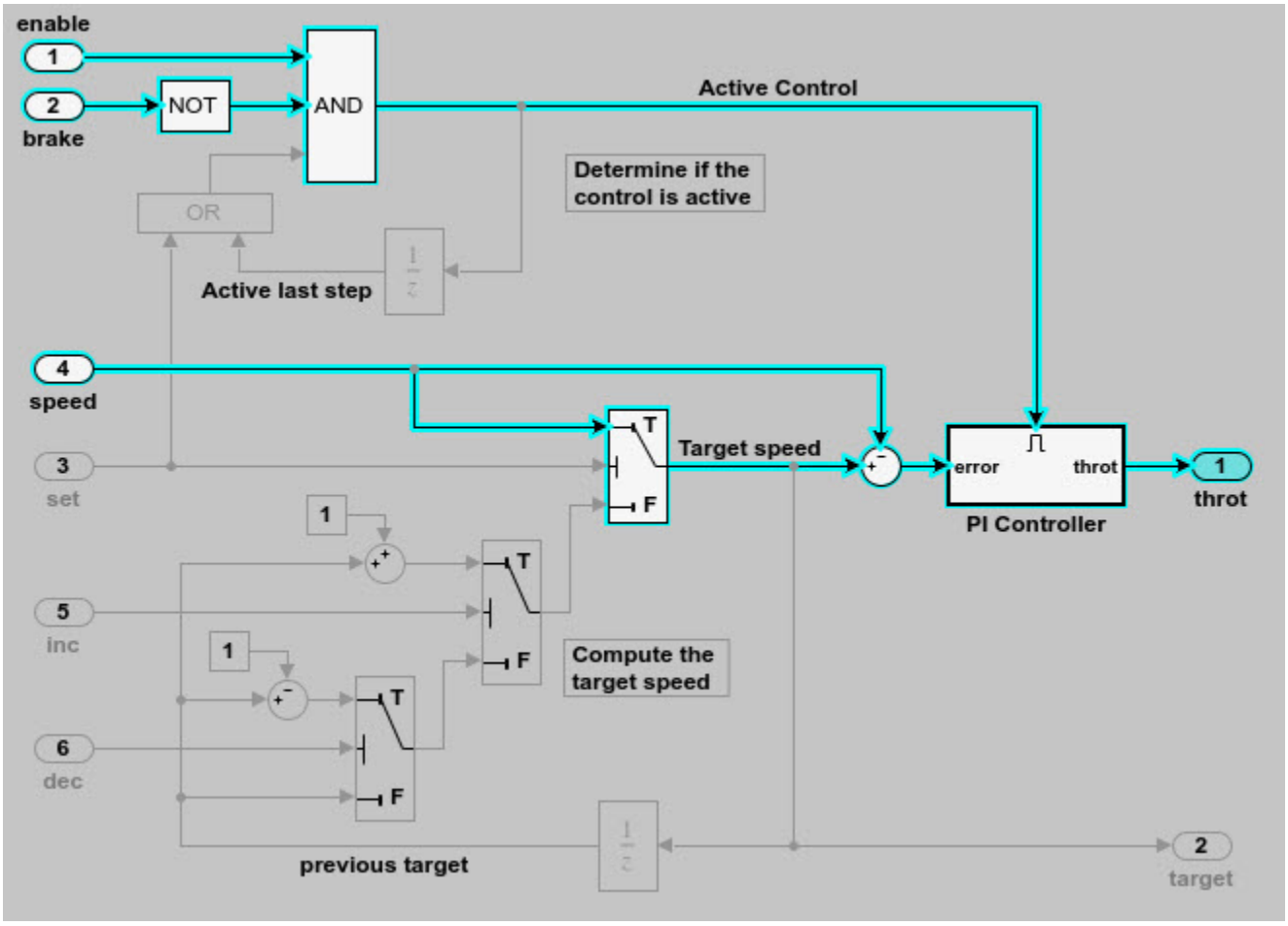
2. In the Model Slice Manager, select **Refine Dead Logic**.

3. Click **Get Dead Logic Data**.



4. Specify the **Analysis time** and run the analysis. You can import existing dead logic results from the `sldvData` file or load existing `.slslicex` data for analysis. For more information, see “Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results” on page 8-63.


Refine Dead Logic
Generate results
Run analysis
Analysis time:
300
Import SLDV data
Browse for SLDV data file
Save As
\sldvSlicerdemo_dead_logic.slslicex
Change
Load results
Browse for existing dead logic results
Cancel



As the `set` input is equal to `true`, the `False` input to switch is removed for dependency analysis. Similarly, the output of block `OR` is always `true` and removed from the model slice.

See Also

More About

- “Refine Highlighted Model” on page 8-12
- “Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results” on page 8-63

Create a Simplified Standalone Model

You can simplify simulation, debugging, and formal analysis of large and complex models by focusing on areas of interest in your model. After highlighting a portion of your model using Model Slicer, you can generate a simplified standalone model incorporating the highlighted portion of your original model. Apply changes to the simplified standalone model based on simulation, debugging, and formal analysis, and then apply these changes back to the original model.

Note Simplified standalone model creation is not supported for highlighted models with exclusion points or constrained switch blocks. If you want to view the effects of exclusion points or constrained switch blocks on a simplified standalone model, first create the simplified standalone model, and then add exclusion points or constrained switch blocks.

1. Highlight a portion of your model using Model Slicer.

   See “Highlight Functional Dependencies” on page 8-2 and “Refine Highlighted Model” on page 8-12.
2. In the Model Slice Manager, click **Generate slice**.
3. In the **Select File to Write** dialog box, select the save location and enter a model name.

   The simplified standalone model contains the highlighted model items.
4. To remove highlighting from the model, close the Model Slice Manager.

When generating a simplified standalone model from a model highlight, you might need to refine the highlighted model before the simplified standalone model can compile. See the “Model Slicer Considerations and Limitations” on page 8-43 for compilation considerations.

See Also

More About

- “Configure Model Highlight and Sliced Models” on page 8-40

Highlight Active Time Intervals by Using Activity-Based Time Slicing

Stateflow states and transitions can be active, inactive, or sleeping during model simulation. You can use Model Slicer to constrain model highlighting to only highlight the time intervals in which certain Stateflow “States” (Stateflow) and “Transitions” (Stateflow) are active. Therefore, you are able to refine your area of interest to only those portions of your model that affect model simulation during the operation of the selected states and transitions. You can also constrain model highlighting to the intersection of the time intervals of two or more states or transitions.

In this section...
“Highlighting the Active Time Intervals of a Stateflow State or Transition” on page 8-29
“Activity-Based Time Slicing Limitations and Considerations” on page 8-35
“Stateflow State and Transition Activity” on page 8-35

Highlighting the Active Time Intervals of a Stateflow State or Transition

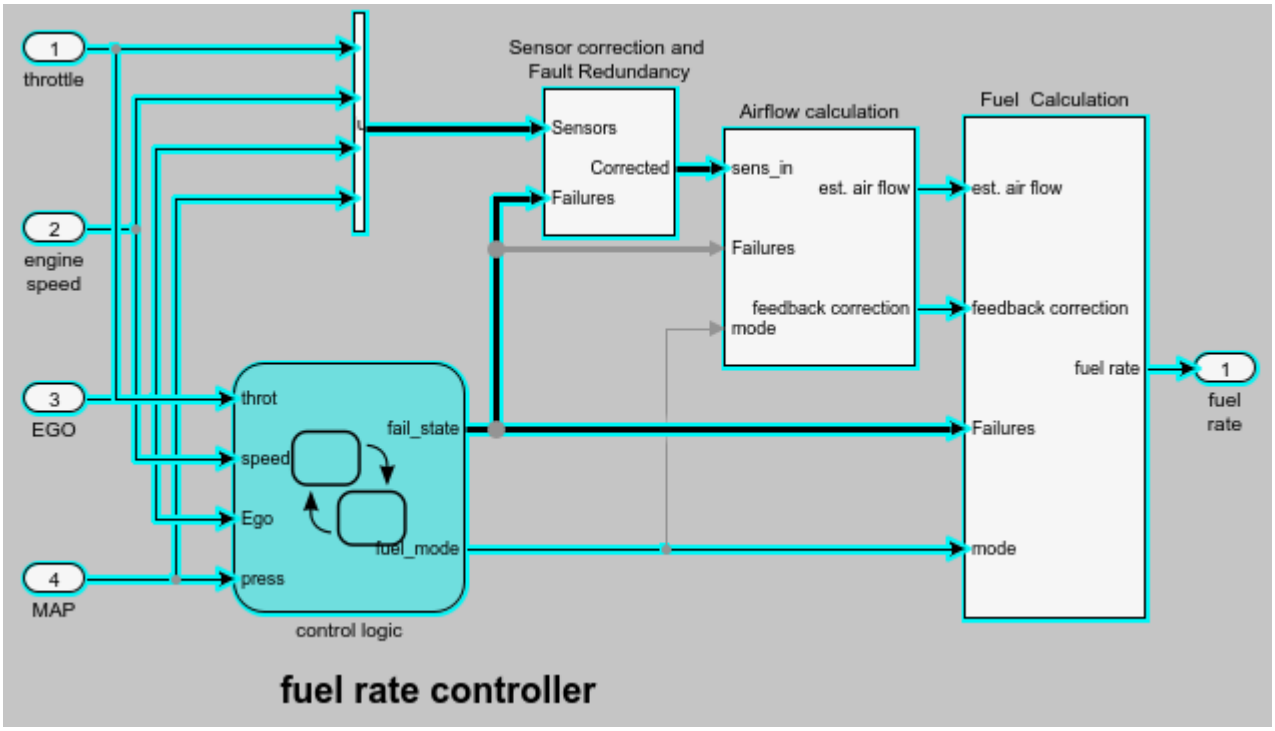
The `slslicer_fuelsys_activity_slicing` model contains a fault-tolerant fuel control system. In this tutorial, you use activity-based time slicing to refine a model highlight to only those time intervals in which certain states and transitions are active. You must be familiar with how to “Highlight Functional Dependencies” on page 8-2 by using Model Slicer.

Create a Dynamic Slice Highlight for an Area of Interest

1. Add the example folder to the search path.

   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   ```
2. Open the `slslicer_fuelsys_activity_slicing` model.

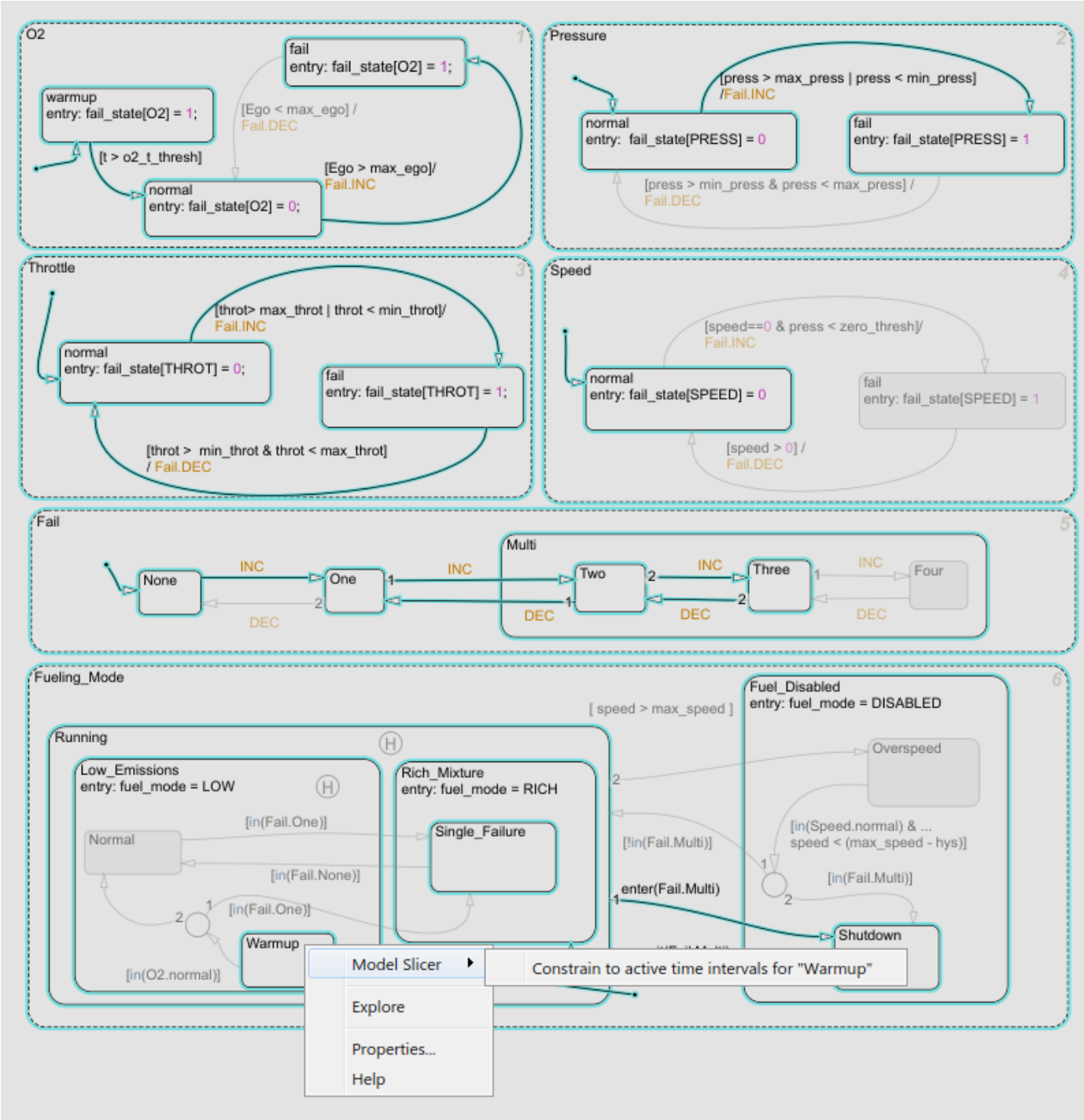
   ```
   open_system('slslicer_fuelsys_activity_slicing')
   ```
3. Open Model Slicer and add the `control logic` Stateflow chart in the fuel rate controller subsystem as a Model Slicer starting point.
4. Highlight the portions of the model that are upstream of the `control logic` Stateflow chart.
5. Simulate the model within a restricted simulation time window (maximum 20 seconds) to highlight only the areas of the model upstream of the starting point and active during the time window of interest.



Constrain the Model Highlight to the Active Time Interval of a Stateflow State

1. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
2. Navigate to the `control logic` Stateflow chart in the `fuel rate controller` subsystem.

   ```
   open_system('slslicer_fuelsys_activity_slicing/fuel rate controller/control logic')
   ```
3. To constrain the model highlight to only those time intervals in which the **Fueling_Mode** > **Running** > **Low_Emissions** > **Warmup** state is active, right-click the `Warmup` state and select **Model Slicer** > **Constrain to active time intervals for "Warmup"**.



Model Slicer is updated to highlight only those portions of the model that are active during the time intervals in which the `warmup` state is active.

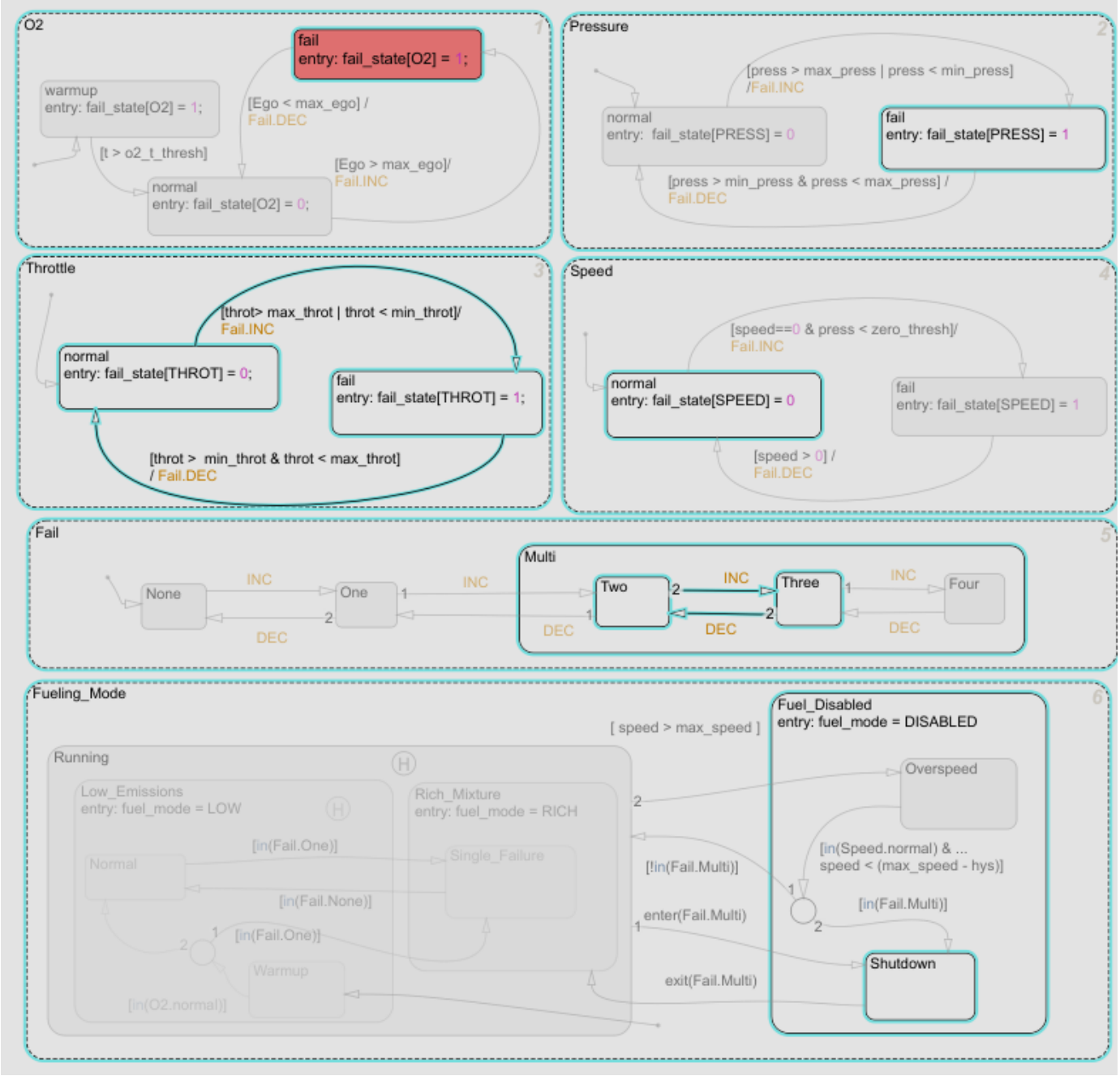
The Model Slice Manager is also updated to show the time interval in which the `warmup` state is active:

Actual simulation time: 0.01 to 3.86 seconds : 1 active interval

The highlight shows a `normal` to `fail` transition in the `Pressure` state, showing that a pressure failure occurred during the time interval in which the `Warmup` state was active.

Constrain the Model Highlight to the Intersection of the Active Time Intervals of a Stateflow State and Transition

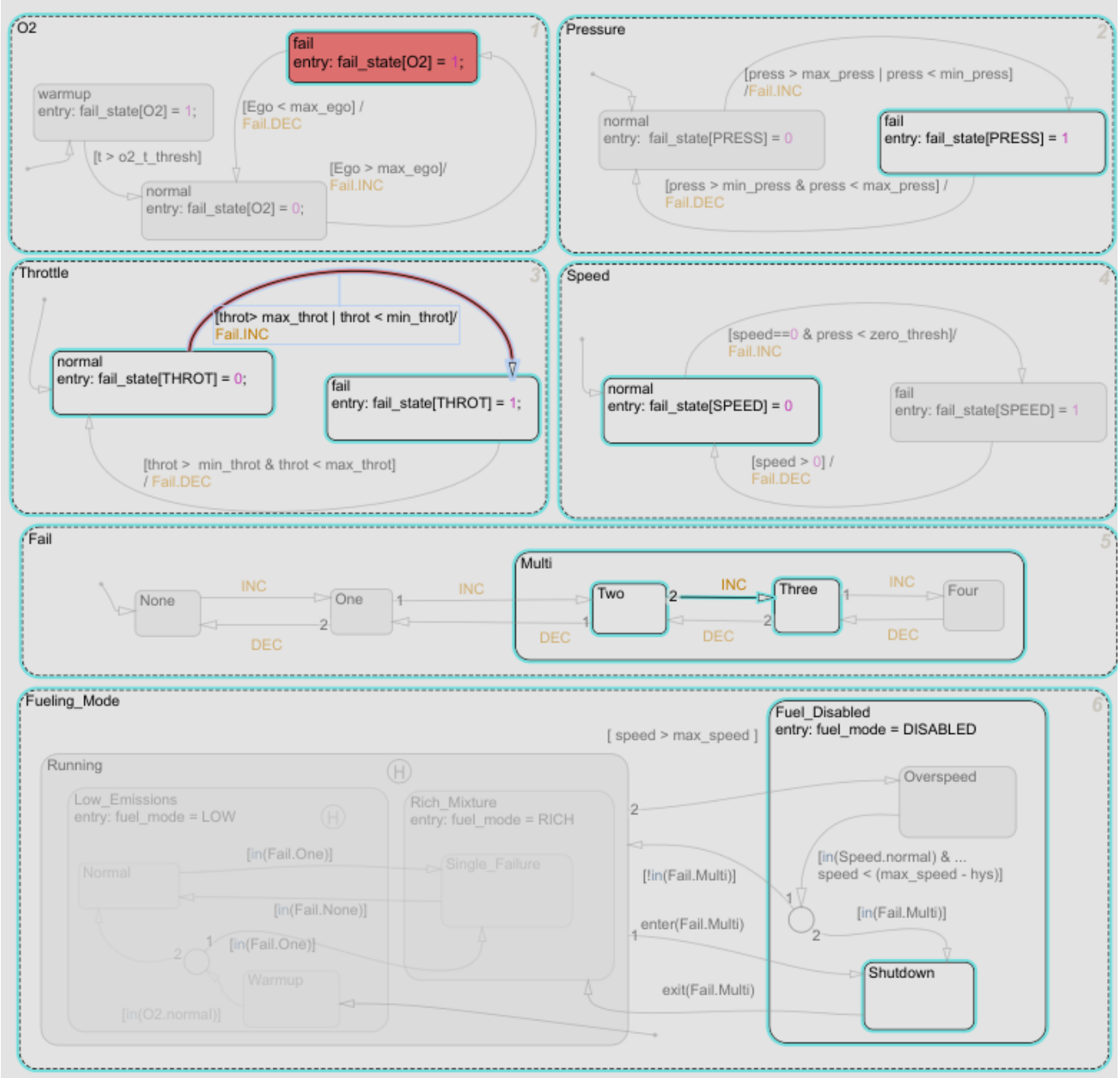
1 Clear any time interval constraints from the Model Slice Manager.
2 Constrain the model highlight to only those time intervals in which the **O2 > fail** state is active.



Model Slicer is updated to highlight only those portions of the model that are active during the time intervals in which the **O2 > fail** state is active. The Model Slice Manager is also updated to show the time interval in which the **O2 > fail** state is active:

Actual simulation time: 4.83 to 20 seconds : 1 active interval

3 To constrain the highlighting to the time interval in which the **O2 > fail** state is active and the `normal` to `fail` transition occurs for the `Throttle` chart, right-click the `normal` to `fail` transition and add it as a constraint. Model Slicer is updated to highlight only those portions of the model that are active during the intersection of the time intervals in which the **O2 > fail** state is active and the `normal` to `fail` transition occurs for the `Throttle` chart.



The Model Slice Manager is also updated to show the time interval in which the **O2 > fail** state is active and the `normal` to `fail` transition occurs for the `Throttle` chart:

Actual simulation time: 13.87 to 13.87 seconds : 1 active interval

Activity-Based Time Slicing Limitations and Considerations

For limitations and considerations of activity-based time slicing, see “Model Slicer Considerations and Limitations” on page 8-43.

Stateflow State and Transition Activity

For more information on Stateflow state and transition activity, see “Chart Simulation Semantics” (Stateflow), “Types of Chart Execution” (Stateflow), and “Syntax for States and Transitions” (Stateflow).

See Also

More About

- “Using Model Slicer with Stateflow” on page 8-49
- “States” (Stateflow)
- “Transitions” (Stateflow)

Simplify a Standalone Model by Inlining Content

You can reduce file dependencies by inlining model content when you generate the sliced model. Inlining brings functional content into the sliced model and can eliminate model references, library links, and variant structures that are often not needed for model refinement or debugging.

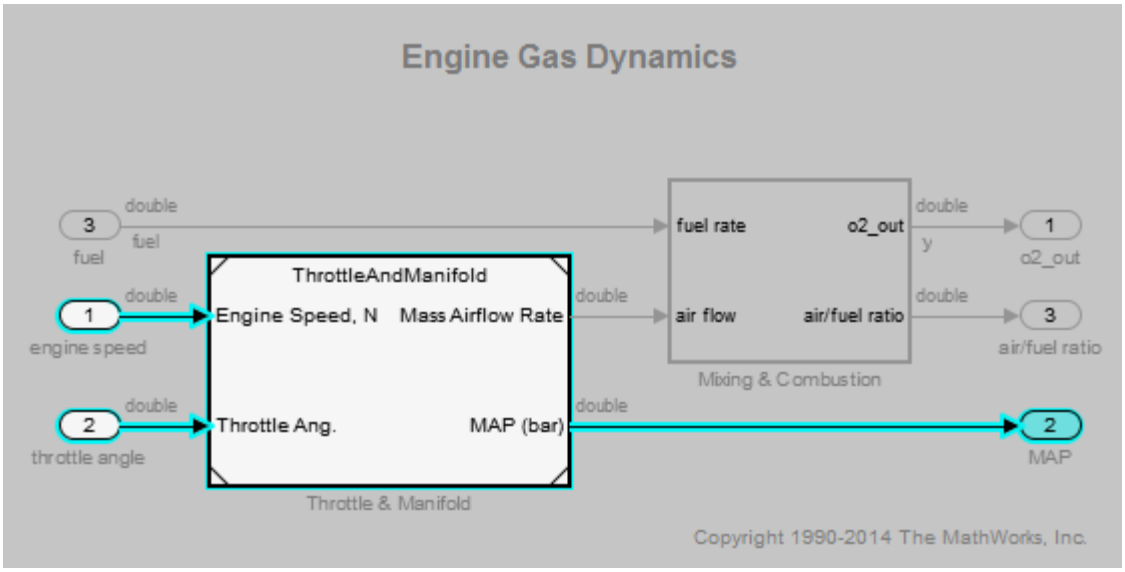
If you want to disable inlining for certain block types, open the Model Slice Manager and click the options button . Select only the block types for which you want to inline content. For information on block-specific inlining behavior, see “Inline Content Options” on page 8-42.

This example demonstrates inlining content of a model referenced by a Model block.

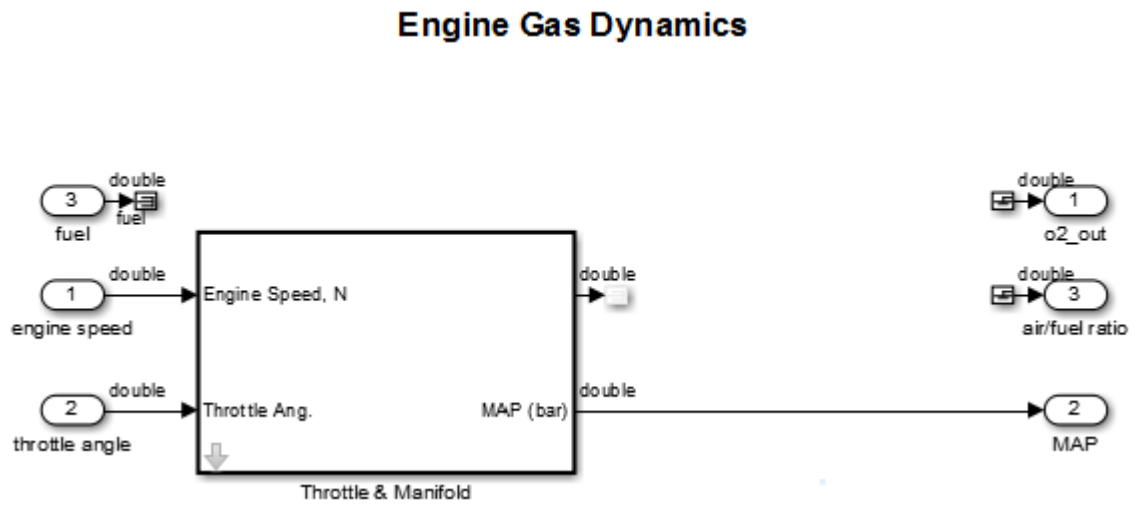
1. Add the path to the example and open the model

   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   open_system('sldvSliceEngineDynamicsExample')
   ```

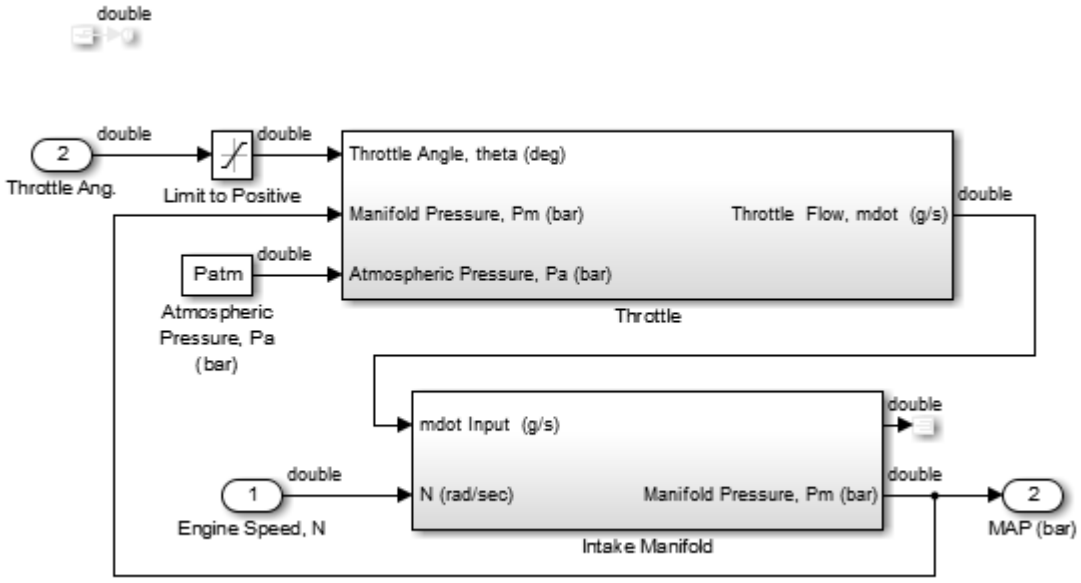
2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
3. In the model, right-click the MAP outport and select **Model Slicer > Add as Starting Point**. The path is highlighted through the Model block.



4. Create a sliced model from the highlight. In the Model Slice Manager, click the **Generate slice** button.
5. Enter a file name for the sliced model.
6. The sliced model contains the highlighted model content. The model reference is removed.



7 Click the arrow to look under the mask of the ThrottleAndManifold subsystem. The content from the referenced model is inlined into the model in the masked subsystem.



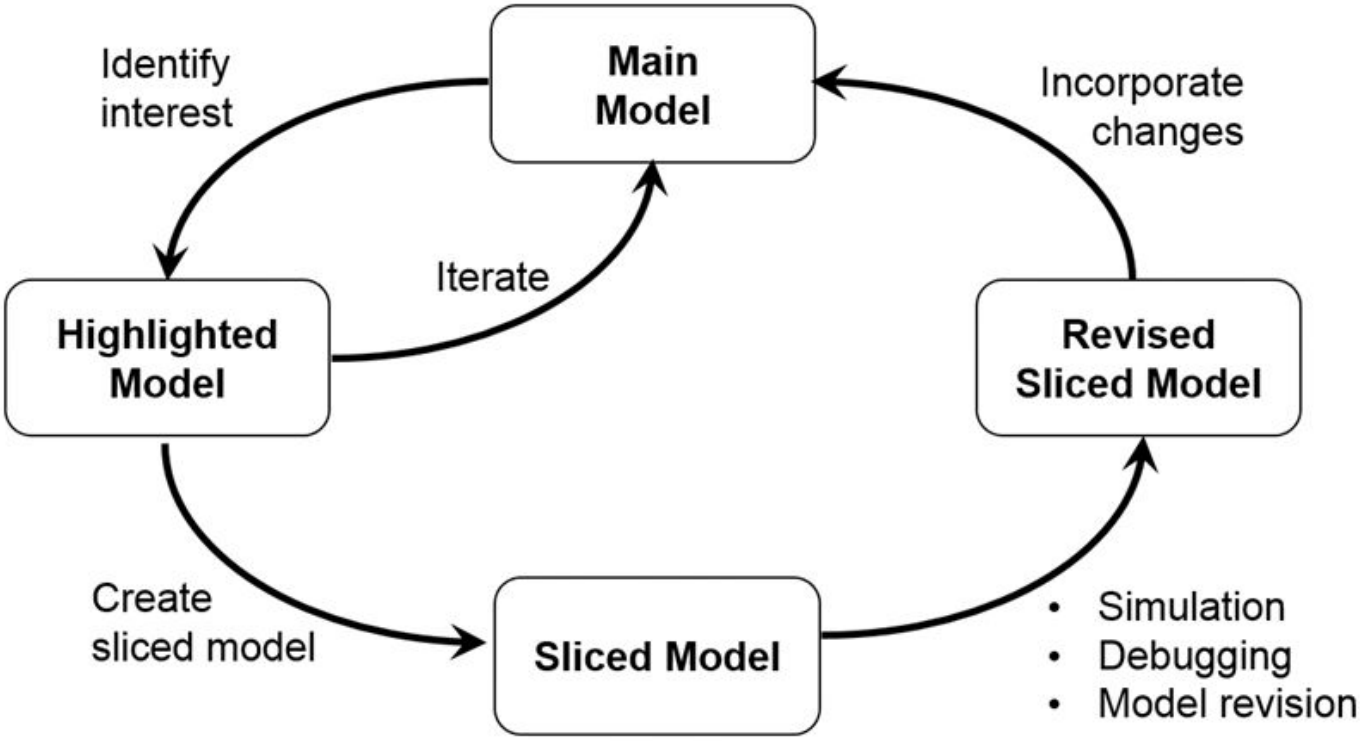
Workflow for Dependency Analysis

In this section...
“Dependency Analysis Workflow” on page 8-38
“Dependency Analysis Objectives” on page 8-38

Model analysis includes determining dependencies of blocks, signals, and model components. For example, to view blocks affecting a subsystem output, or trace a signal path through multiple switches and logic. Determining dependencies can be a lengthy process, particularly for large or complex models. Use Model Slicer as a simple way to understand functional dependencies in large or complex models. You can also use Model Slicer to create simplified standalone models that are easier to understand and analyze, yet retain their original context.

Dependency Analysis Workflow

The dependency analysis workflow identifies the area of interest in your model, generates a sliced model, revises the sliced model, and incorporates those revisions in the main model.



Dependency Analysis Objectives

To identify the area of interest in your model, determine objectives such as:

- What item or items are you analyzing? Analysis begins with at least one starting point.
- In what direction does the analysis propagate? The dependency analysis propagates upstream, downstream, or bidirectionally from the starting points.

- What model items or paths do you want to exclude from analysis?
- What paths do you want to constrain? If your model has switches, you can constrain the switch positions for analysis.
- Is your model a closed-loop system? If so, the highlighted portion of the model can include model dependencies from the feedback loop. Consider excluding blocks from the feedback loop to refine the highlighted portion of the model.
- Do you want to analyze static dependencies, or include simulation effects? Static analysis considers model dependencies for possible simulation paths. Simulation-based analysis highlights only paths active during simulation.

See Also

Related Examples

- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12
- “Create a Simplified Standalone Model” on page 8-28

Configure Model Highlight and Sliced Models

In this section...
“Model Slice Manager” on page 8-40
“Model Slicer Options” on page 8-40
“Storage Options” on page 8-40
“Refresh Highlighting Automatically” on page 8-41
“Sliced Model Options” on page 8-41
“Trivial Subsystems” on page 8-41
“Inline Content Options” on page 8-42

Model Slice Manager

Set the properties of your model highlight and standalone sliced model using the Model Slice Manager.

Click the toggle mode button to switch between model edit mode and model highlight mode.

If automatic highlighting is disabled in the slice settings, refresh the model highlight using the refresh button . Refresh the highlight after changing the slice configuration.

Model Slicer Options

You can customize the slice behavior using the options dialog box, which is accessed with the options button .

Storage Options

Changes you make to a model slice configuration are saved automatically. You can store the slice configuration in the model SLX file, or in an external SLMS file. Saving the configuration externally can be useful if your SLX file is restricted by a change control system.

To set the storage location, click the options button in the Model Slice Manager and set the location in the **Storage options** pane.

Settings

Store in <model_name>.slx

Saves the model slice configuration in your model’s SLX file

Store in external file

Saves the model slice configuration in a separate SLMS file you specify by clicking the **Save As** button. The model slice configuration filename is shown in **File**.

Refresh Highlighting Automatically

Enables automatic refresh of a model highlight after changing the slice configuration.

Settings

on (default)

Model highlighting refreshes automatically.

off

Model highlighting must be refreshed manually. Click the refresh button in the Model Slice Manager to refresh the highlight.

Sliced Model Options

You can control what items are retained when you create a sliced model from a model highlight using the options in the **Sliced model options** pane.

Option	On (selected)	Off (cleared)
Retain signal observers	Signal observers, such as scopes, displays, and test condition blocks, are retained in the sliced model.	Signal observers are not retained in the sliced model (default).
Retain root-level inports and outports	Root-level ports are retained in the sliced model (default).	Root-level ports are not retained in the sliced model.
Expand trivial subsystems	Trivial subsystems are expanded in the sliced model and the subsystem boundary is removed (default).	Trivial subsystems are not expanded in the sliced model and the subsystem boundary is retained. See“Trivial Subsystems” on page 8-41.

Trivial Subsystems

If a subsystem has all of these characteristics, Model Slicer considers the subsystem trivial:

- If the subsystem is virtual, it contains three or fewer nonvirtual blocks.
- If the subsystem is atomic, it contains one or fewer nonvirtual blocks.
- The subsystem has two or fewer inports.
- The subsystem has two or fewer outports.
- The active inport or outport blocks of the subsystem have default block parameters.
- The system does not contain Goto Tag Visibility blocks.
- In the Block Properties dialog box, the subsystem **Priority** is empty.
- The data type override parameter (if applicable) is set to use local settings.

Note If you generate a sliced model which does not remove contents of a particular subsystem, the subsystem remains intact in the sliced model.

Inline Content Options

When you create a sliced model from a highlight, model items can be inlined into the sliced model. The **Inline content options** pane controls which model components are inlined in generating a sliced model.

Model Component	Inlining on (selected)	Inlining off (cleared)
Libraries	Model items inside sliced libraries are inlined in the sliced model and the library link is removed. (default)	Model items inside sliced libraries are not inlined in the sliced model and library link remains in place.
Masked subsystems	Model items inside sliced masked subsystems are inlined in the sliced model. (default) The mask is retained in the sliced model.	Model items inside sliced masked subsystems are not inlined in the sliced model and the mask is retained.
Model blocks	Model items are inlined to the sliced model from the model referenced by the Model block. The Model block is removed. (default) **Note** Model Slicer cannot inline model blocks that are not in `Normal` mode.	Model items are not inlined to the sliced model from the model referenced by the Model block. The Model block is retained.
Variants	Model items are inlined to the sliced model from the active variant. Variants are removed. (default)	Model items are not inlined to the sliced model from the variant. The variant is retained.

See Also

Related Examples

- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12
- “Simplify a Standalone Model by Inlining Content” on page 8-36

Model Slicer Considerations and Limitations

When you work with the Model Slicer, consider these behaviors and limitations:

In this section...
“Model Compilation” on page 8-43
“Model Highlighting and Model Editing” on page 8-43
“Standalone Sliced Model Generation” on page 8-43
“Sliced Model Considerations” on page 8-43
“Port Attribute Considerations” on page 8-44
“Simulation Time Window Considerations” on page 8-45
“Simulation-based Sliced Model Simplifications” on page 8-45
“Starting Points Not Supported” on page 8-46
“Model Slicer Support Limitations for Simulink Software Features” on page 8-46
“Model Slicer Support Limitations for Simulation Stepper” on page 8-46
“Model Slicer Support Limitations for Simulink Blocks” on page 8-46
“Model Slicer Support Limitations for Stateflow” on page 8-47

Model Compilation

When you open Model Slice Manager, the model is compiled. To avoid a compilation error, before you open Model Slice Manager, make sure that the model is compilable.

Model Highlighting and Model Editing

When a slice highlight is active, you cannot edit the model. You can switch to model edit mode and preserve the highlights. When you switch back to slice mode, the slice configuration is recomputed and the highlight is updated.

Standalone Sliced Model Generation

Sliced model generation requires one or more starting points for highlighting your model. Sliced model generation is not supported for:

- Forward-propagating (including bidirectional) dependencies
- Constraints
- Exclusion points

Sliced model generation requires a writable working folder in MATLAB.

Sliced Model Considerations

When you generate a sliced model from a model highlight, simplifying your model can change simulation behavior or prevent the sliced model from compiling. For example:

- Model simplification can change the sorted execution order in a sliced model compared to the original model, which can affect the sliced model simulation behavior.
- If you generate a sliced model containing a bus, but not the source signal of that bus, the sliced model can contain unresolved bus elements.
- If you generate a sliced model that inlines a subset of the contents of a masked block, make sure that the subsystem contents resolve to the mask parameters. If the contents and mask do not resolve, it is possible that the sliced model does not compile.
- If the source model uses a bus signal, ensure that the sliced model signals are initialized correctly. Before you create the sliced model, consider including an explicit copy of the bus signal in the source model. For example, you can include a Signal Conversion block with the **Output** option set to `Signal Copy`.
- For solver step sizes set to `auto`, Simulink calculates the maximum time step in part based on the blocks in the model. If the sliced model removes blocks that affect the time step determination, the time step of the sliced model can differ from the source model. The time step difference can cause simulation differences. Consider setting step sizes explicitly to the same values calculated in the source model.

Port Attribute Considerations

You can use blocks that the Model Slicer removes during model simplification to determine compiled attributes, such as inherited sample times, signal dimensions, and data types. The Model Slicer can change sliced model port attributes during model simplification to resolve underspecified model port attributes. If the Model Slicer cannot resolve these inconsistencies, you can resolve some model port attribute inconsistencies by:

- Explicitly specifying attributes in the source model instead of relying on propagation rules.
- Including in the sliced model the blocks that are responsible for the attribute propagation in your source model. Before you slice the model, add these blocks as additional starting points in the source model highlighting.
- Not inlining the model blocks that are responsible for model port attributes into the sliced model. For more information on model items that you can inline into the sliced model, see “Inline Content Options” on page 8-42.

Because of the way Simulink handles model references, you cannot simultaneously compile two models that both contain a model reference to the same model. When you generate a sliced model, the Model Slicer enters the **Slicer Locked (for attribute checking)** mode if these conditions are true:

- The parent model contains a referenced model.
- The highlighted portion of the parent model contains the referenced model.
- The referenced model is not inlined in the sliced model due to one of the following
  - You choose not to inline model blocks in the **Inline content options** pane of the **Model Slicer options**.
  - The Model Slicer cannot inline the referenced model. For more information on model items that Model Slicer cannot inline, see “Inline Content Options” on page 8-42.

To continue refining the highlighted portion of the parent model, you must first activate the slice highlight mode .

Simulation Time Window Considerations

Depending on the step size of your model and the values that you enter for the start time and stop time of the simulation time window, Model Slicer might alter the actual simulation start time and stop time.

- If you enter a stop or start time that falls between time steps for your model solver, the Model Slicer instead uses a stop or start time that matches the time step previous to the value that you entered. For more information on step sizes in Simulink, see “Compare Solvers” (Simulink).
- The stop time for the simulation time window cannot be greater than the total simulation time.

Simulation-based Sliced Model Simplifications

When you slice a model by using a simulation time window, some blocks in the source model, such as switch blocks, logical operator blocks, and others, can be replaced when creating the simplified standalone model. For example, a switch block that always passes one input is removed, and the active input is directly connected to the output destination. The unused input signal is also removed from the standalone model.

This table describes the blocks that the Model Slicer can replace during model simplification.

Block in Source Model	Simplification
Switch Multiport Switch	If only one input port is active, the switch is replaced by a signal connecting the active input to the block output.
Enabled Subsystem or Model	If the subsystem or model is always enabled, remove the control input and convert to a standard subsystem or model. If the subsystem is never enabled, replace the subsystem with a constant value defined by the initial condition.
Triggered Subsystem or Model	If the subsystem or model is always triggered, remove the trigger input and convert to a standard subsystem or model. If the subsystem is never triggered, replace the subsystem with a constant value defined by the initial condition.
Enabled and Triggered Subsystem or Model	If the subsystem is always executed, convert to a standard subsystem or model If the subsystem is never executed, replace the subsystem with a constant value defined by the initial condition.
Merge	If only one input port is active, the merge is replaced by a signal connecting the active input to the block output.

Block in Source Model	Simplification
If If Action	If only one action subsystem is active, convert to a standard subsystem or model and remove the If block.
Switch Case Switch Case Action	If only one action subsystem is active, convert to a standard subsystem or model and remove the Switch Case block.
Logical operator	Replace with constant when the block always outputs true or always outputs false. Replace the input signal with a constant if the input signal is always true or always false.

Starting Points Not Supported

The Model Slicer does not support these model items as starting points:

- Virtual blocks, other than subsystem Inport and Outport blocks
- Output signals from virtual blocks that are not subsystems

Model Slicer Support Limitations for Simulink Software Features

The Model Slicer does not support these features:

- Arrays of buses
- Analysis of Simulink Test test harnesses
- Models that contain Simscape physical modeling blocks
- Models that contain algebraic loops
- Loading initial states from the source model for sliced model generation, such as data import/export entries. Define initial states explicitly for the sliced model in the sliced model configuration parameters.
- Component slicing of the subsystems and referenced models that have multiple rates.
- Component slicing of the “Conditional Models” (Simulink) and Conditionally Executed Subsystems (Simulink).

Model Slicer Support Limitations for Simulation Stepper

When using Model Slicer with Simulation Stepper, the slice highlight after a Step Back may not be limited to a single step. The highlight can be influenced by the **Simulation Stepping Options > Interval between stored back steps**. For more information, see “Interval between stored back steps” (Simulink).

Model Slicer Support Limitations for Simulink Blocks

The table lists the Model Slicer support limitations for Simulink Blocks.

Block	Limitation
For Each Subsystem block	The simulation impact is ignored for blocks in a For Each subsystem. Therefore, applying a simulation time window returns the same dependency analysis result as a dependency analysis that does not use a simulation time window.
Function Caller block	Model Slicer does not support Function Caller blocks.
MATLAB Function block	Model Slicer assumes that any output depends on all inputs in the upstream direction and any input affects all outputs in the downstream direction.
Merge block	If you generate a slice by using a simulation time window, Merge blocks are removed in the standalone model if only a single path is exercised.
Model block	Model Slicer does not resolve data dependencies generated by global data store memory in Model blocks with **Simulation mode** set to `Accelerator`. Model Slicer does not support function-call root-level Inport blocks. For more information, see Export-Function Models (Simulink). Model Slicer does not analyze the contents within a reference to a "Reference Protected Models from Third Parties" (Simulink). When you slice a model that contains a protected model reference, the Model Slicer includes the entire model reference in the sliced model.
Resettable Subsystem block	Model Slicer does not support Resettable Subsystem blocks.
S-function block	Model Slicer assumes that any output depends on all inputs in the upstream direction and any input affects all outputs in the downstream direction. Model Slicer does not determine dependencies that result from an S-function block accessing model information dependent on a simulation time window.
State Read block	Model Slicer does not support State Read blocks.
State Write block	Model Slicer does not support State Write blocks.

Model Slicer Support Limitations for Stateflow

- When you highlight models containing a Stateflow chart or state transition table, Model Slicer assumes that any output from the Chart block or State Transition Table block depends on all inputs to the Chart block or State Transition Table block.
- When you slice a model with a Stateflow chart or a state transition table, Model Slicer does not simplify the chart or table. The chart or table is included in its entirety in the sliced model.
- If you do not "Define a Simulation Time Window" on page 8-12 when you highlight functional dependencies in a Stateflow chart or state transition table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.

- When you highlight functional dependencies in a Stateflow chart or state transition table for a defined simulation time window, Model Slicer does not highlight only the states and transitions that affect the selected starting point. Instead, the Model Slicer highlights elements that are active in the time window that you specify.
- The Model Slicer does not determine dependencies between Stateflow graphical functions and function calls in other Stateflow charts.
- Graphical functions and their contents that were not active during the selected time window can potentially remain highlighted in some cases.
- Entry into states that are preempted due to events can potentially remain highlighted in some cases. For example, after a parent state is entered, an event action can exit the state and preempt entry into the child state. In such a case, the Model Slicer highlights the entry into the child state.
- The Model Slicer does not support:
  - Embedded MATLAB Function blocks
  - Simulink functions
  - Truth Table blocks
  - Machine-parented data or events in Stateflow.

  .

Activity-Based Time Slicing Considerations for Stateflow

As measured by the 'Executed Substate' decision coverage, state activity refers to these during/exit actions:

- Entry into a state does not constitute activity.
- The active time interval for a state or transition includes the moment in which the selected state exits and the subsequent state is entered.
- Indirect exits from a state or transition do not constitute activity. For example, if a state *C* exits because its parent state *P* exits, state C is not considered active.

For more information on decision coverage for Stateflow charts, see “Decision Coverage for Stateflow Charts” (Simulink Coverage).

When you “Highlight Active Time Intervals by Using Activity-Based Time Slicing” on page 8-29, you can select states and transitions only as activity constraints. You cannot select these Stateflow objects as constraints:

- Parallel states
- Transitions without conditions, such as unlabeled transitions which do not receive decision coverage
- States or transitions within library-linked charts
- XOR states without siblings. For example, if a state *P* has only one child state *C*, you cannot select state *C* as an activity constraints because state *P* does not receive decision coverage for the executed substate

See Also

“Algebraic Loop Concepts” (Simulink) | “Solver Pane” (Simulink)

Using Model Slicer with Stateflow

In this section...
“Model Slicer Highlighting Behavior for Stateflow Elements” on page 8-49
“Using Model Slicer with Stateflow State Transition Tables” on page 8-49
“Support Limitations for Using Model Slicer with Stateflow” on page 8-49

You can use Model Slicer highlighting to visually verify the logic in your Stateflow charts or tables. After you “Define a Simulation Time Window” on page 8-12, you use Model Slicer to highlight and slice Stateflow elements that are active within the selected time window.

Note If you do not “Define a Simulation Time Window” on page 8-12 when you highlight functional dependencies in a Stateflow chart or table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.

In this section...
“Model Slicer Highlighting Behavior for Stateflow Elements” on page 8-49
“Using Model Slicer with Stateflow State Transition Tables” on page 8-49
“Support Limitations for Using Model Slicer with Stateflow” on page 8-49

Model Slicer Highlighting Behavior for Stateflow Elements

Model Slicer highlights a Stateflow element if it was executed in the specified time window. Some examples include:

- A chart, if it is activated in the specified a time window.
- A state, if its entry, exit, or during actions are executed in the specified a time window.
- A parent state, if its child state is highlighted in the specified a time window.
- A transition, if it is taken in the specified time window, such as inner, outer, and default. If the conditions of a transition are evaluated, but the transition is not taken, Model Slicer does not highlight the transition.

Using Model Slicer with Stateflow State Transition Tables

Model Slicer does not directly highlight the contents of Stateflow state transition tables. To view highlighted functional dependencies in a state transition table, you must view the auto-generated diagram for the state transition table. For instructions on how to view the auto-generated diagram for the state transition table, see “Generate Diagrams from State Transition Tables” (Stateflow).

Support Limitations for Using Model Slicer with Stateflow

For support limitations when you use Model Slicer with Stateflow, see “Model Slicer Support Limitations for Stateflow” on page 8-47.

See Also

More About

- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12
- “Chart Simulation Semantics” (Stateflow)

Isolating Dependencies of an Actuator Subsystem

This example demonstrates highlighting model items that a subsystem depends on. It also demonstrates generating a standalone model slice from the model highlight.

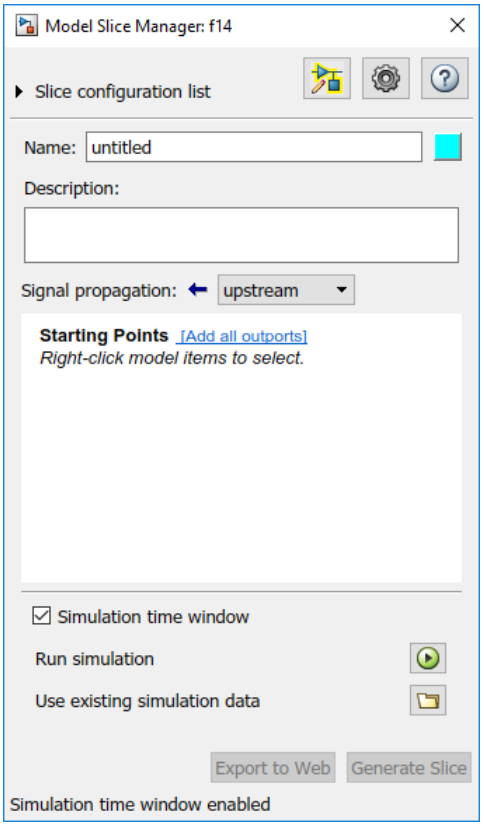
In this section...
“Choose Starting Points and Direction” on page 8-51
“View Precedents and Generate Model Slice” on page 8-52

Choose Starting Points and Direction

1 Open the `f14` example model.

```
f14
```

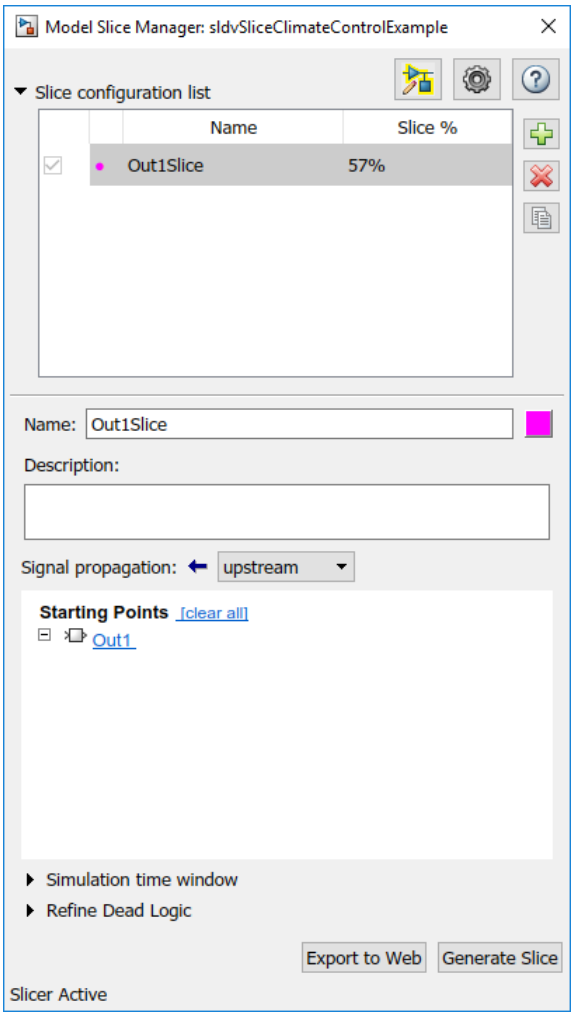
2 On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.



3 In the Model Slice Manager, click the arrow to expand the **Slice configuration list** list. Set the slice properties:

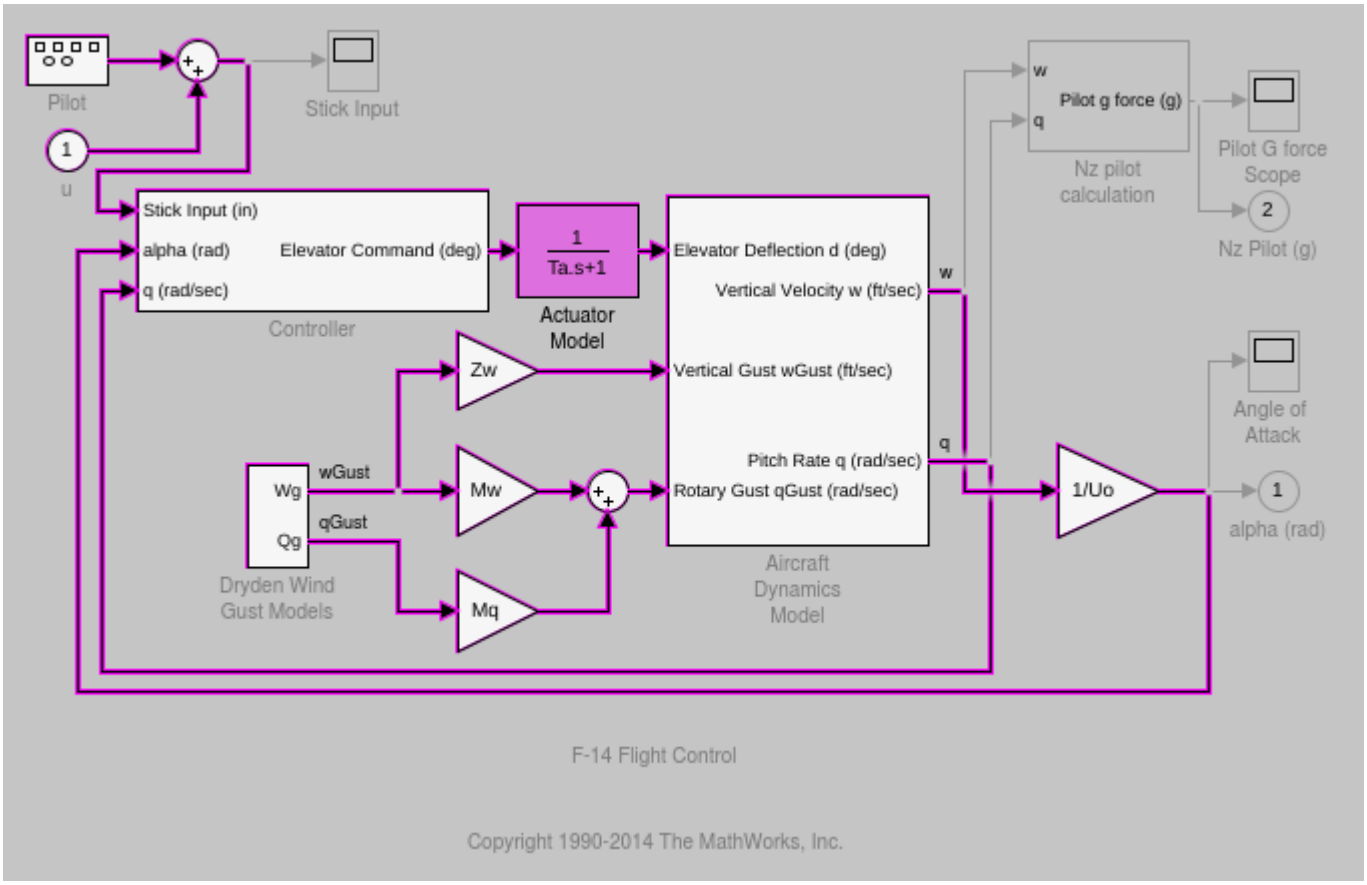
- **Name**: `Actuator_slice`
- To the right of **Name**, click the colored square to set the highlight color. Choose magenta from the palette.
- **Signal Propagation**: `upstream`.

4 Add the `Actuator Model` subsystem as a starting point. In the model, right-click the `Actuator Model` subsystem and select **Model Slicer > Add as Starting Point**.



View Precedents and Generate Model Slice

1 The model highlights the upstream dependencies of the `Actuator Model` subsystem.



Trace the following dependency path. `Aircraft Dynamics Model` is highlighted via the `Pitch Rate q` signal, which is an input to `Controller`, the output of which feeds `Actuator Model`.

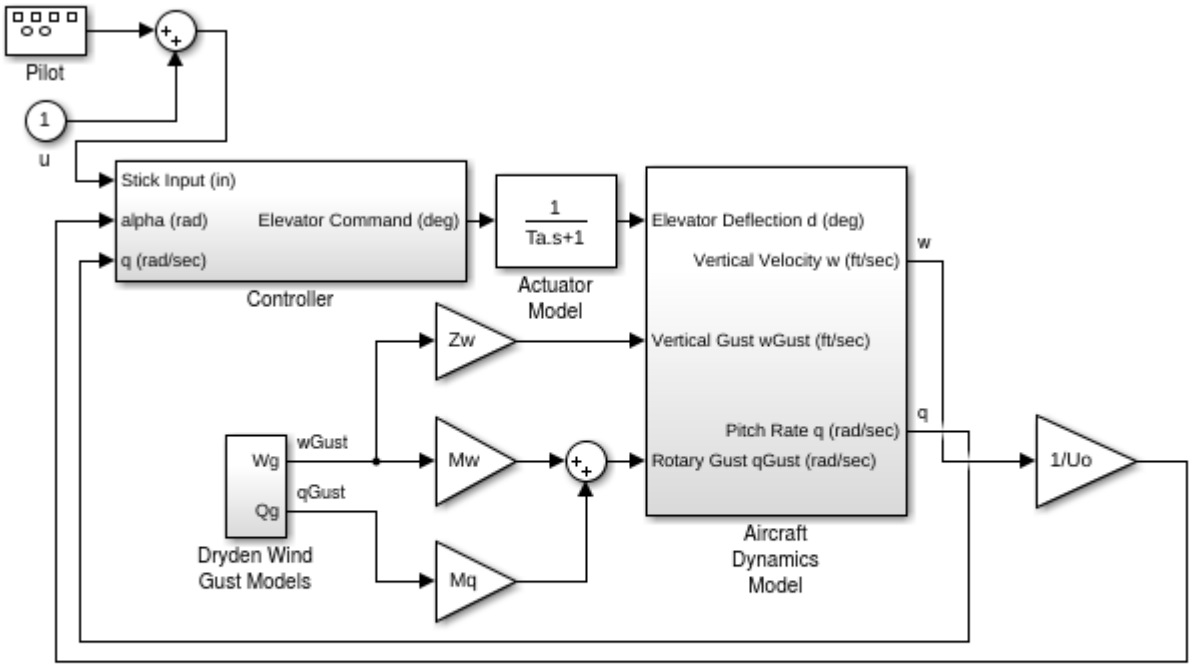
2. Generate a standalone model containing the highlighted model items:

   a. In the Model Slice Manager, click **Generate slice**.

   b. In the **Select File to Write** dialog box, select the save location and enter `actuator_slice_model`.

   c. Click **Save**.

3. The sliced model contains the highlighted model items.



F-14 Flight Control

Copyright 1990-2014 The MathWorks, Inc.

4 To remove highlighting from the model, close the Model Slice Manager.

Isolate Model Components for Functional Testing

You can create a standalone model for the model designed using “Design Model Architecture” (Simulink). The model slice isolates the model components and relevant signals for debugging and refinement.

Isolate Subsystems for Functional Testing

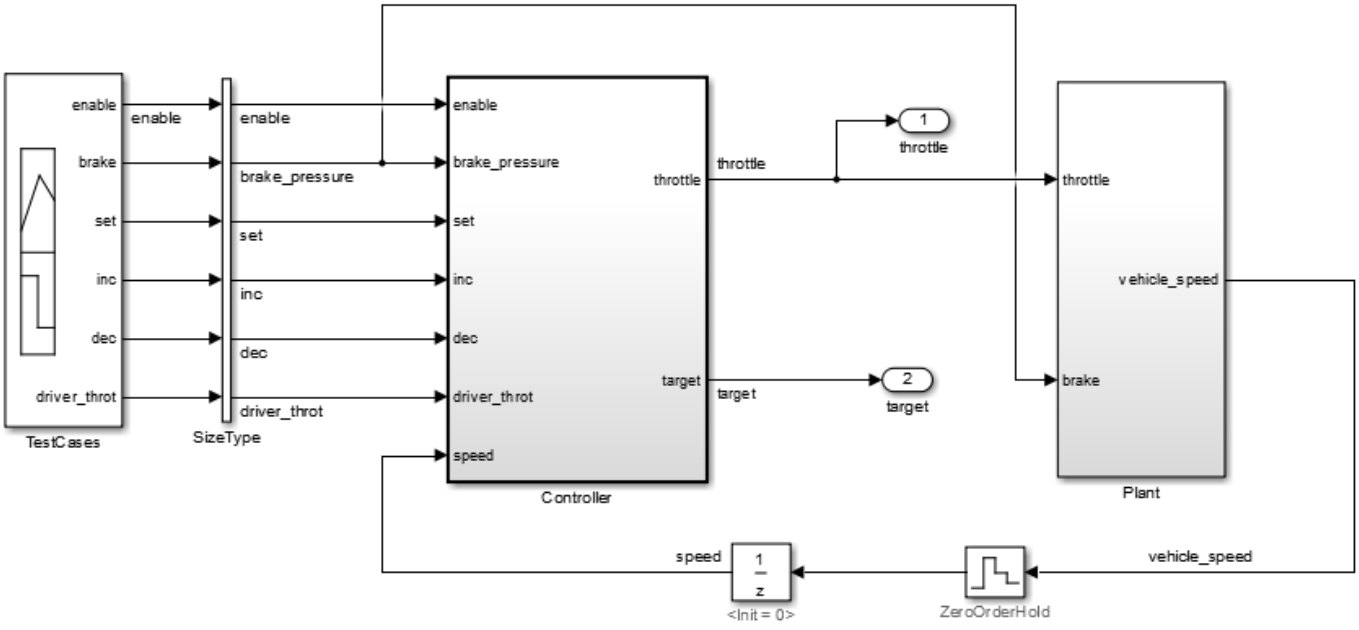
To debug and refine a subsystem of your model, create a standalone model. The standalone model isolates the subsystem and relevant signals. You can observe the subsystem behavior without simulating the entire source model.

Note You cannot slice virtual subsystems. To isolate a virtual subsystem, first convert it to an atomic subsystem.

Isolate a Subsystem with Simulation-Based Inputs

To observe the simulation behavior of a subsystem, include logged signal inputs in the standalone model. When you configure the model slice, specify a simulation time window. For large models, observing subsystem behavior in a separate model can save time compared to compiling and running the entire source model.

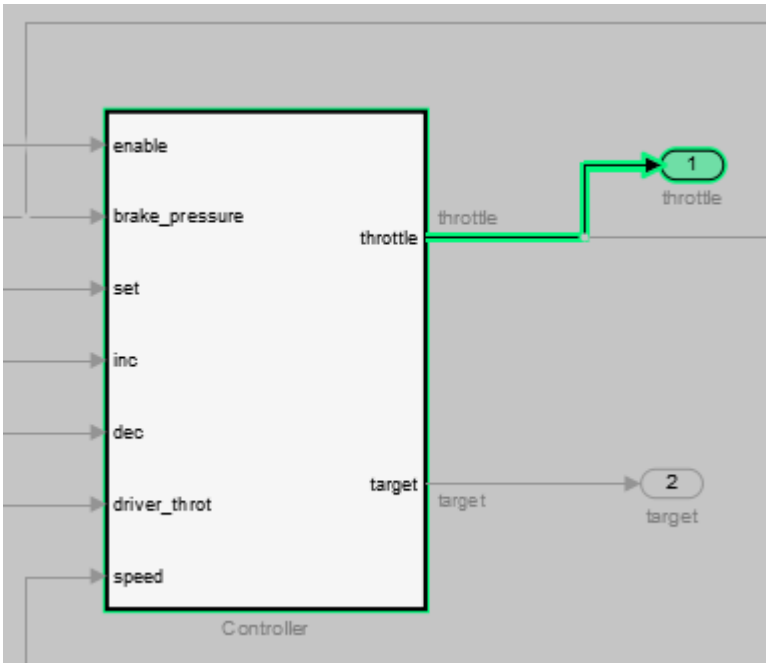
This example shows how to include simulation effects for the Controller subsystem of a cruise control system.



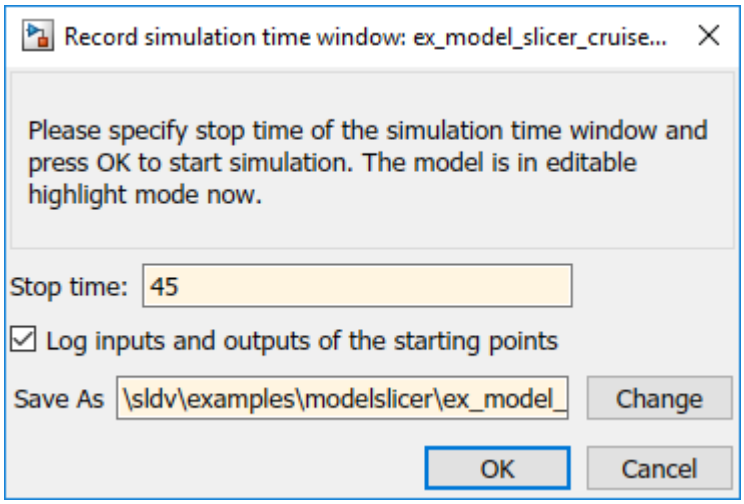
1 To open the Model Slice Manager, on the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

2. To select the starting point for dependency analysis, right-click a block, signal, or a port, and select **Model Slicer > Add as Starting point**.
3. To isolate the subsystem in the sliced model, right-click the subsystem, and select **Model Slicer > Slice component**.

   In the example model, selecting **Slice component** for the Controller subsystem limits the dependency analysis to the path between the starting point (the throttle outport) and the Controller subsystem.



4. To specify the simulation time window:
   a. In the Model Slice Manager, select **Simulation time window**.
   b. Click the run simulation button .
   c. Enter the simulation stop time, and click **OK**.

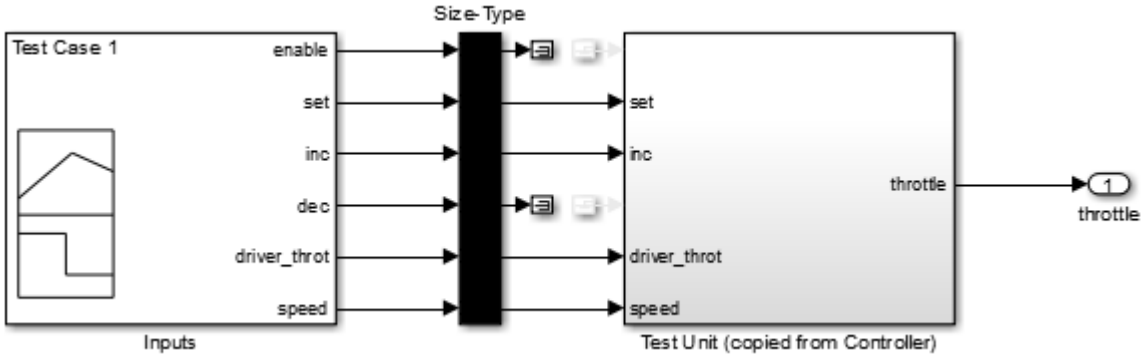


The Model slicer analyzes the model dependencies for the simulation interval.

5 To extract the subsystem and logged signals, click **Generate slice**. Enter a file name for the sliced model.

Based on the dependency analysis, a Signal Builder block supplies the signal inputs to the subsystem.

In the sliced model shown, the sliced model Signal Builder block contains one test case representing the signal inputs to the Controller subsystem for simulation time 0–45 seconds.



Isolate Referenced Model for Functional Testing

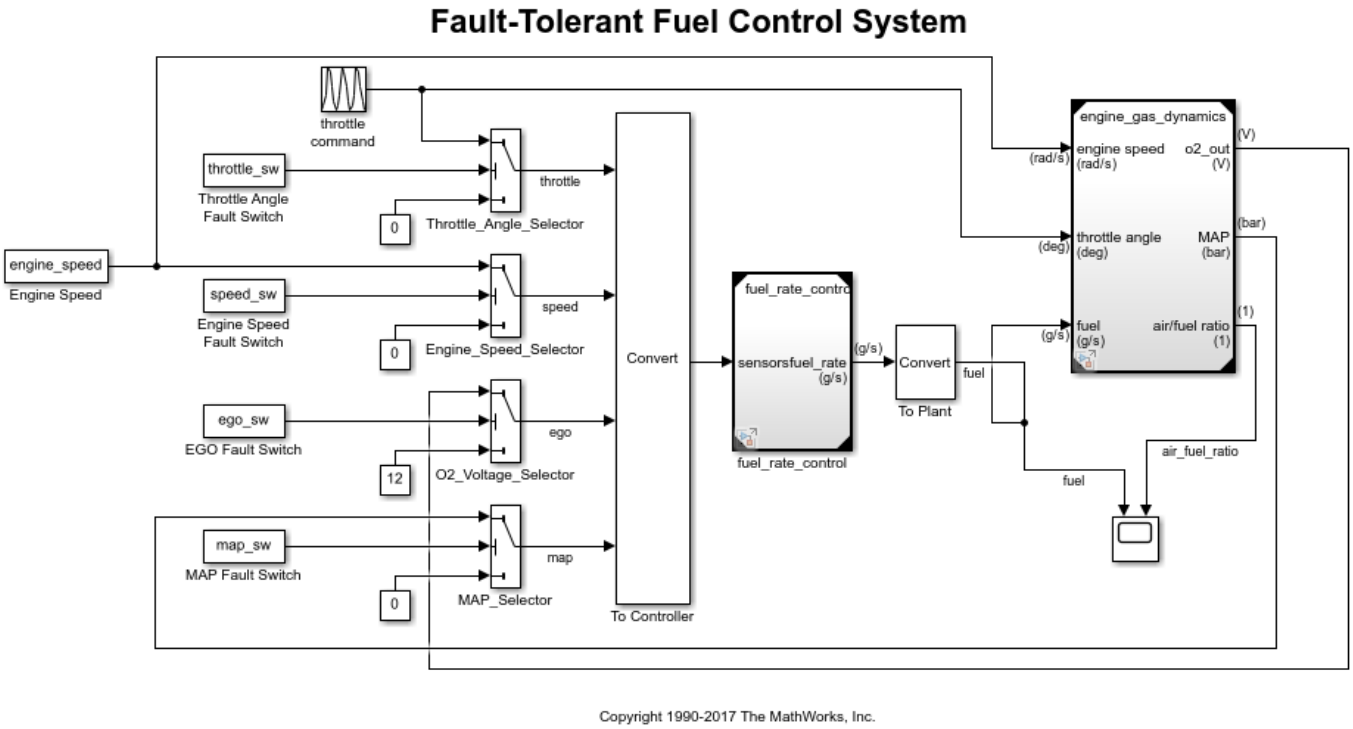
To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

This example shows how to slice the referenced model controller of a fault-tolerant fuel control system for functional testing. To create a simplified open-loop referenced model for debugging and refinement, you generate a slice of the referenced controller.

Step 1: Open the Model

The fault-tolerant fuel control system model contains a referenced model controller `fuel_rate_control`.

```
open_system('sldvSlicerdemo_fuelsys');
```



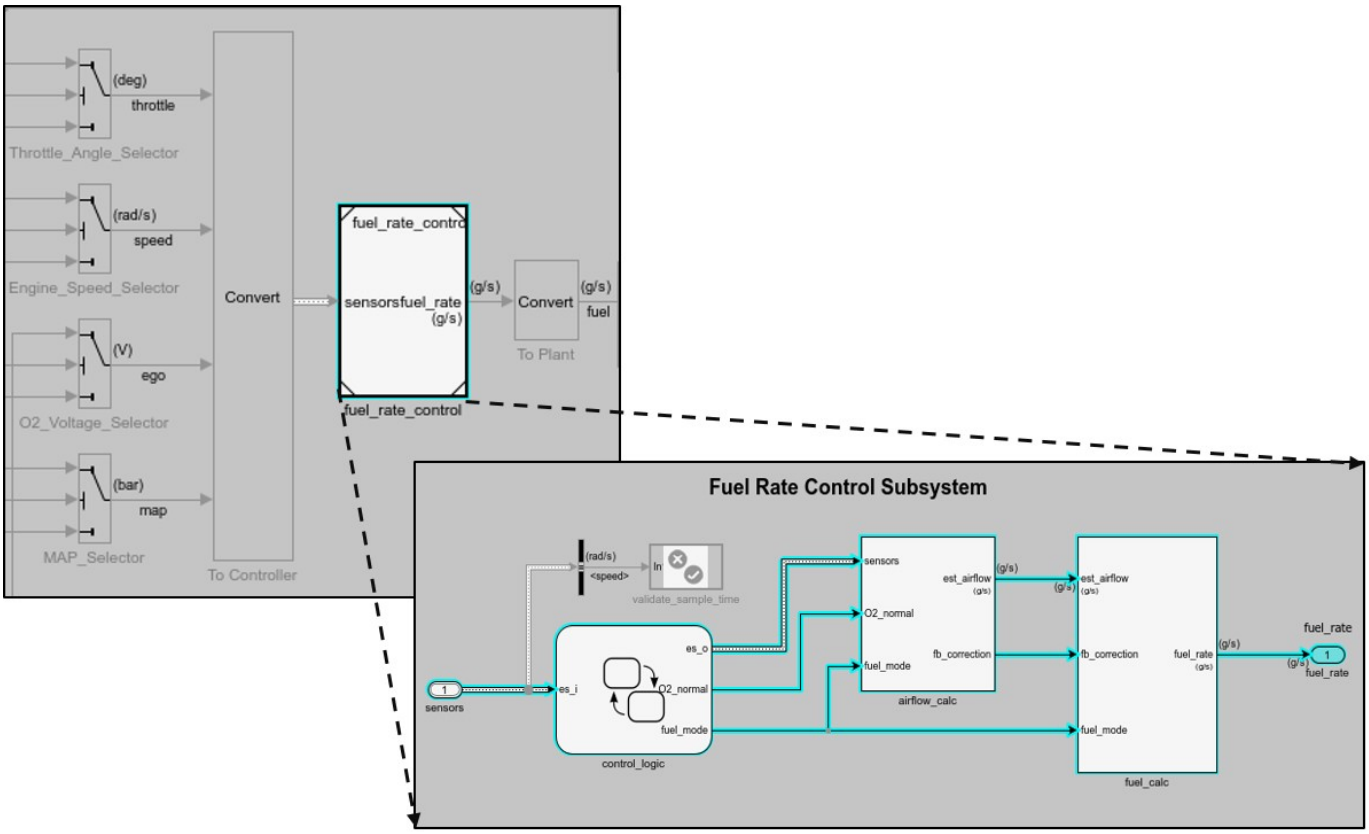
Step 2: Slice the Referenced Model

To analyze the `fuel_rate_control` referenced model, you slice it to create a standalone open-loop model. To open the Model Slice Manager, select **Apps** > **Model Verification, Validation, and Test** > **Model Slicer**, or right-click the `fuel_rate_control` model and select **Model Slicer** > **Slice component**. When you open the Model Slice Manager, the Model Slicer compiles the model. You then configure the model slice properties.

Note: The simulation mode of the `sldvSlicerdemo_fuelsys` model is `Accelerator` mode. When you slice the referenced model, the software configures the simulation mode to `Normal` mode and sets it back to its original simulation mode while exiting the Model Slicer.

Step 3: Select Starting Point

Open the `fuel_rate_control` model, right-click the `fuel-rate` port, and select **Model Slicer** > **Add as starting point**. The Model Slicer highlights the upstream constructs that affect the `fuel_rate`.



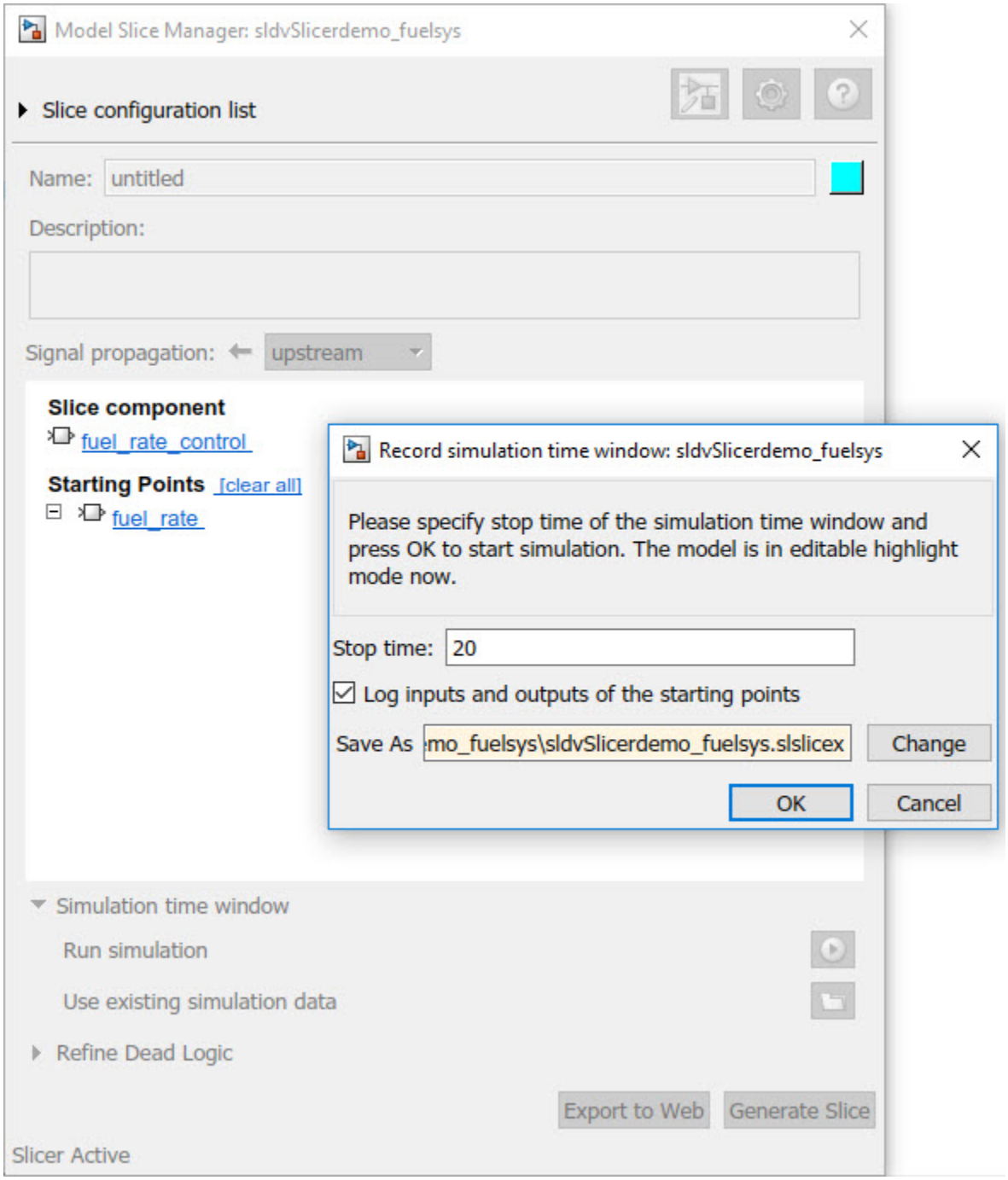
Step 4: Generate Slice

a. In the Model Slice Manager dialog box, select the **Simulation time window**.

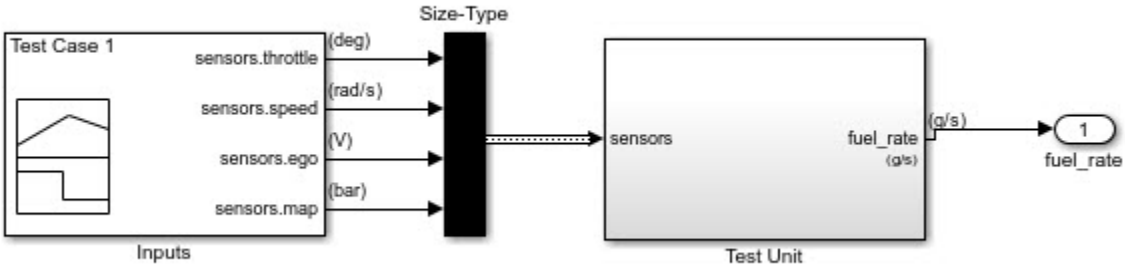
b. Click **Run simulation**.

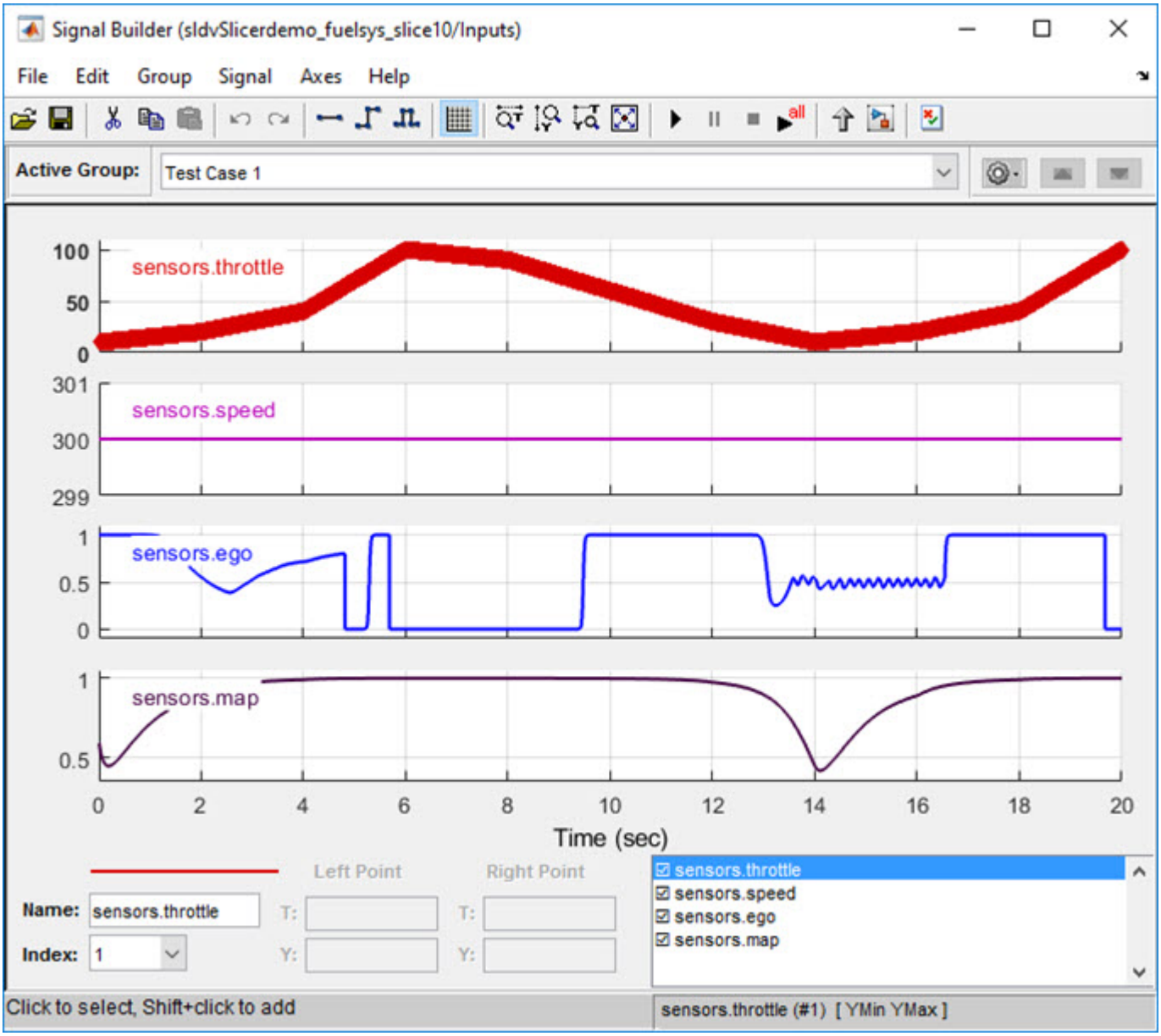
c. For the **Stop time**, enter `20`. Click **OK**.

d. Click **Generate Slice**. The software simulates the sliced referenced model by using the inputs of the close-loop `sldvSlicerdemo_fuelsys` model.



For the sliced model, in the Signal Builder window, one test case is displayed that represents the signals input to the referenced model for simulation time 0–20 seconds.


Size-Type
Test Case 1
sensors.throttle
(deg)
sensors.speed
(rad/s)
sensors.ego
(V)
sensors.map
(bar)
Inputs
sensors
fuel_rate
(g/s)
Test Unit
(g/s)
1
fuel_rate



See Also

“Model Slicer Considerations and Limitations” on page 8-43 | “Highlight Functional Dependencies” on page 8-2

Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results

When you run simulation or refine dead logic, Model Slicer saves your simulation results at the default location `<current_folder>\modelslicer\<model_name>\<model_name>.slslicex`. For large or complex models, the simulation time can be lengthy. To refine the highlighted slice, you can use the existing Model Slicer simulation data or dead logic results.

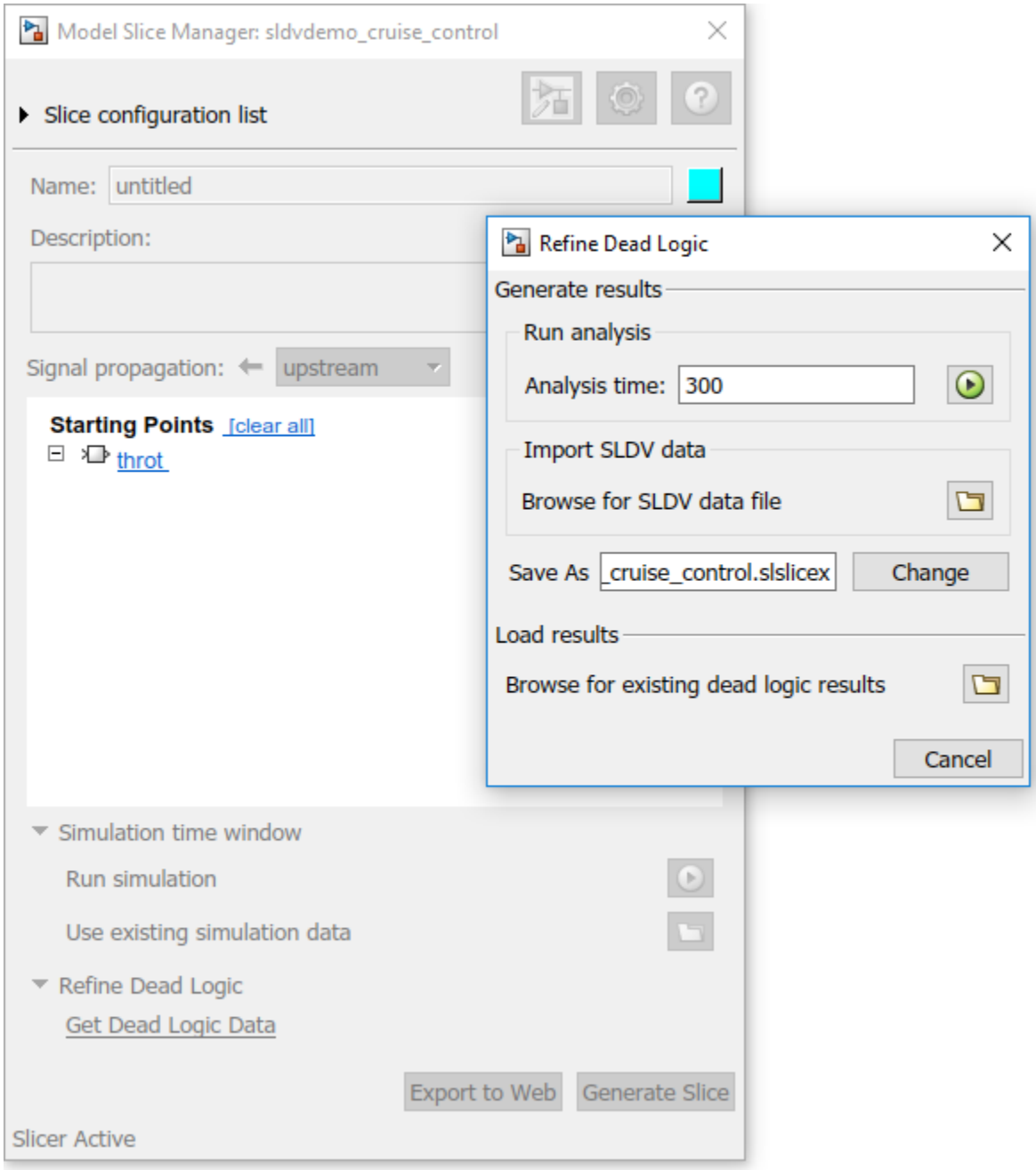
If you want to highlight functional dependencies in the model again at another time, you can use the existing `.slslicex` simulation time window data without needing to resimulate the model. Model Slicer then uses the existing simulation data to highlight the model.

1. Open the Simulink model.
2. To open the Model Slice Manager, On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
3. Select **Simulation time window**.
4. Click **Use existing simulation data** .
5. Navigate to the existing `.slslicex` data and click **Open**.

To refine the dead logic for dependency analysis, you can import the existing Simulink Design Verifier data file or use the existing `.slslicex` dead logic results. For more information see, “Dead Logic Detection” (Simulink Design Verifier) and “Simulink Design Verifier Data Files” (Simulink Design Verifier).

1. In Model Slice Manager, select **Refine Dead Logic** and click **Get Dead Logic Data**.
2. To import the Simulink Design Verifier data file, click **Browse for SLDV data file** .

   To load the existing dead logic results, click **Browse for existing dead logic results** .
3. Navigate to the existing data and click **Open**.



See Also

More About

- “Highlight Functional Dependencies” on page 8-2
- “Configure Model Highlight and Sliced Models” on page 8-40
- “Refine Dead Logic for Dependency Analysis” on page 8-22

Simplification of Variant Systems

In this section...
“Use the Variant Reducer to Simplify Variant Systems” on page 8-65
“Use Model Slicer to Simplify Variant Systems” on page 8-65

If your model contains “Variant Systems” (Simulink), you can reduce the model to a simplified, standalone model containing only selected variant configurations.

Use the Variant Reducer to Simplify Variant Systems

After you Add and Validate Variant Configurations (Simulink), you can reduce the model from the Variant Manager:

1. Open a model containing at least one valid variant configuration.
2. Select **View >> Variant Manager**, or right-click a variant system and select **Variant >> Open in Variant Manager**.
3. Click **Reduce model...**.
4. Select one or more variant configurations.
5. Set the **Output directory**.
6. Click **Reduce** to create a simplified, standalone model containing only the selected variant configurations.

The Variant Reducer creates a simplified, standalone model in the output directory you specified containing only the variant configurations you selected. For more information, see “Reduce Models Containing Variant Blocks” (Simulink).

Use Model Slicer to Simplify Variant Systems

After you Add and Validate Variant Configurations (Simulink), you can use Model Slicer to create a simplified, standalone model containing only the active variant configuration. When you “Highlight Functional Dependencies” on page 8-2 in a model containing variant systems, only active variant choices are highlighted. When you “Create a Simplified Standalone Model” on page 8-28 from a model highlight that includes variant systems, Model Slicer removes the variant systems and replaces them with the active variant configurations.

For instructions on how to change the active variant configuration and how to set default variant choices, see “Working with Variant Choices” (Simulink).

See Also

More About

- “Create a Simple Variant Model” (Simulink)
- “Define, Configure, and Activate Variants” (Simulink)
- “Introduction to Variant Controls” (Simulink)
- “Reduce Models Containing Variant Blocks” (Simulink)

Programmatically Resolve Unexpected Behavior in a Model with Model Slicer

In this example, you evaluate a Simulink® model, detect unexpected behavior, and use Model Slicer to programmatically isolate and resolve the unexpected behavior. When you plan to reuse your API commands and extend their use to other models, a programmatic approach is useful.

Prerequisites

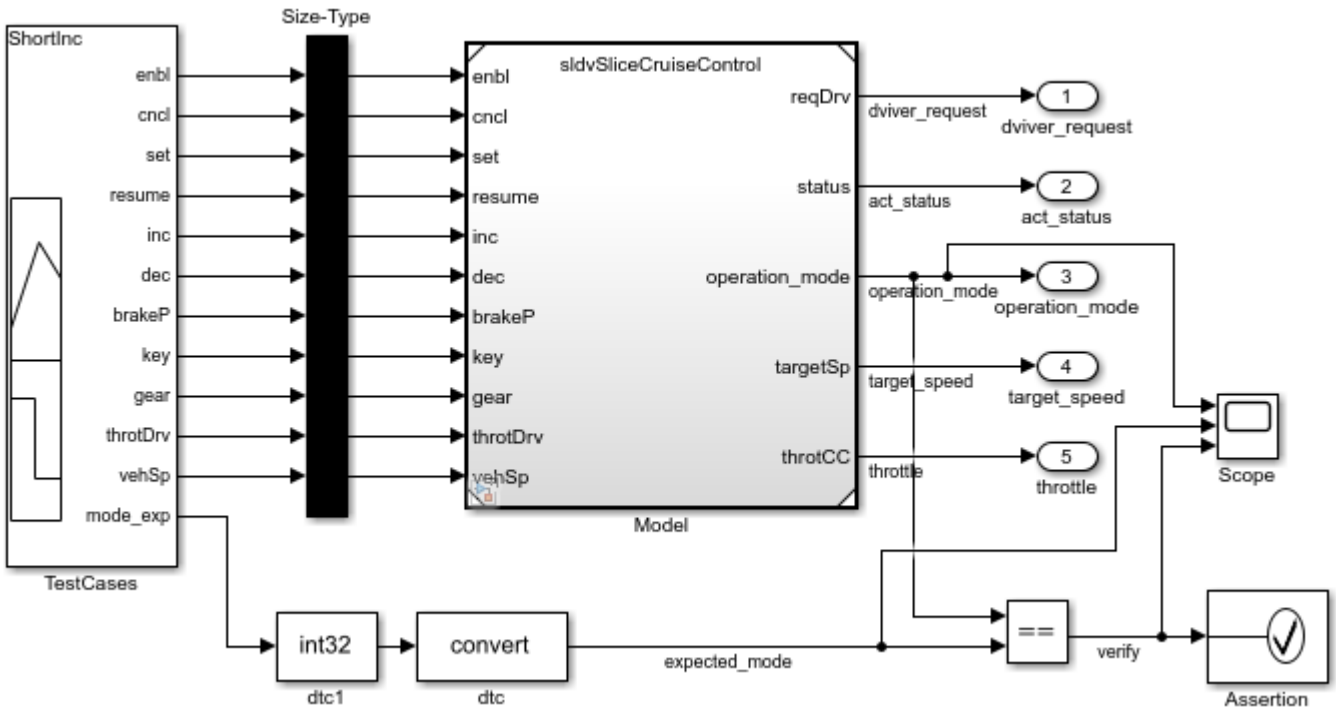
Be familiar with the behavior and purpose of Model Slicer and the functionality of the Model Slicer API. “Highlight Functional Dependencies” on page 8-2 outlines how to use Model Slicer user interface to explore models. The `slslicer`, `slsliceroptions`, and `slslicertrace` function reference pages contain the Model Slicer API command help.

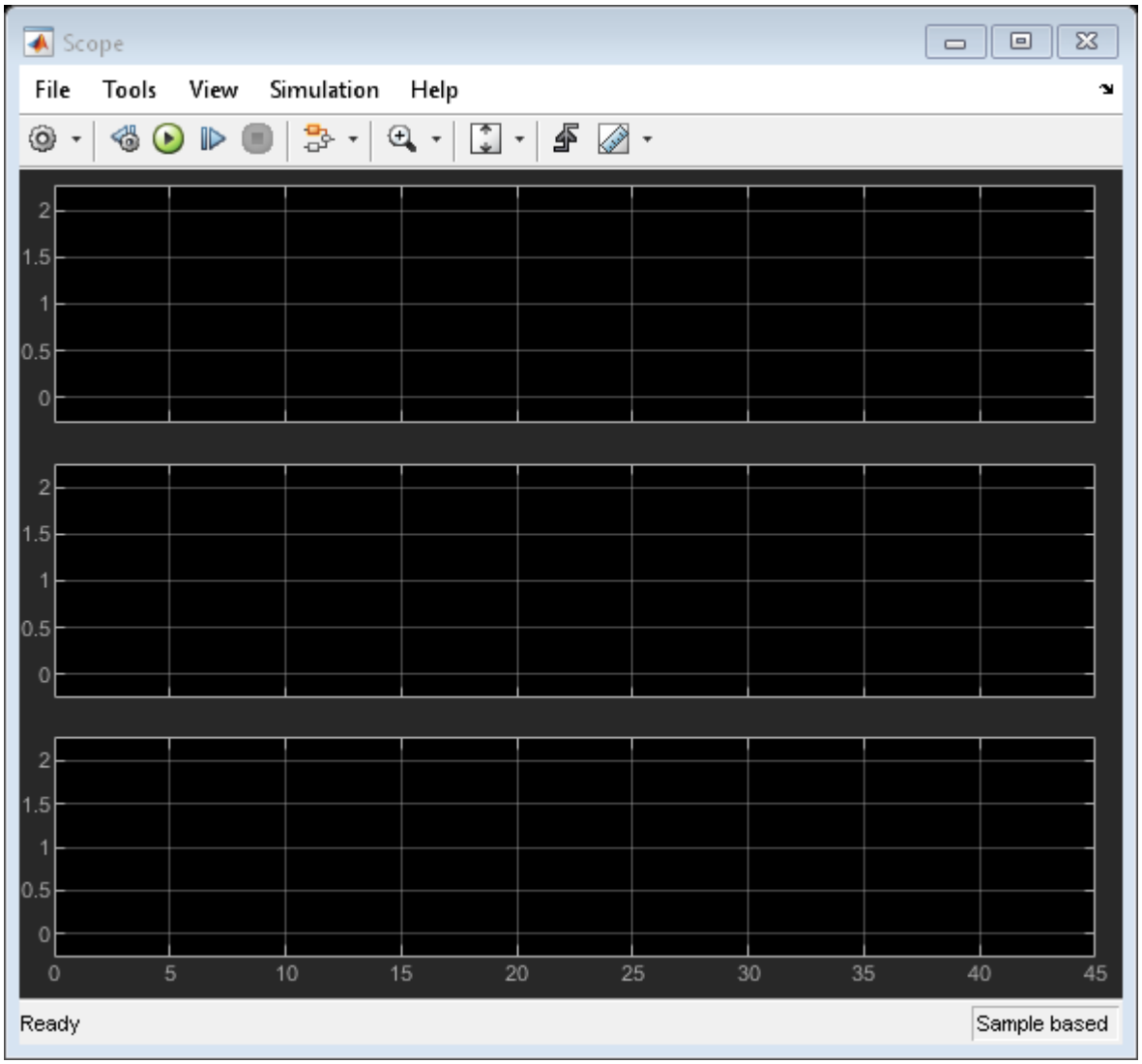
Find the Area of the Model Responsible for Unexpected Behavior

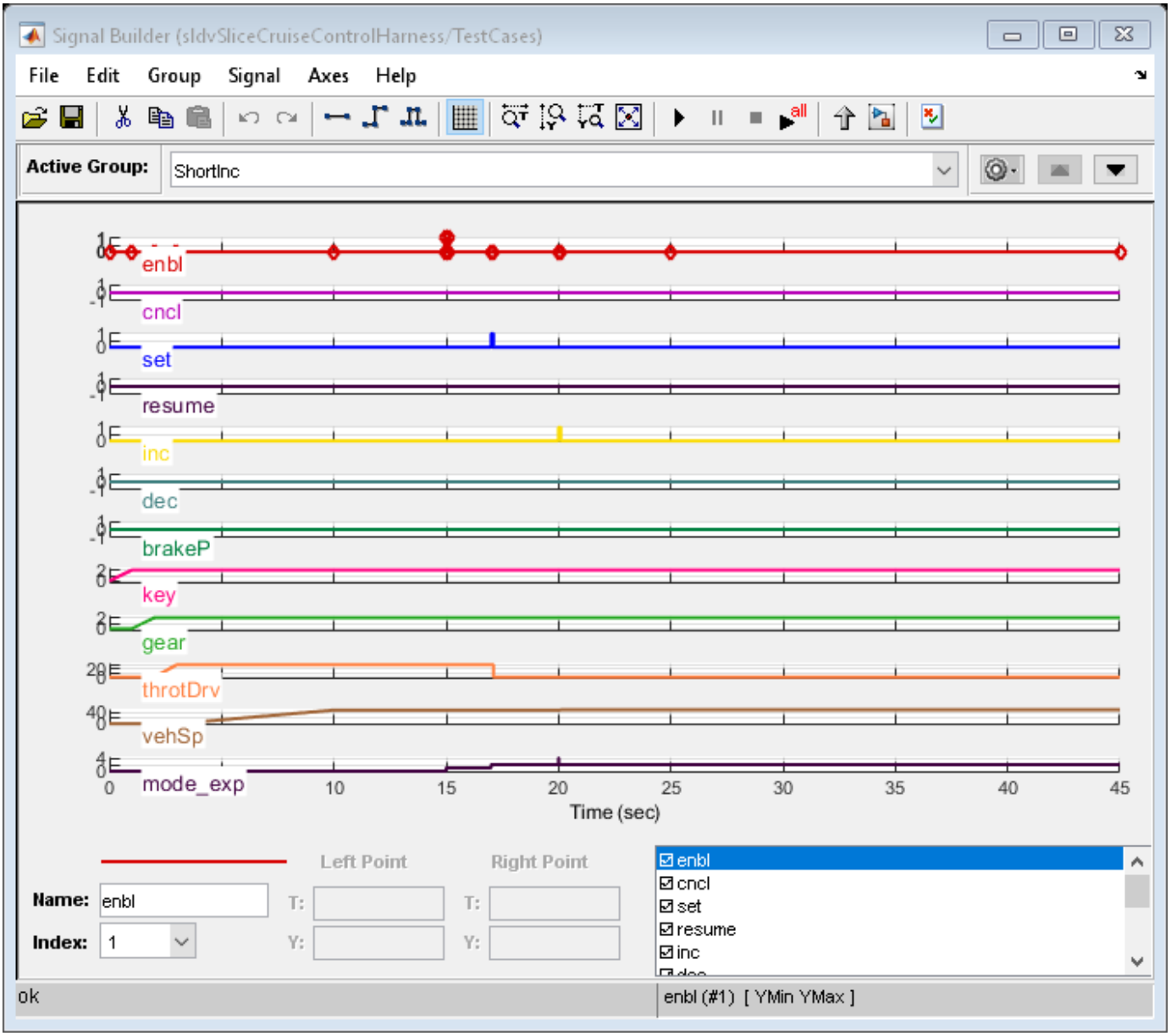
The **sldvSliceCruiseControlHarness** test harness model contains a cruise controller subsystem **sldvSliceCruiseContro*l and a block, *TestCases**, containing a test case for this subsystem. You first simulate the model to execute the test case. You then evaluate the behavior of the model to find and isolate areas of the model responsible for unexpected behavior:

1. Open the **sldvSliceCruiseControlHarness** test harness for the cruise control model.

```
open_system('sldvSliceCruiseControlHarness')
```




Scope
File
Tools
View
Simulation
Help
2
1.5
1
0.5
0
0
5
10
15
20
25
30
35
40
45
Ready
Sample based



Note: The Assertion block is set to **Stop simulation when assertion fails** when the actual operation mode is not the same as the expected operation mode.

The TestCases block contains several test inputs for sldvSliceCruiseControl.

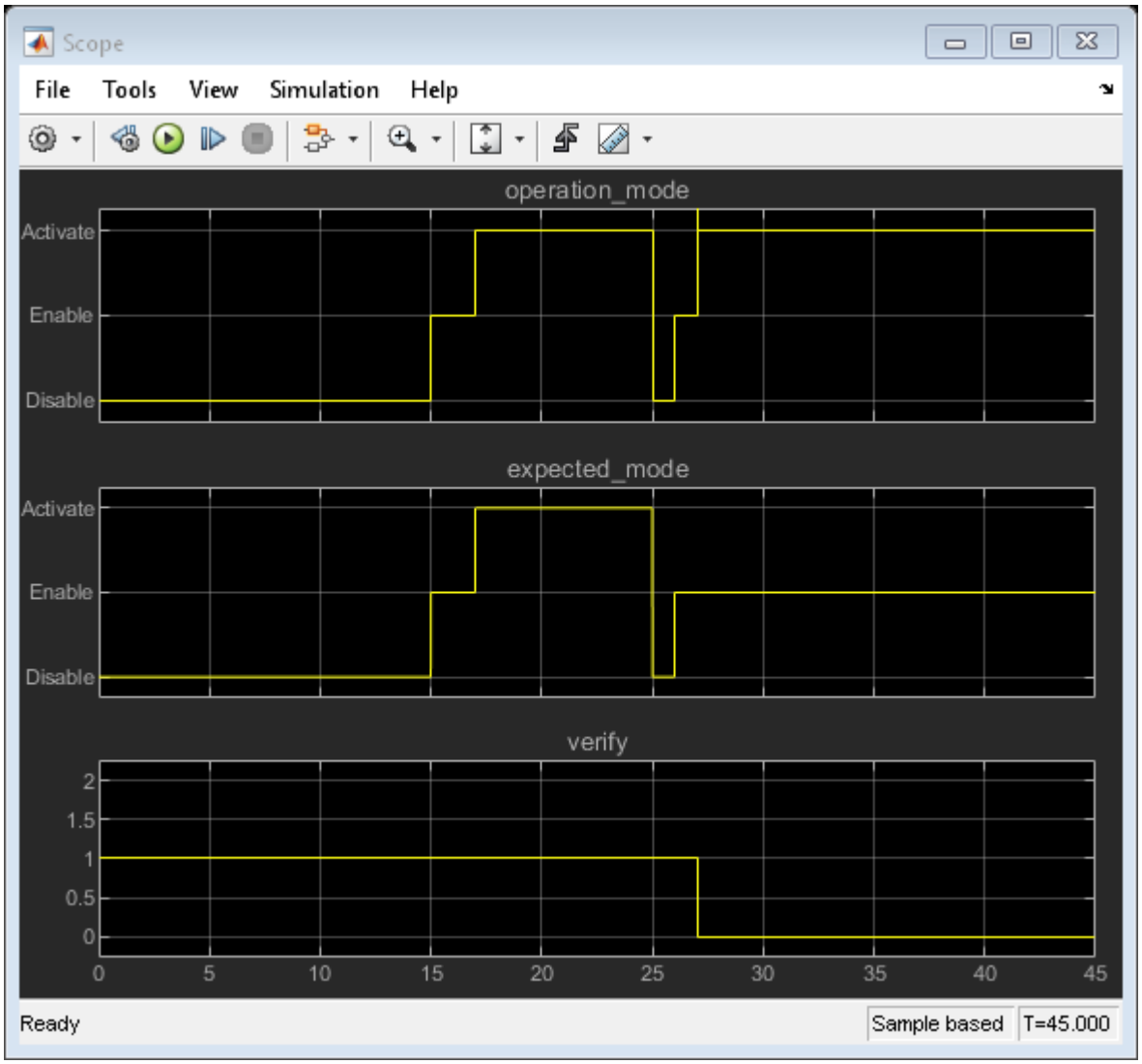
2. In the TestCases Signal Builder click the **Run all** button to run all of the included test cases. You receive an error during the ResumeWO test case. The Assertion block halted simulation at 27 seconds, because the actual operation mode was not the same as the expected operation mode. Click **OK** to close this error message.

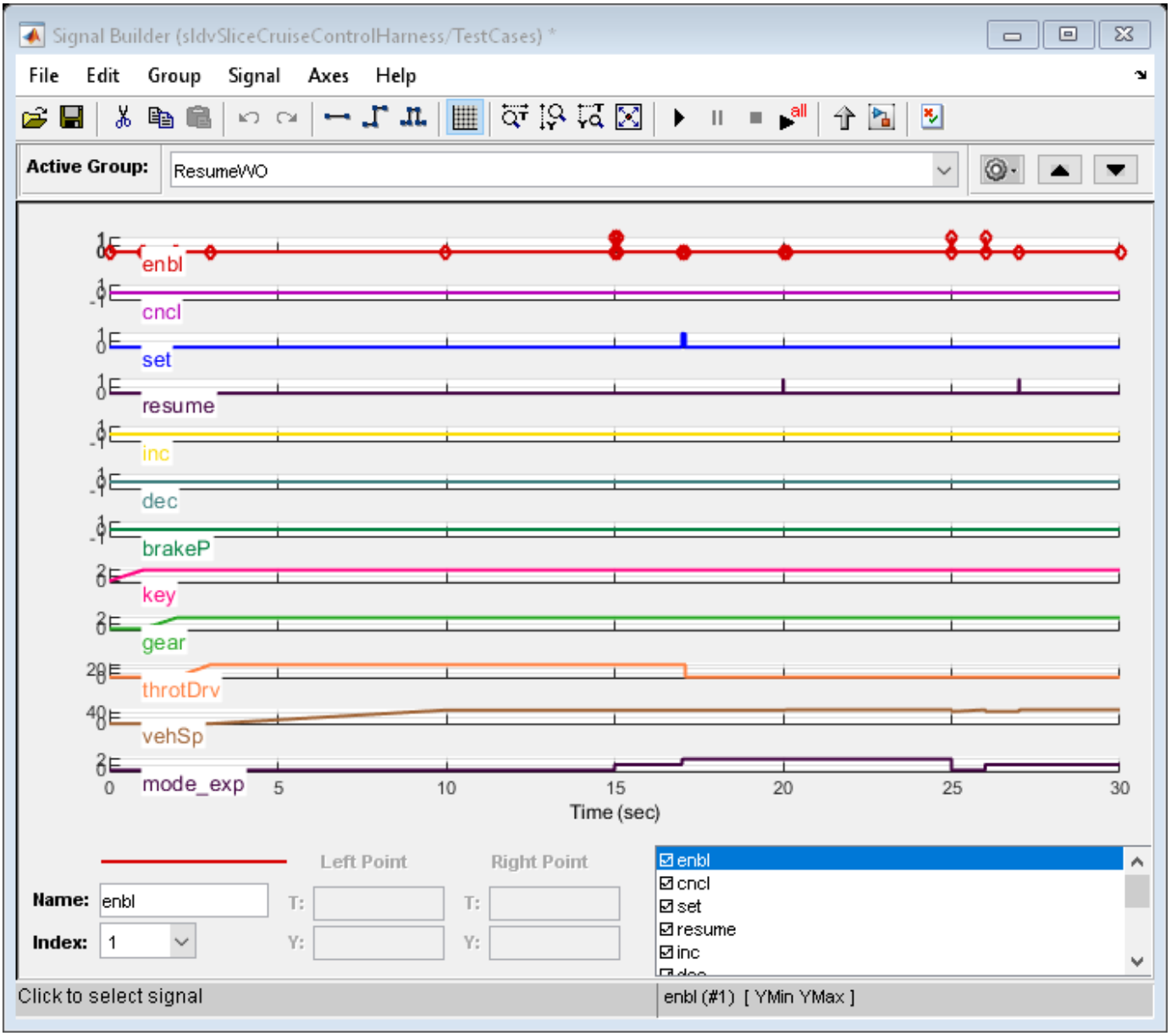
3. In the sldvSliceCruiseControlHarness model, double-click the Assertion block, clear **Enable assertion**, and click **OK**.

```
set_param('sldvSliceCruiseControlHarness/Assertion','Enabled','off')
```

4. Set the **Active Group** of the *TestCases* Signal Builder block to the test case containing the error and run the simulation again.

```
signalbuilder('sldvSliceCruiseControlHarness/TestCases', 'ACTIVEGROUP', 12)
sim('sldvSliceCruiseControlHarness')
```





The Scope block in the model contains three signals:

- *operation_mode* - displays the actual operation mode of the subsystem.
- *expected_mode* - displays the expected operation mode of the subsystem that the test case provides.
- *verify* - displays a Boolean value comparing the operation mode and the expected mode.

The scope shows a disparity between the expected operation mode and the actual operation mode beginning at time **27**. Now that you know the outport displaying the unexpected behavior and the time window containing the unexpected behavior, use Model Slicer to isolate and analyze the unexpected behavior.

Isolate the Area of the Model Responsible for Unexpected Behavior

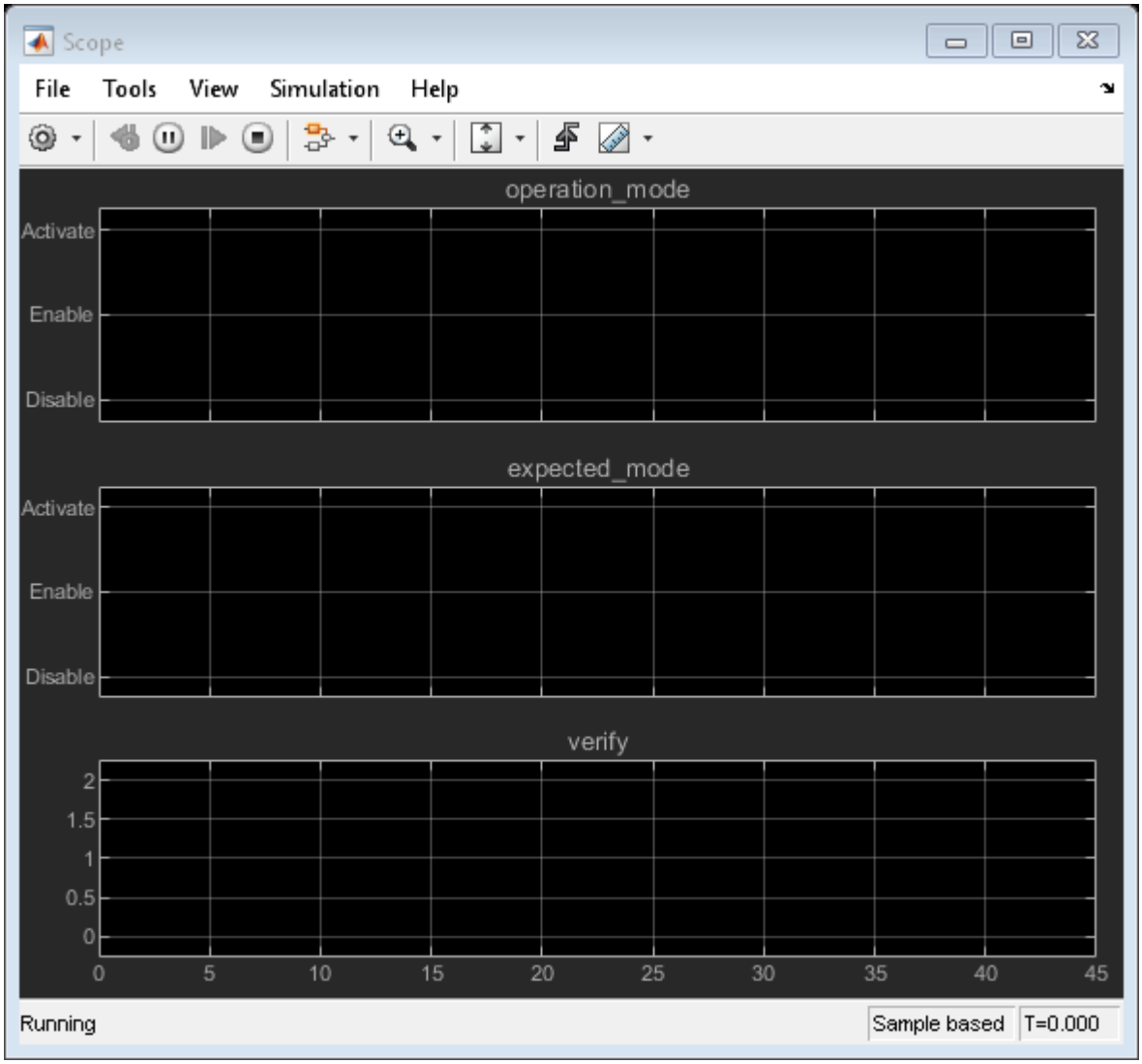
1. Create a Model Slicer configuration object for the model using `slslicer`. The Command Window displays the slice properties for this Model Slicer configuration.

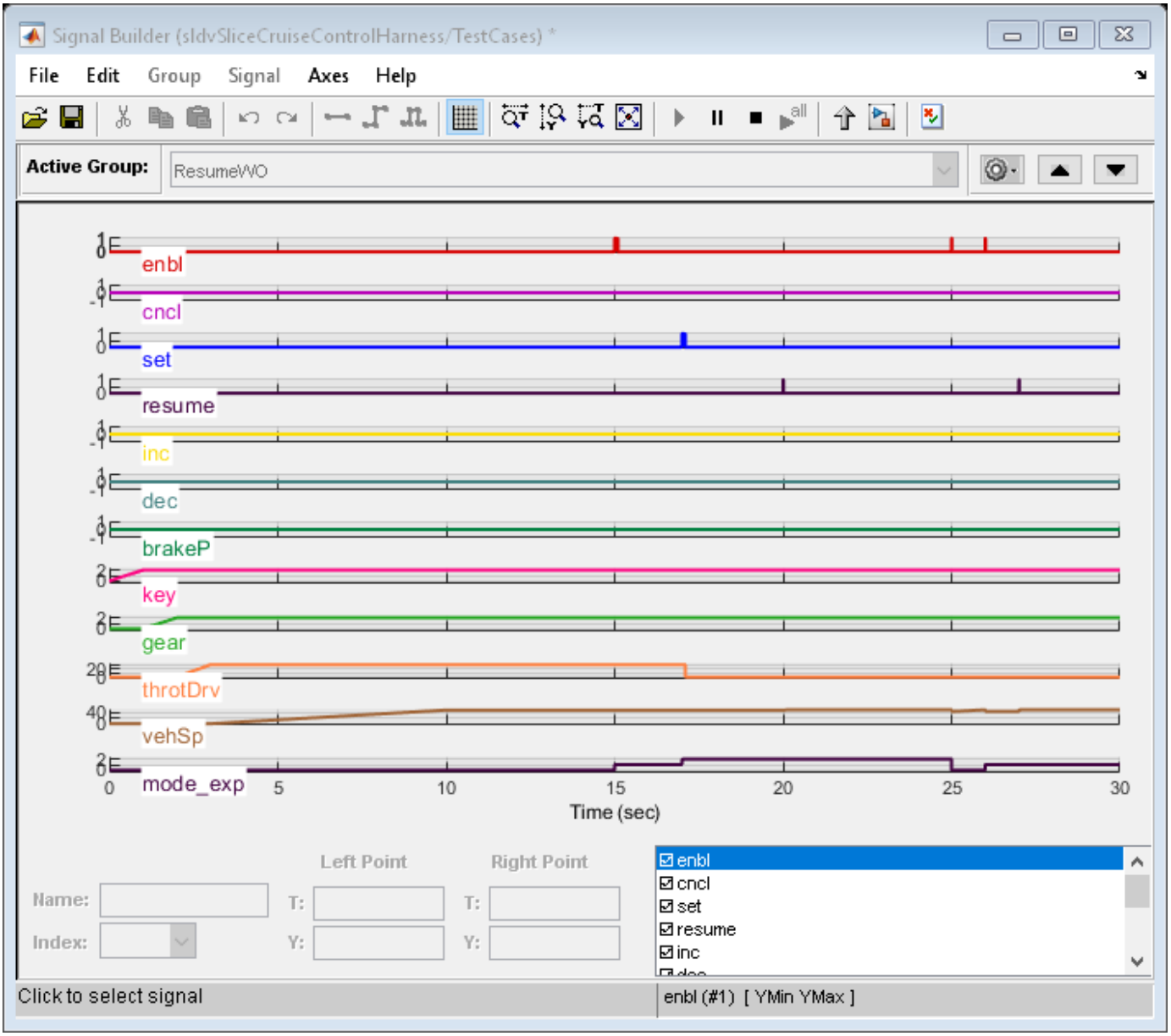
```
obj = slslicer('sldvSliceCruiseControlHarness')
```

```
obj =

  SLSlicer with properties:

       Configuration: [1x1 SLSlicerAPI.SLSlicerConfig]
        ActiveConfig: 1
     DisplayedConfig: []
      StorageOptions: [1x1 struct]
     AnalysisOptions: [1x1 struct]
        SliceOptions: [1x1 struct]
       InlineOptions: [1x1 struct]

   Contents of active configuration:
                Name: 'untitled'
         Description: ''
               Color: [0 1 1]
    SignalPropagation: 'upstream'
        StartingPoint: [1x0 struct]
       ExclusionPoint: [1x0 struct]
           Constraint: [1x0 struct]
       SliceComponent: [1x0 struct]
        UseTimeWindow: 0
         CoverageFile: ''
         UseDeadLogic: 0
        DeadLogicFile: ''
```

2. Activate the slice highlighting mode of Model Slicer to compile the model and prepare it for dependency analysis.

```
activate(obj)
```


Scope
File
Tools
View
Simulation
Help
operation_mode
Activate
Enable
Disable
expected_mode
Activate
Enable
Disable
verify
2
1.5
1
0.5
0
0
5
10
15
20
25
30
35
40
45
Running
Sample based
T=0.000



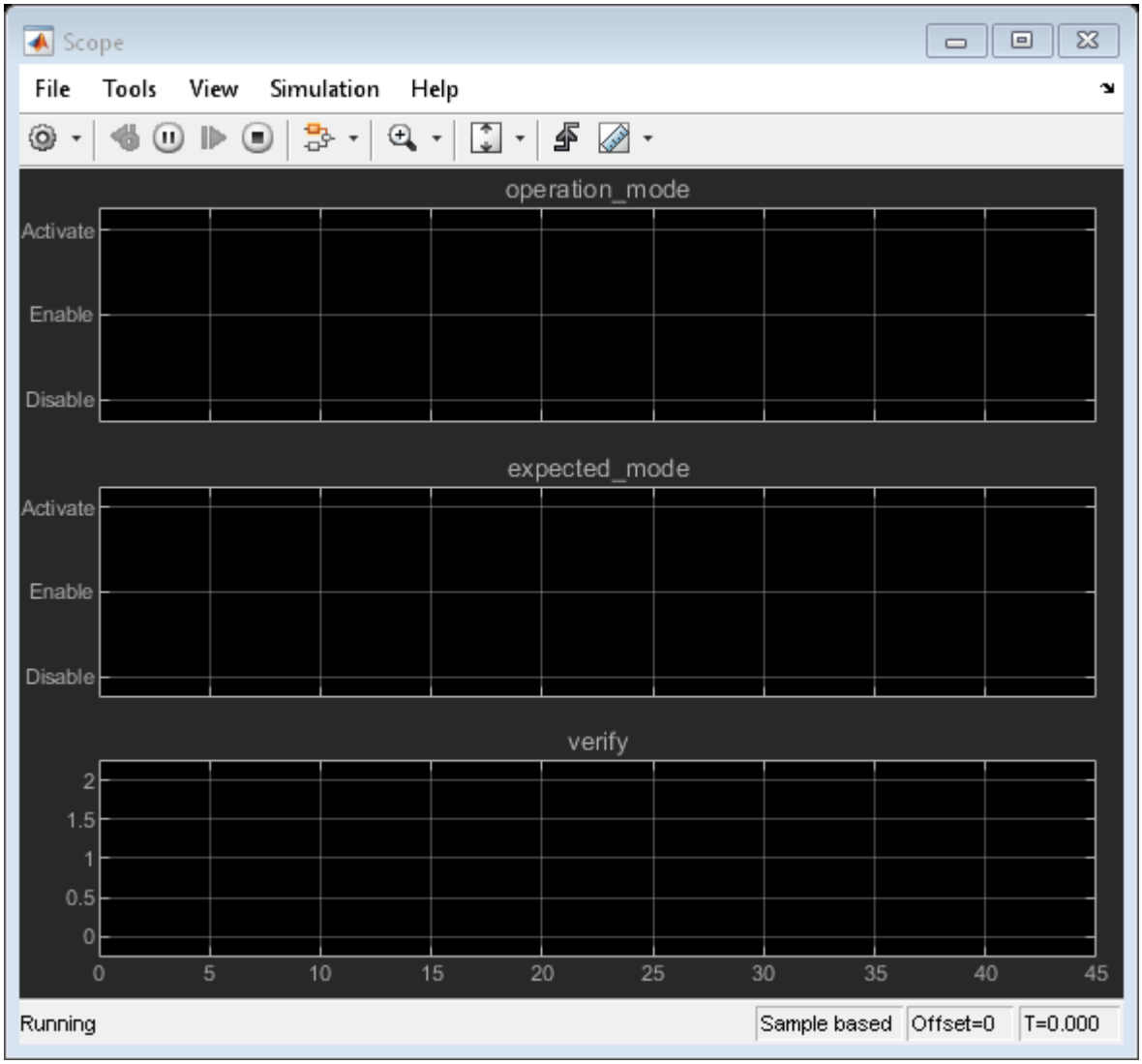
3. Add the **operation_mode** outport block as a starting point and highlight it.

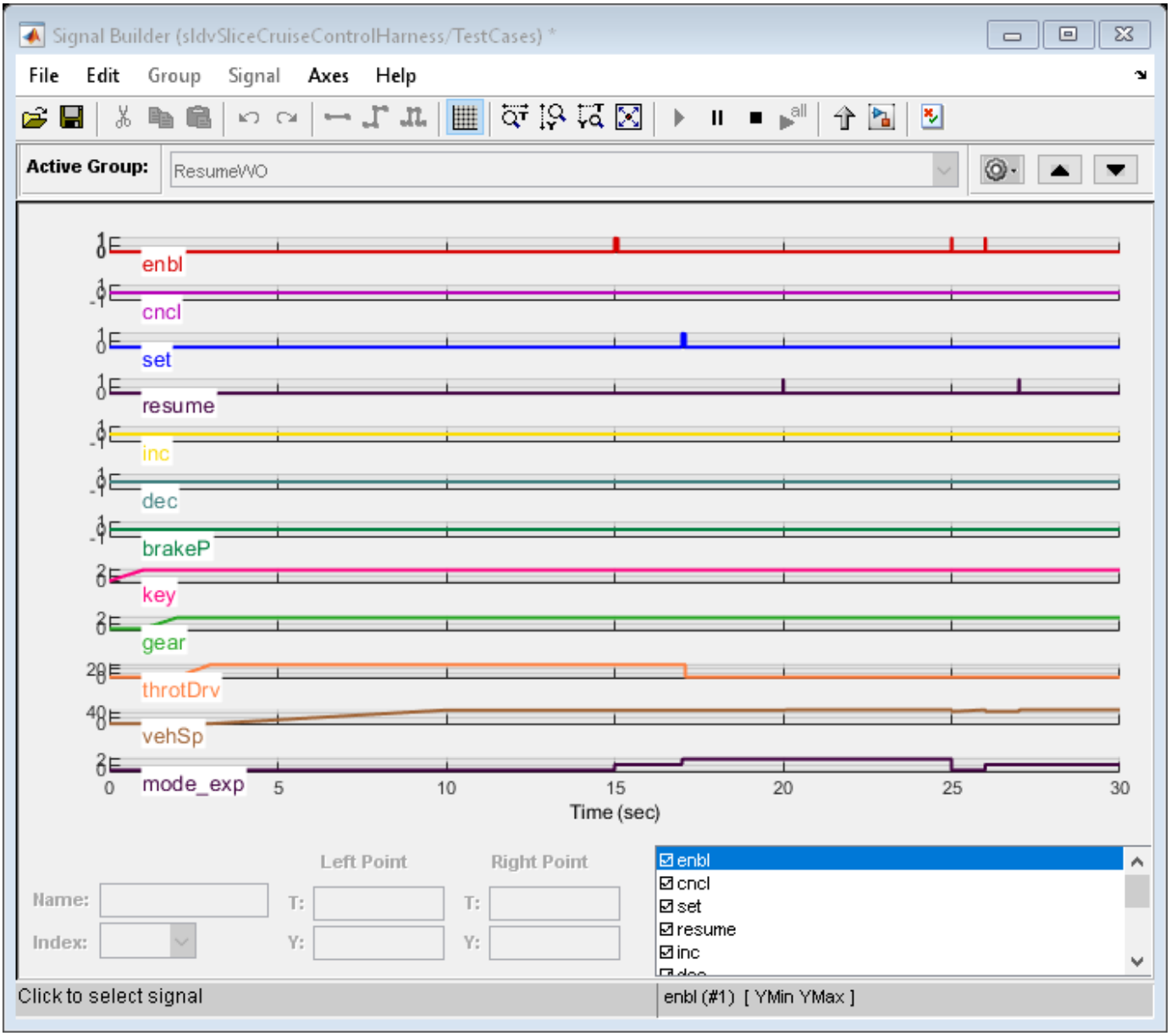
```
addStartingPoint(obj,'sldvSliceCruiseControlHarness/operation_mode')
highlight(obj)
```

The area of the model upstream of the starting point and active during simulation is highlighted.

4. Simulate the model within a restricted simulation time window (maximum 30 seconds) to highlight only the area of the model upstream of the starting point and active during the time window of interest.

```
simulate(obj,0,30)
```


Scope
File
Tools
View
Simulation
Help
operation_mode
Activate
Enable
Disable
expected_mode
Activate
Enable
Disable
verify
2
1.5
1
0.5
0
0
5
10
15
20
25
30
35
40
45
Running
Sample based
Offset=0
T=0.000



Only the portion of the model upstream of the starting point and active during the simulation time window is highlighted.

5. You can further narrow the simulation time window by changing the start time to 20 seconds.

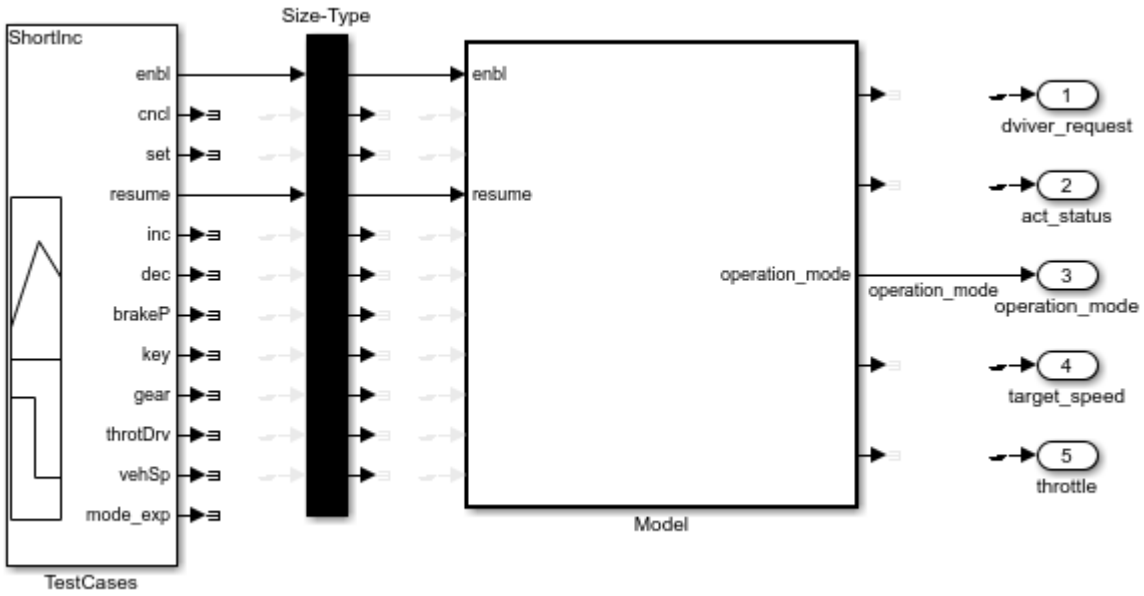
```
setTimeWindow(obj,20,30)
```

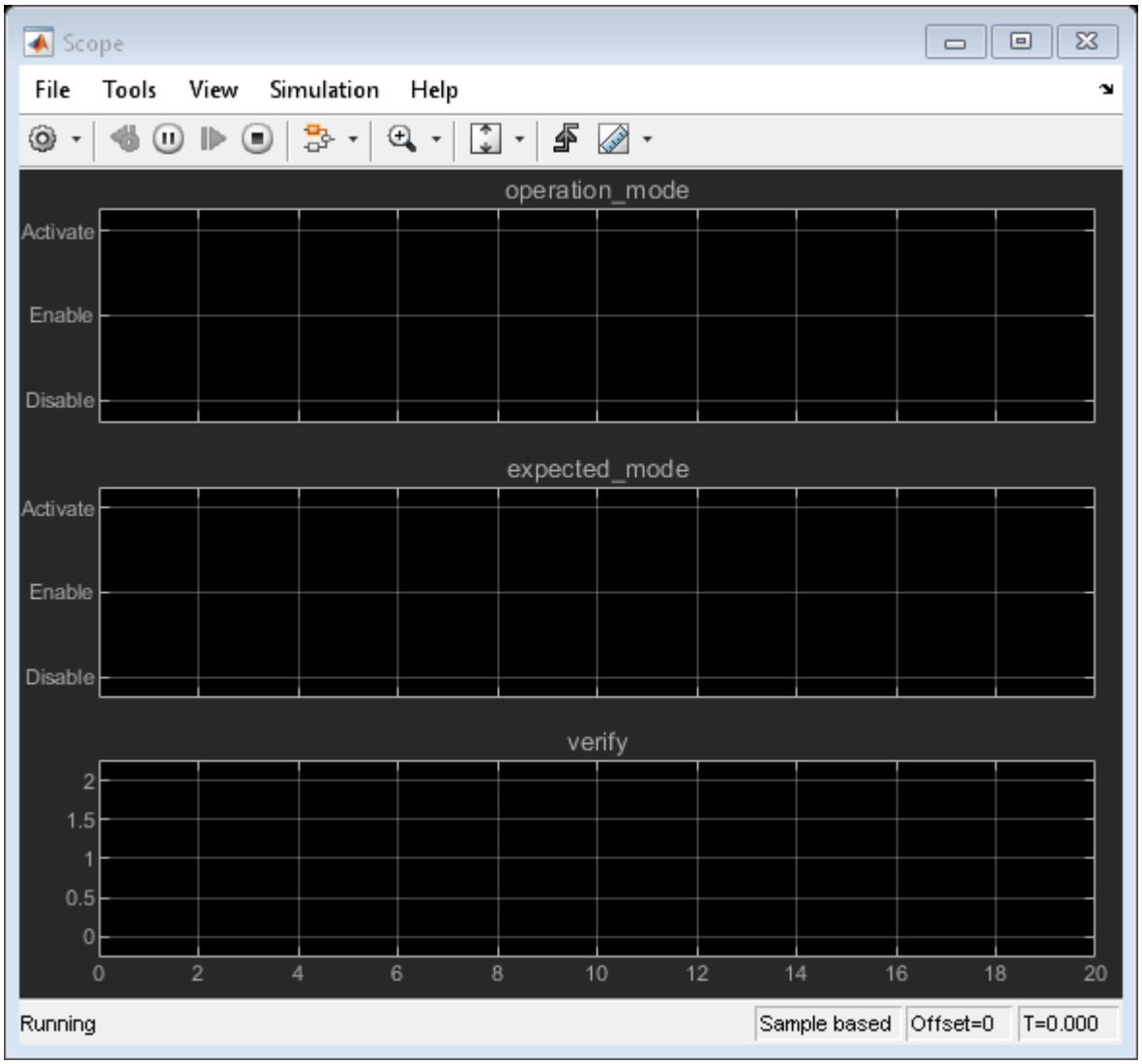
6. Create a sliced model **sldvSliceCruiseControlHarness_sliced** containing only the area of interest.

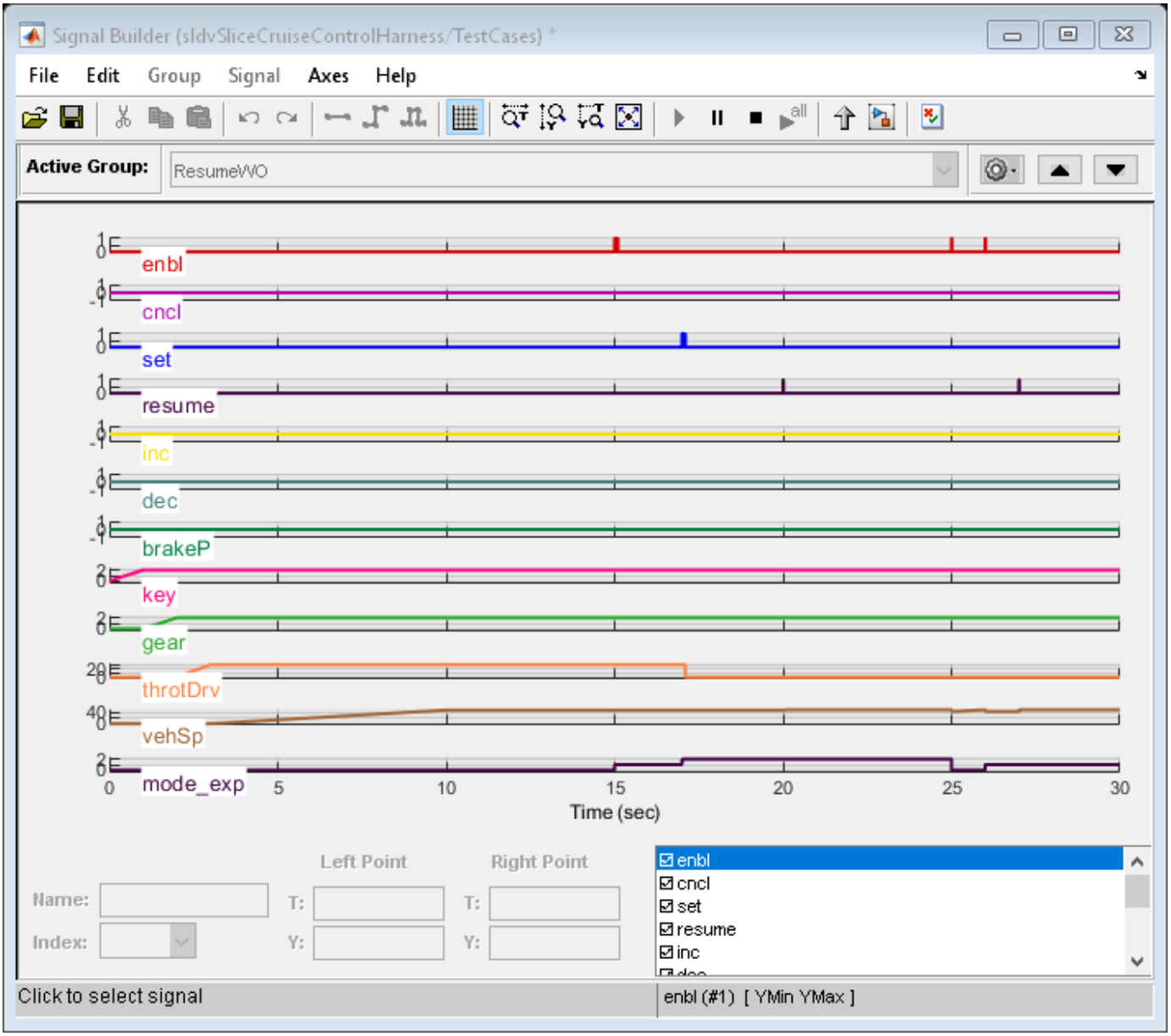
```
slicedModel = slice(obj,'sldvSliceCruiseControlHarness_sliced')
open_system('sldvSliceCruiseControlHarness_sliced')
```

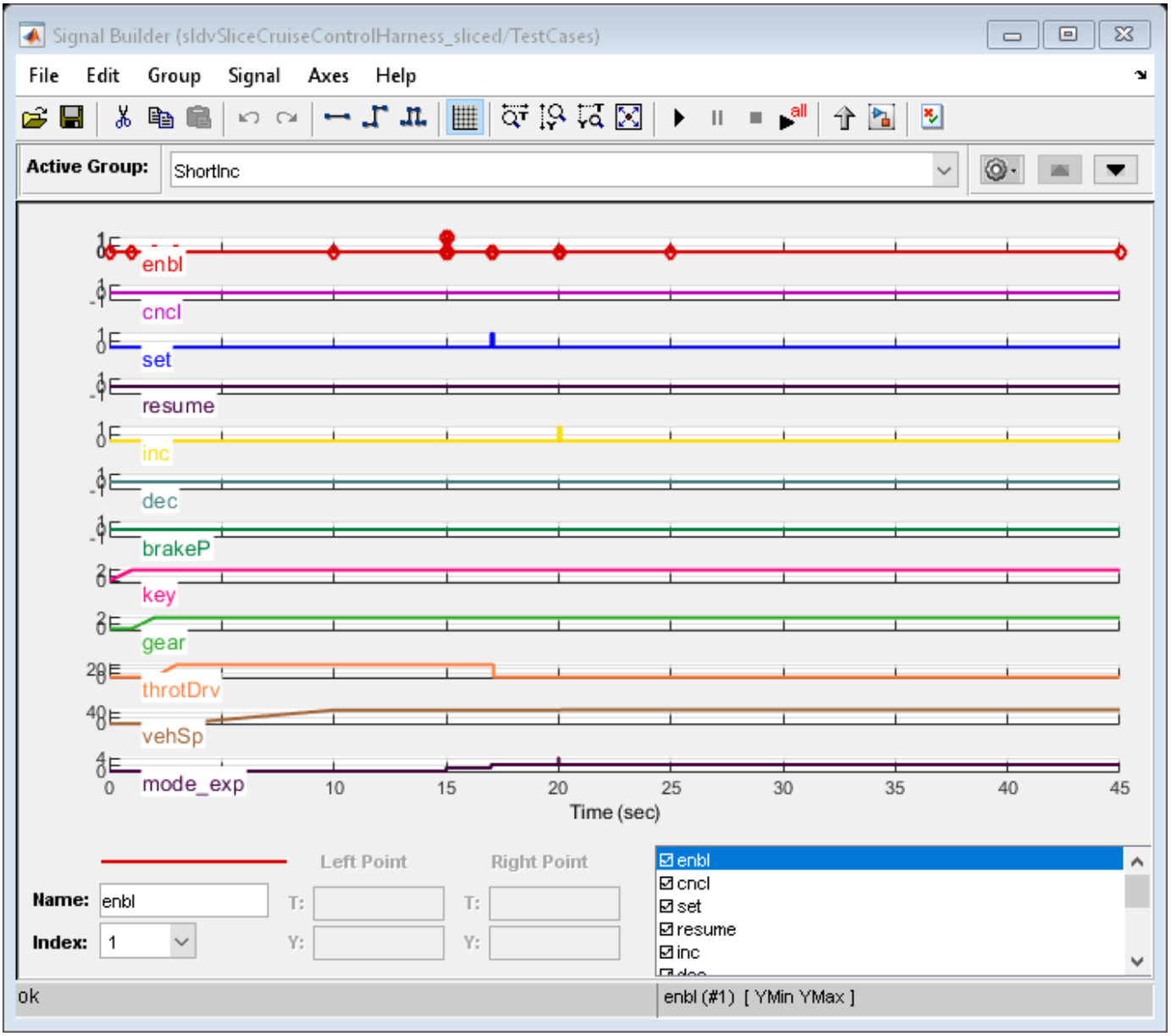
```
slicedModel =
```

```
'sldvSliceCruiseControlHarness_sliced'
```




Scope
File
Tools
View
Simulation
Help
operation_mode
Activate
Enable
Disable
expected_mode
Activate
Enable
Disable
verify
2
1.5
1
0.5
0
0
2
4
6
8
10
12
14
16
18
20
Running
Sample based
Offset=0
T=0.000


Signal Builder (sldvSliceCruiseControlHarness/TestCases) *
File Edit Group Signal Axes Help
Active Group:
ResumeWO
enbl
cncl
set
resume
inc
dec
brakeP
key
gear
throtDrv
vehSp
mode_exp
0 5 10 15 20 25 30
Time (sec)
Left Point
Right Point
Name:
Index:
T:
Y:
enbl
cncl
set
resume
inc
Click to select signal
enbl (#1) [YMin YMax]



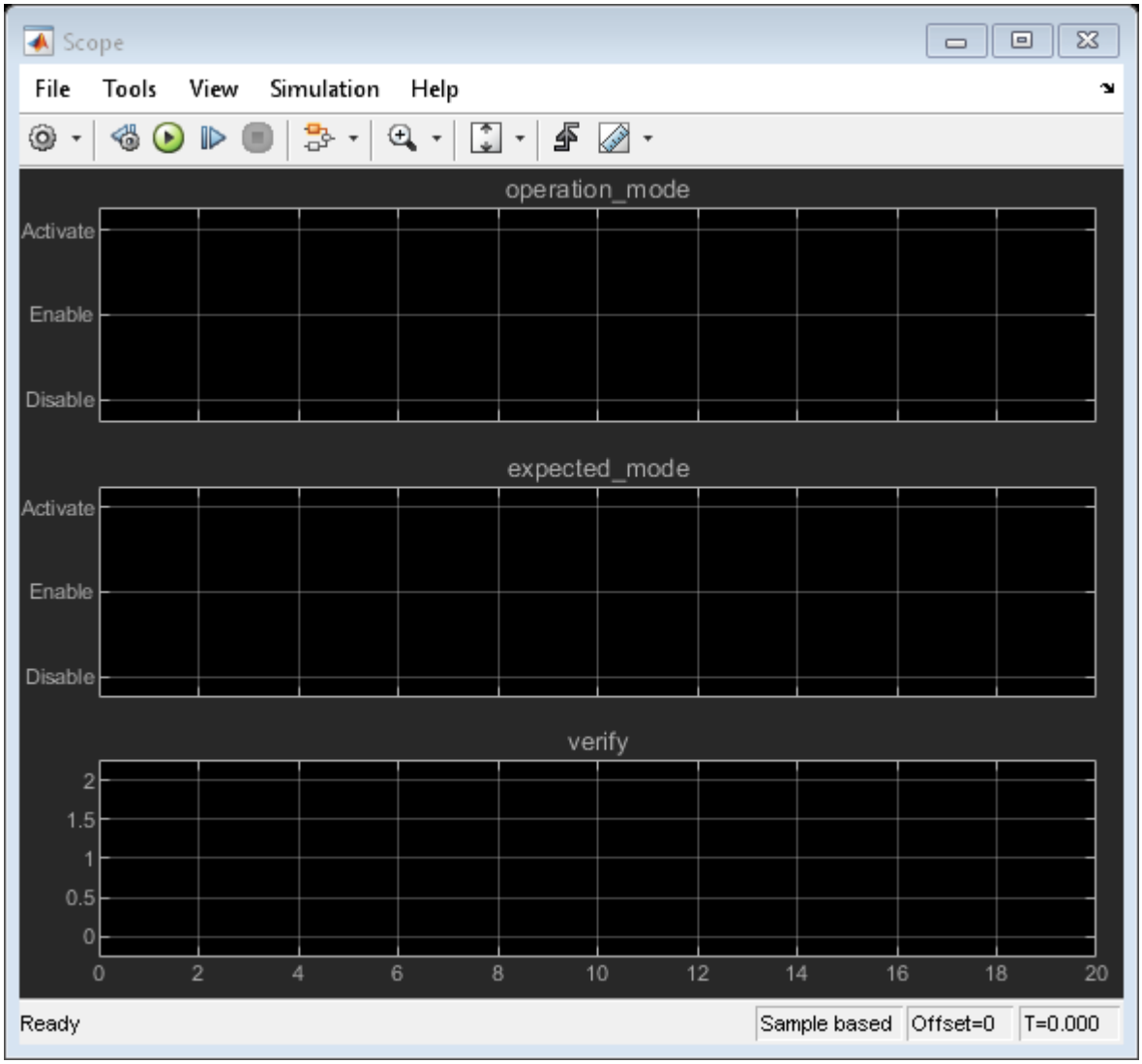
The sliced model **sldvSliceCruiseControlHarness_sliced** now contains a simplified version of the source model **sldvSliceCruiseControlHarness**. The simplified standalone model contains only those parts of the model that are upstream of the specified starting point and active during the time window of interest.

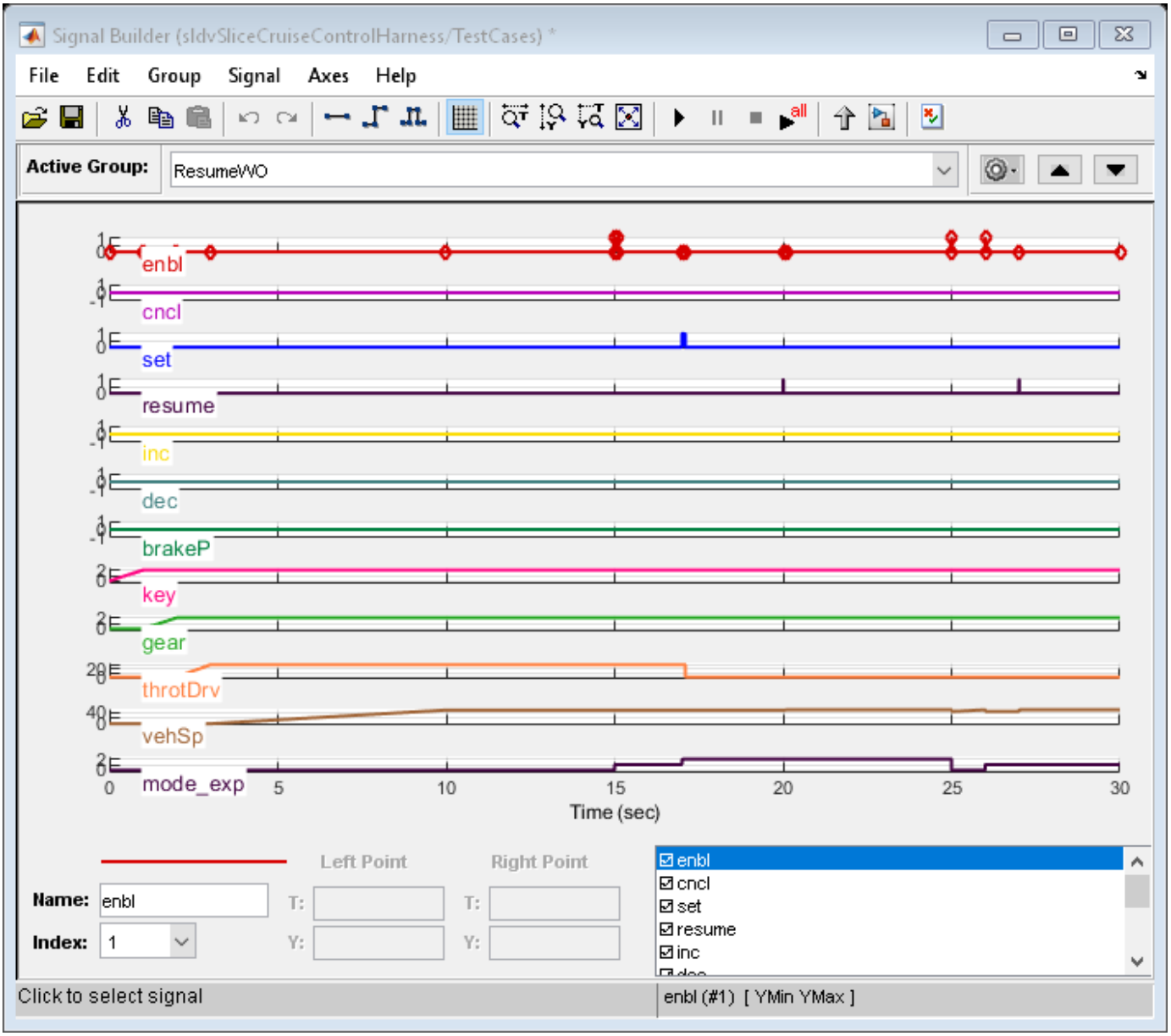
Investigate the Sliced Model and Debug the Source Model

You can now debug the unexpected behavior in the simplified standalone model and then apply changes to the source model.

1. To enable editing the model again, terminate the Model Slicer mode.

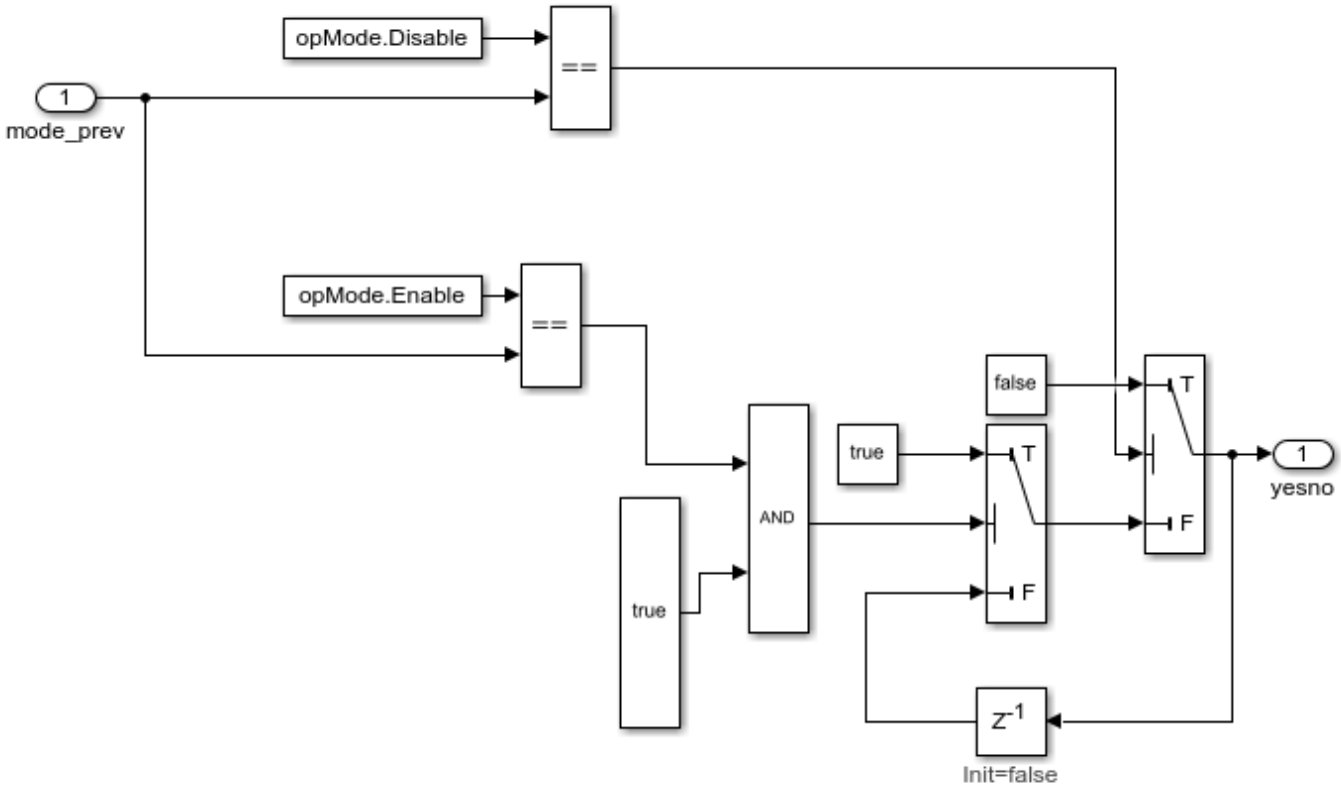
```
terminate(obj)
```


Scope
File
Tools
View
Simulation
Help
operation_mode
Activate
Enable
Disable
expected_mode
Activate
Enable
Disable
verify
2
1.5
1
0.5
0
0
2
4
6
8
10
12
14
16
18
20
Ready
Sample based
Offset=0
T=0.000



2. Navigate to the area of the sliced model that contains the unexpected behavior.

```
open_system('sldvSliceCruiseControlHarness_sliced/Model/CruiseControlMode/opMode/resumeCondition/
```

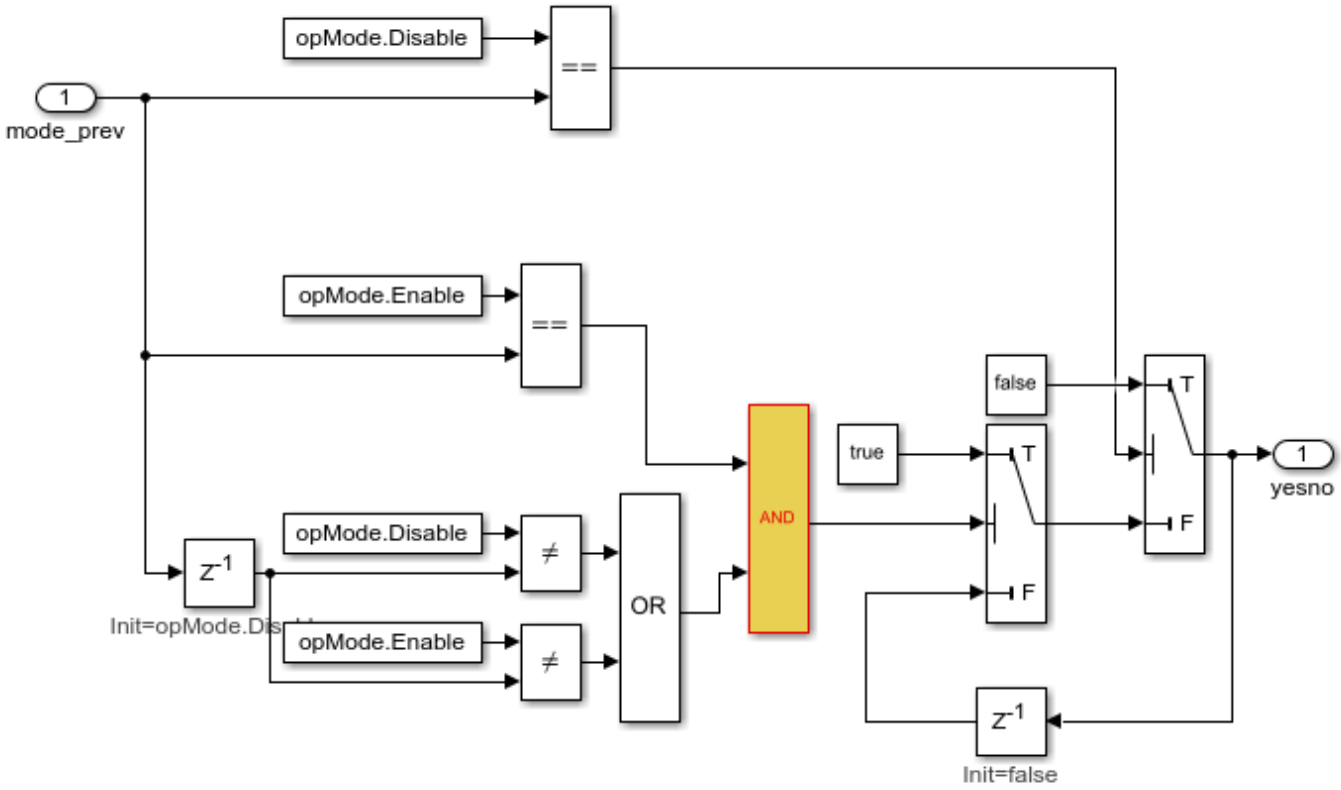


The *AND* Logical Operator block in this subsystem has a truncated *true* constant attached to its second input port. This *true* constant indicates that the second input port is always *true* during the restricted time window for this sliced model, causing the cruise control system not to enter the "has canceled" state.

3. Navigate to the equivalent *AND* Logical Operator block in the source system by using `slslicertrace` to view the blocks connected to the second input port.

```
h = slslicertrace('SOURCE',...
 'sldvSliceCruiseControlHarness_sliced/Model/CruiseControlMode/opMode/resumeCondition/hasCancele
hilite_system(h)
```

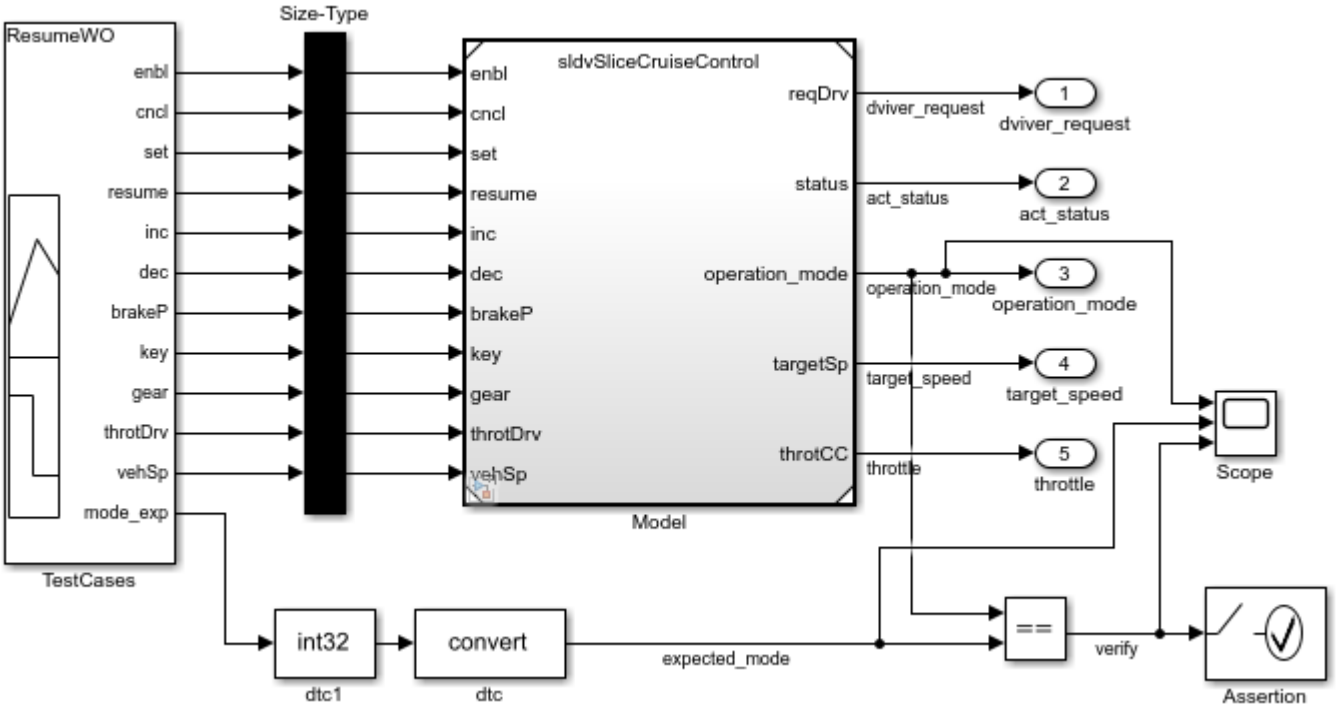
```
h =

  670.0006
```

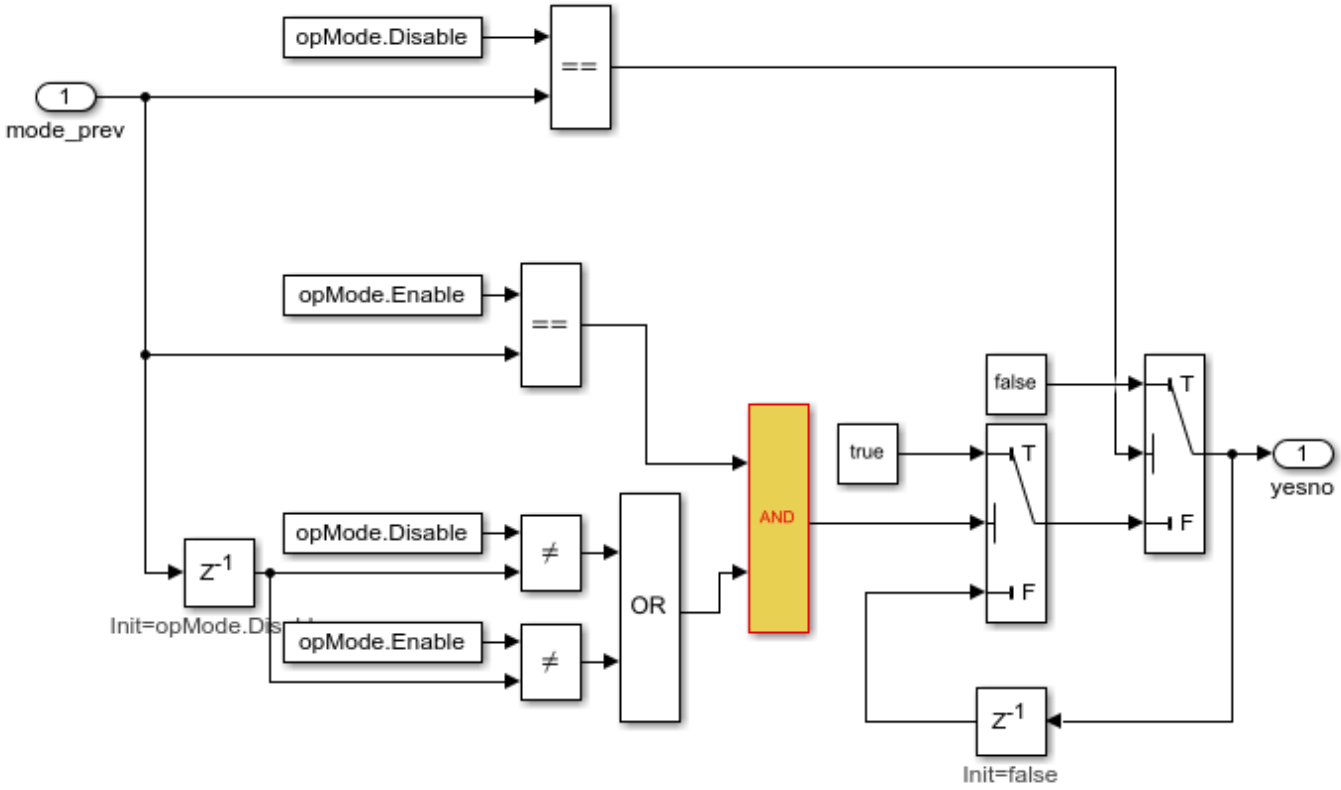


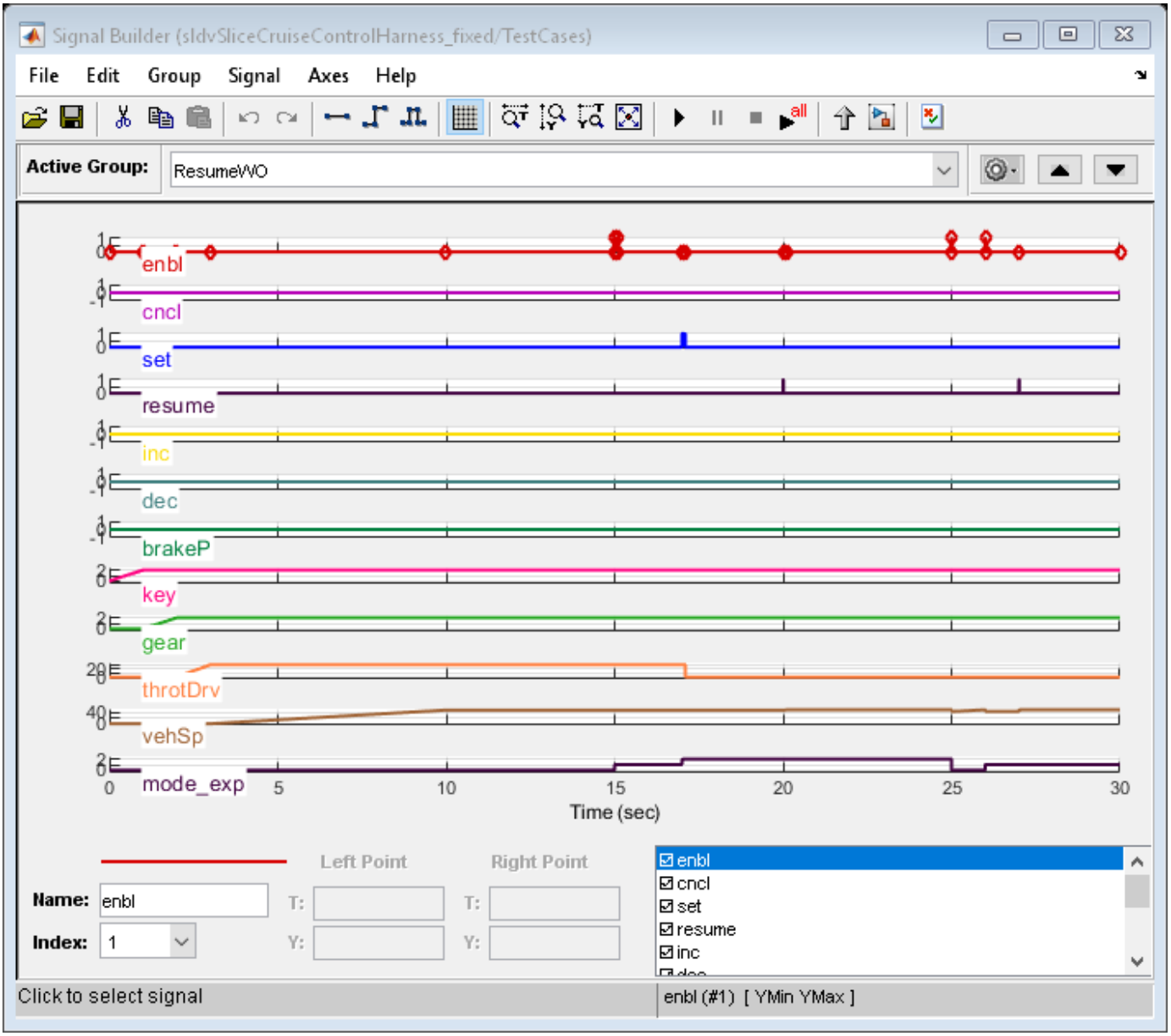
The *OR* Logical Operator block in this subsystem is always *true* in the current configuration. Changing the *OR* Logical Operator block to an *AND* Logical Operator block rectifies this error.

4. Before making edits, create new copies of the cruise control model and the test harness model.

```
save_system('sldvSliceCruiseControl','sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness','sldvSliceCruiseControlHarness_fixed')
```


ResumeWO
enbl
cncl
set
resume
inc
dec
brakeP
key
gear
throtDrv
vehSp
mode_exp
TestCases
Size-Type
sldvSliceCruiseControl
reqDrv
status
operation_mode
targetSp
throtCC
Model
dviver_request
act_status
operation_mode
target_speed
throttle
Scope
int32
dtc1
convert
dtc
expected_mode
==
verify
Assertion


opMode.Disable
==
1
mode_prev
opMode.Enable
==
false
T
true
T
AND
1
yesno
opMode.Disable
≠
OR
F
z-1
Init=opMode.Dis
opMode.Enable
≠
F
z-1
Init=false



5. Update the model reference in the test harness to refer to the newly saved model.

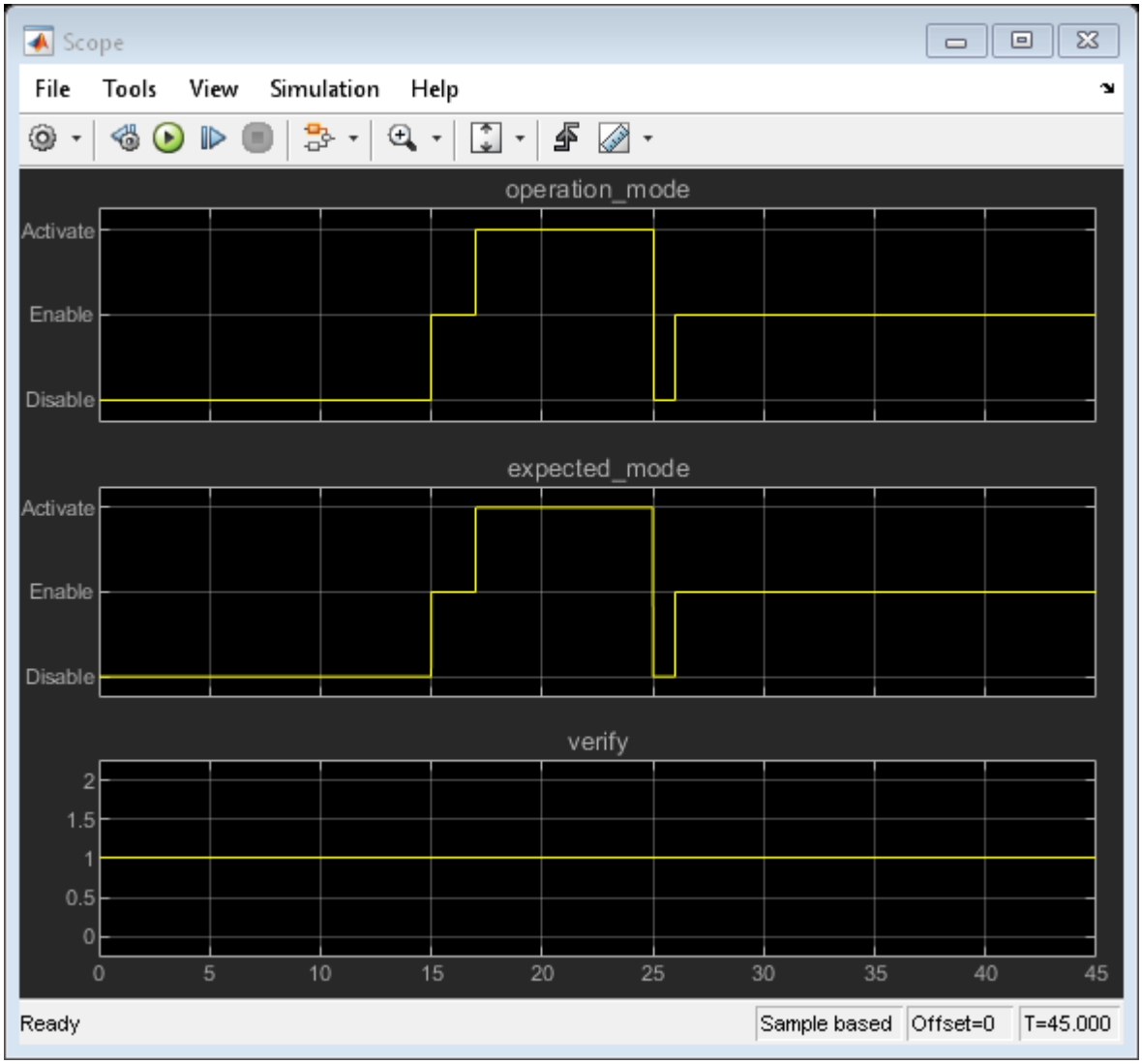
```
set_param('sldvSliceCruiseControlHarness_fixed/Model','ModelNameDialog','sldvSliceCruiseControl_
```

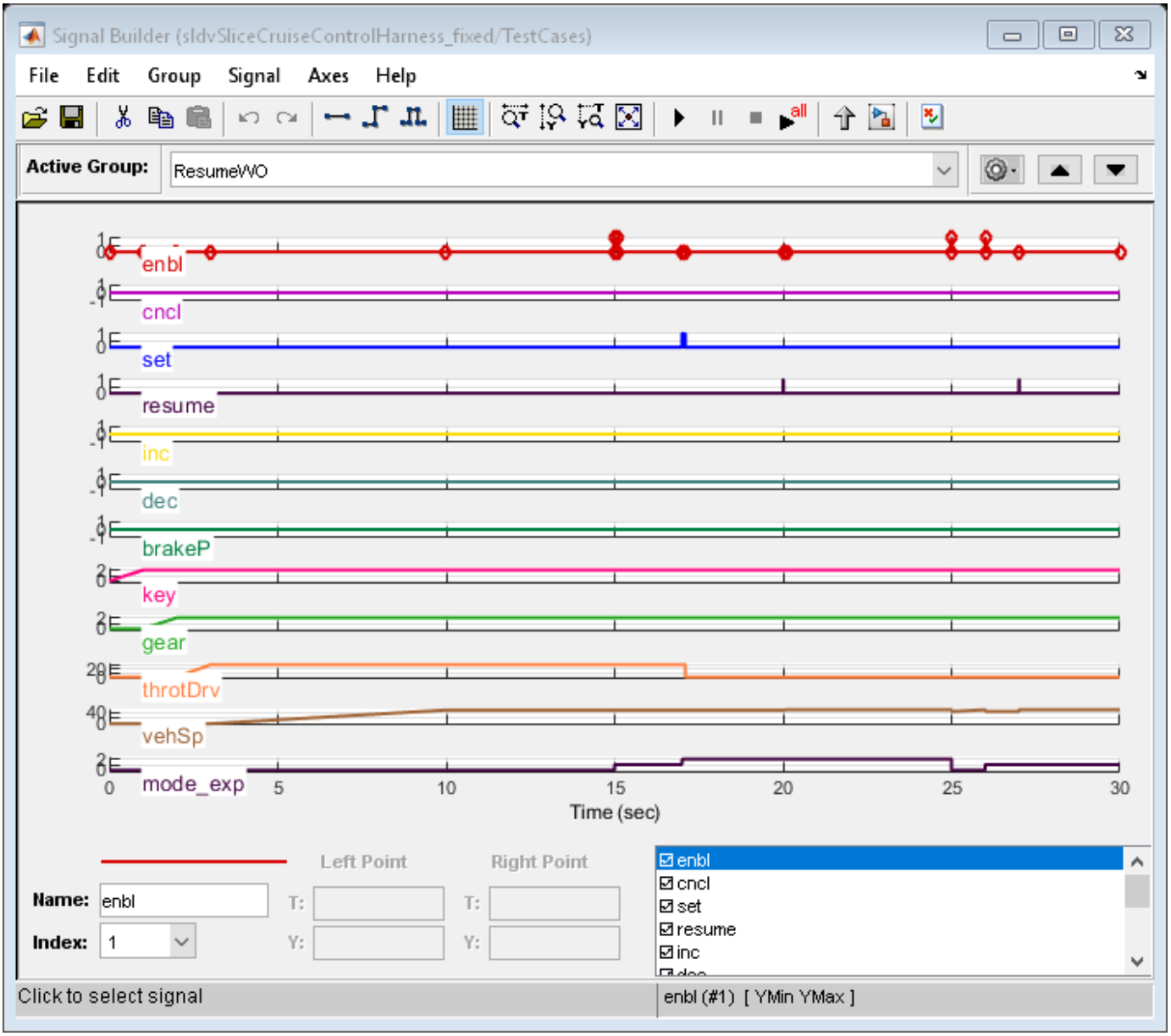
6. Use the block path of the erroneous Logical Operator block to fix the error.

```
set_param('sldvSliceCruiseControl_fixed/CruiseControlMode/opMode/resumeCondition/hasCanceled/Logi
```

7. Simulate the test harness for 45 seconds with the fixed model to confirm the corrected behavior.

```
sim('sldvSliceCruiseControlHarness_fixed')
```


Scope
File
Tools
View
Simulation
Help
operation_mode
Activate
Enable
Disable
expected_mode
Activate
Enable
Disable
verify
2
1.5
1
0.5
0
0
5
10
15
20
25
30
35
40
45
Ready
Sample based
Offset=0
T=45.000



The scope now shows that the expected operation mode is the same as the actual operation mode.

Clean Up

To complete the demo, save and close all models and remove the Model Slicer configuration object.

```
save_system('sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness_fixed')
close_system('sldvSliceCruiseControl_fixed')
close_system('sldvSliceCruiseControlHarness_fixed')
close_system('sldvSliceCruiseControlHarness_sliced')
clear obj
```

Programmatically Resolve Unexpected Behavior in a Model with Model Slicer

In this example, you evaluate a Simulink model, detect unexpected behavior, and use Model Slicer to programmatically isolate and resolve the unexpected behavior. When you plan to reuse your API commands and extend their use to other models, a programmatic approach is useful.

Prerequisites

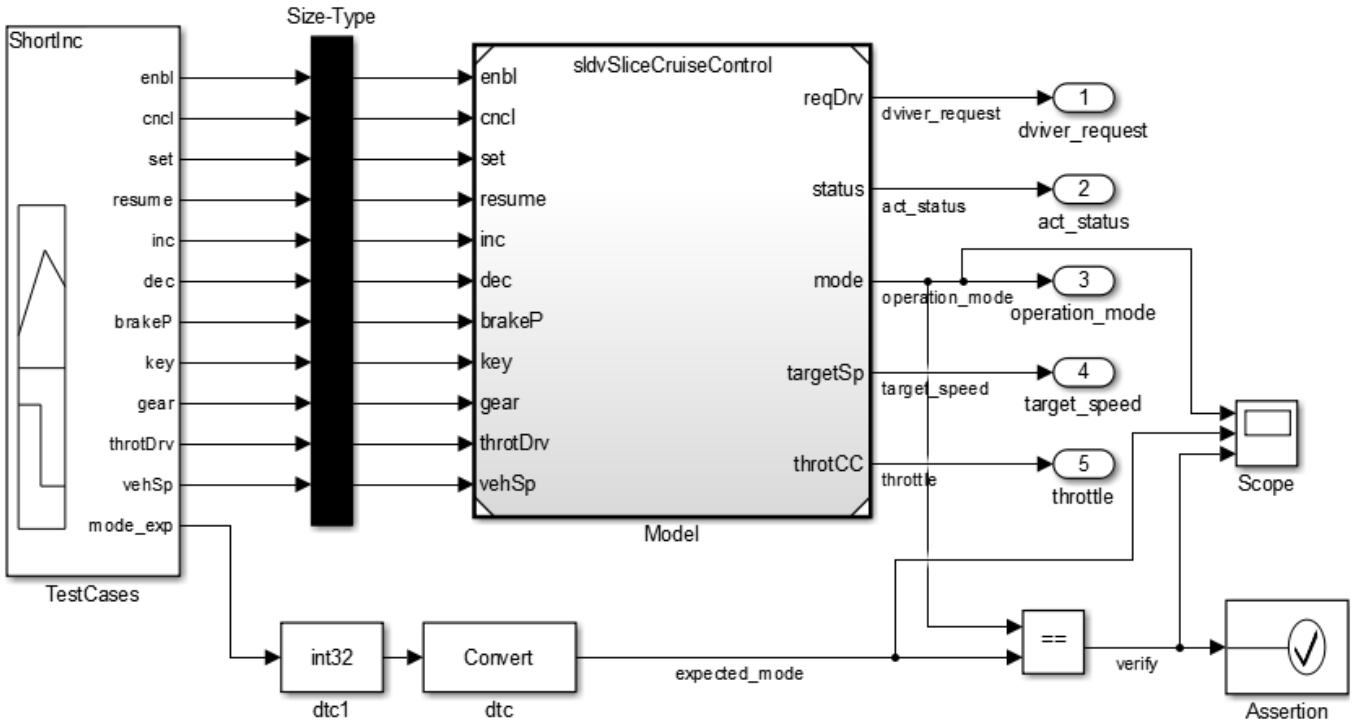
Be familiar with the behavior and purpose of Model Slicer and the functionality of the Model Slicer API. “Highlight Functional Dependencies” on page 8-2 outlines how to use Model Slicer user interface to explore models. The `slslicer`, `slsliceroptions`, and `slslicertrace` function reference pages contain the Model Slicer API command help.

Find the Area of the Model Responsible for Unexpected Behavior

The `sldvSliceCruiseControlHarness` test harness model contains a cruise controller subsystem `sldvSliceCruiseControl` and a block, `TestCases`, containing a test case for this subsystem. You first simulate the model to execute the test case. You then evaluate the behavior of the model to find and isolate areas of the model responsible for unexpected behavior:

1 Open the `sldvSliceCruiseControlHarness` test harness for the cruise control model.

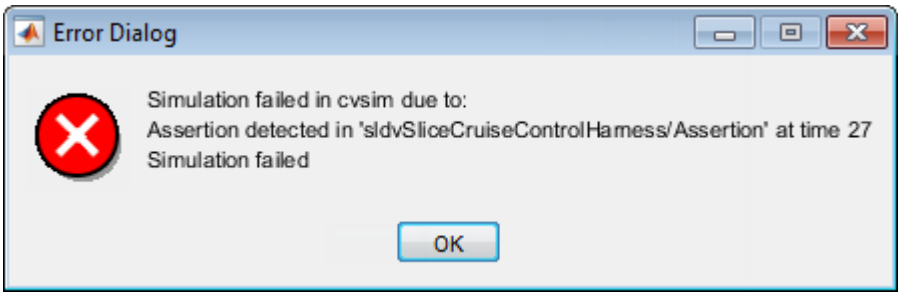
```
open_system('sldvSliceCruiseControlHarness')
```



Note The Assertion block is set to **Stop simulation when assertion fails** when the actual operation mode is not the same as the expected operation mode.

The TestCases block contains several test inputs for sldvSliceCruiseControl.

2 In the TestCases Signal Builder click the **Run all** button to run all of the included test cases. You receive an error during the ResumeWO test case.



The Assertion block halted simulation at 27 seconds, because the actual operation mode was not the same as the expected operation mode. Click **OK** to close this error message.

3 In the sldvSliceCruiseControlHarness model, double-click the Assertion block, clear **Enable assertion**, and click **OK**.

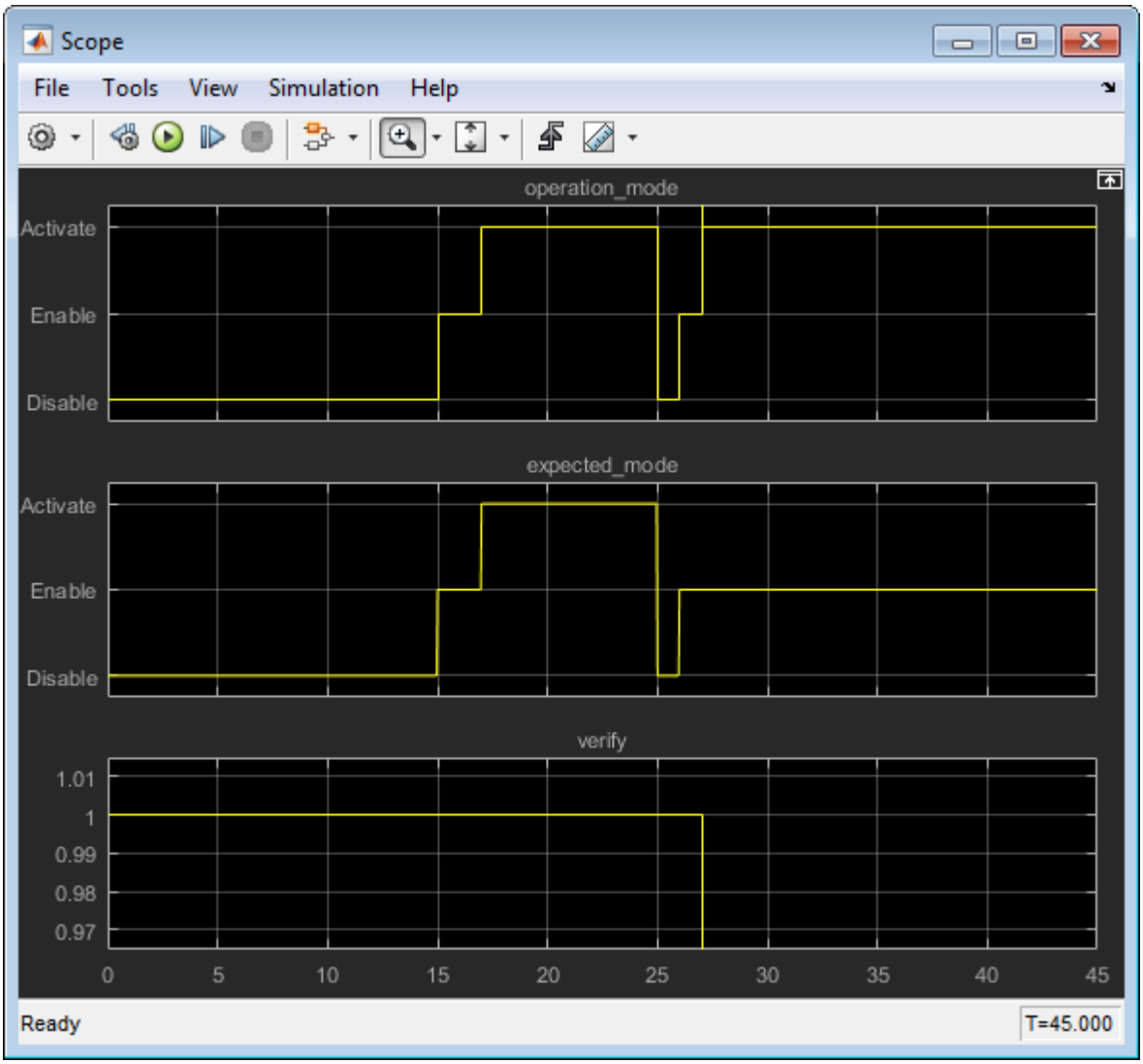
```
set_param('sldvSliceCruiseControlHarness/Assertion','Enabled','off')
```

4 Set the **Active Group** of the `TestCases` Signal Builder block to the test case containing the error and run the simulation again.

```
signalbuilder('sldvSliceCruiseControlHarness/TestCases', 'ACTIVEGROUP', 12)
sim('sldvSliceCruiseControlHarness')
```

The Scope block in the model contains three signals:

- `operation_mode` – displays the actual operation mode of the subsystem.
- `expected_mode` – displays the expected operation mode of the subsystem that the test case provides.
- `verify` – displays a Boolean value comparing the operation mode and the expected mode.



The scope shows a disparity between the expected operation mode and the actual operation mode beginning at time 27. Now that you know the outport displaying the unexpected behavior and the time window containing the unexpected behavior, use Model Slicer to isolate and analyze the unexpected behavior.

Isolate the Area of the Model Responsible for Unexpected Behavior

1 Create a Model Slicer configuration object for the model using `slslicer`. The Command Window displays the slice properties for this Model Slicer configuration.

```
obj = slslicer('sldvSliceCruiseControlHarness')
```

```
obj =

  SLSlicer with properties:

        Configuration: [1x1 SLSlicerAPI.SLSlicerConfig]
         ActiveConfig: 1
```

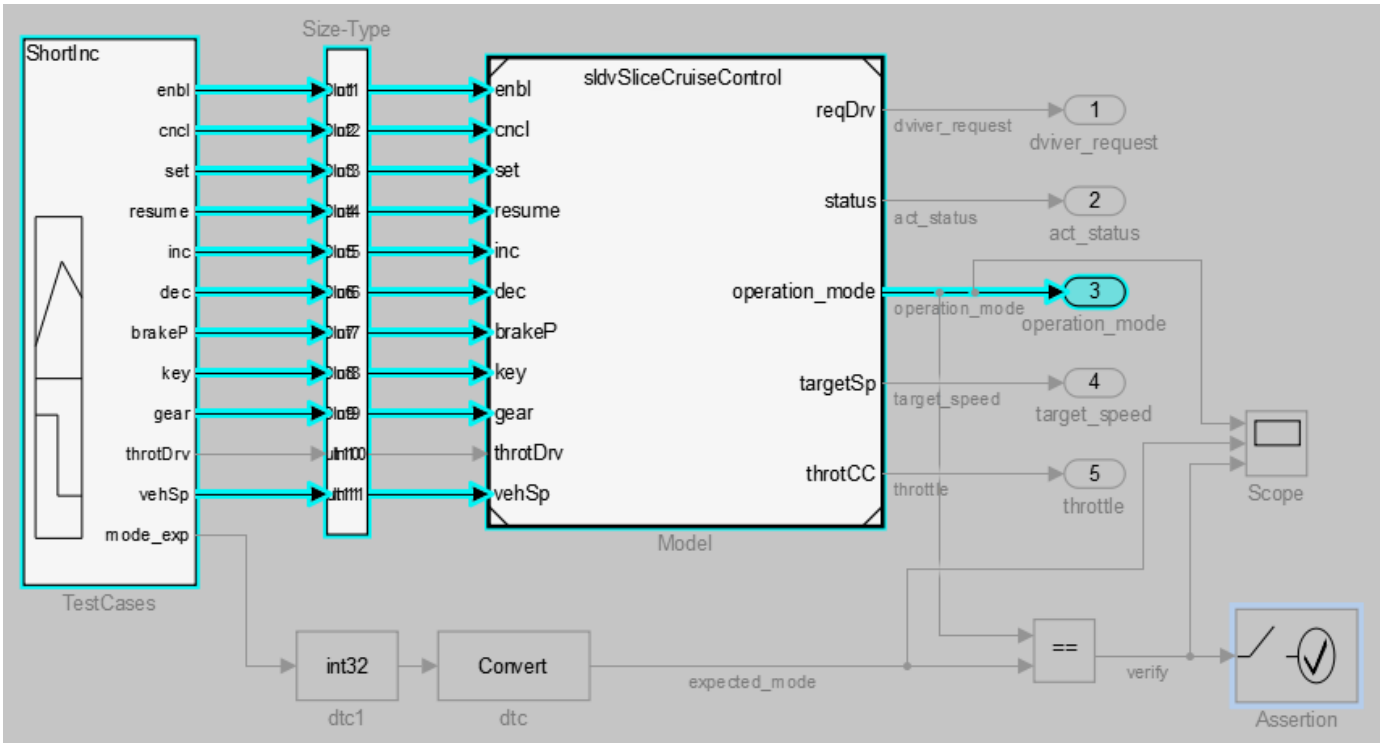
```
      DisplayedConfig: []
       StorageOptions: [1x1 struct]
      AnalysisOptions: [1x1 struct]
         SliceOptions: [1x1 struct]
        InlineOptions: [1x1 struct]

  Contents of active configuration:
                 Name: 'untitled'
          Description: ''
                Color: [0 1 1]
   SignalPropagation: 'upstream'
       StartingPoint: [1x0 struct]
      ExclusionPoint: [1x0 struct]
          Constraint: [1x0 struct]
      SliceComponent: [1x0 struct]
       UseTimeWindow: 0
        CoverageFile: ''
        UseDeadLogic: 0
       DeadLogicFile: ''
```

2 Activate the slice highlighting mode of Model Slicer to compile the model and prepare it for dependency analysis.

```
activate(obj)
```

3 Add the `operation_mode` outport block as a starting point and highlight it.

```
addStartingPoint(obj,'sldvSliceCruiseControlHarness/operation_mode')
highlight(obj)
```

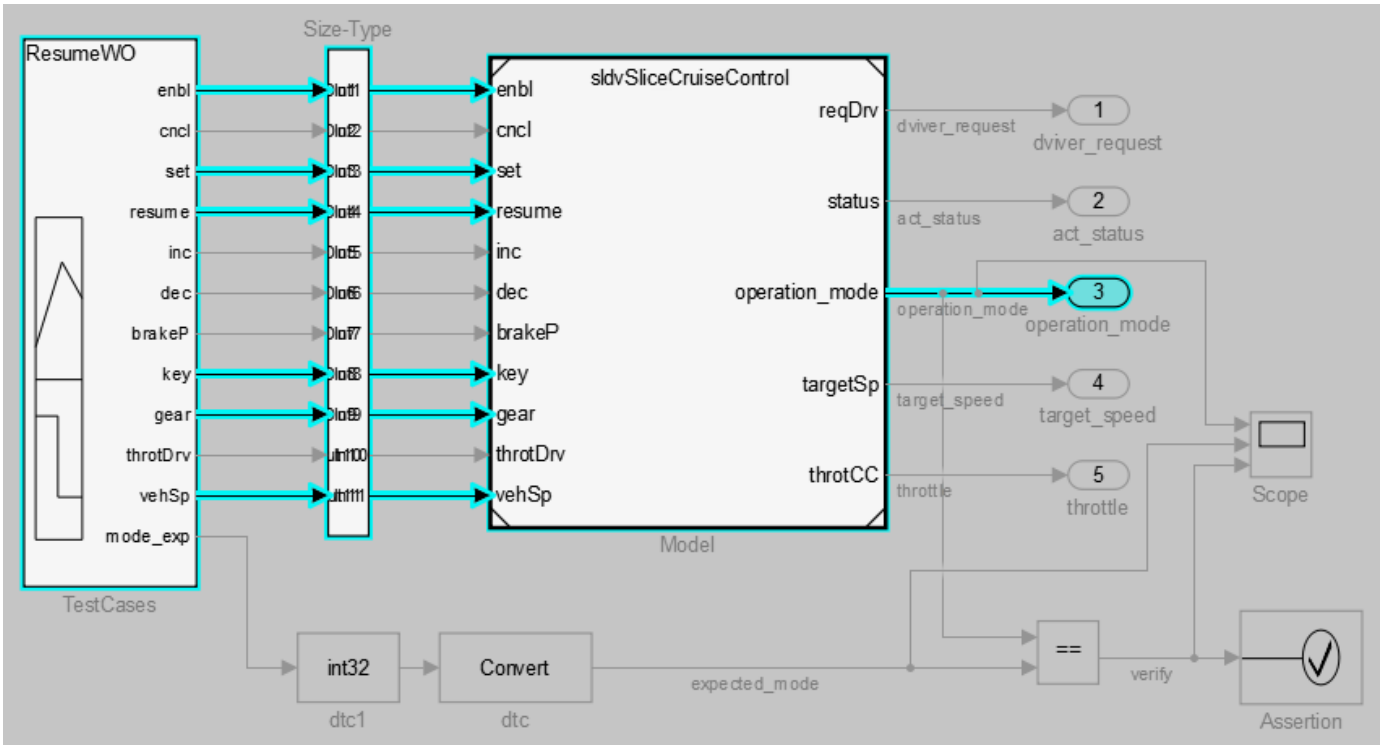


The area of the model upstream of the starting point and active during simulation is highlighted.

4 Simulate the model within a restricted simulation time window (maximum 30 seconds) to highlight only the area of the model upstream of the starting point and active during the time window of interest.

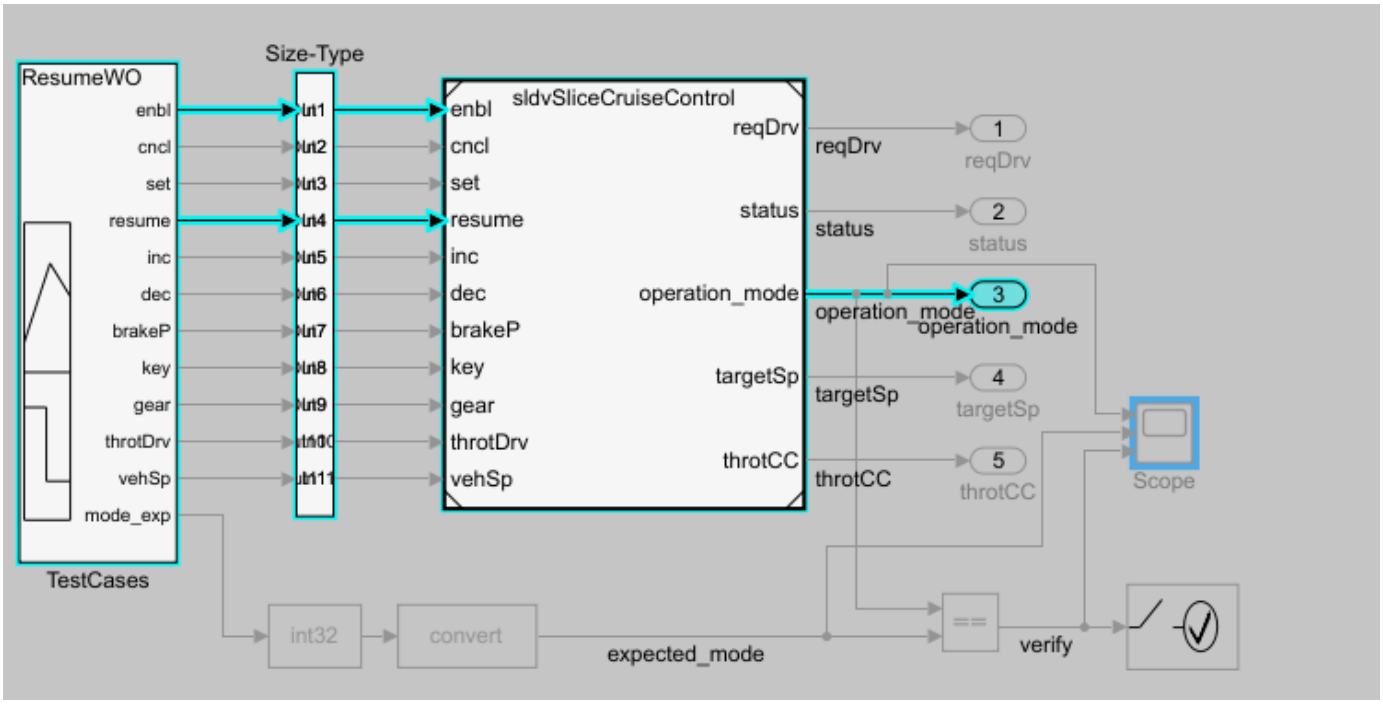
```
simulate(obj,0,30)
```

Only the portion of the model upstream of the starting point and active during the simulation time window is highlighted.



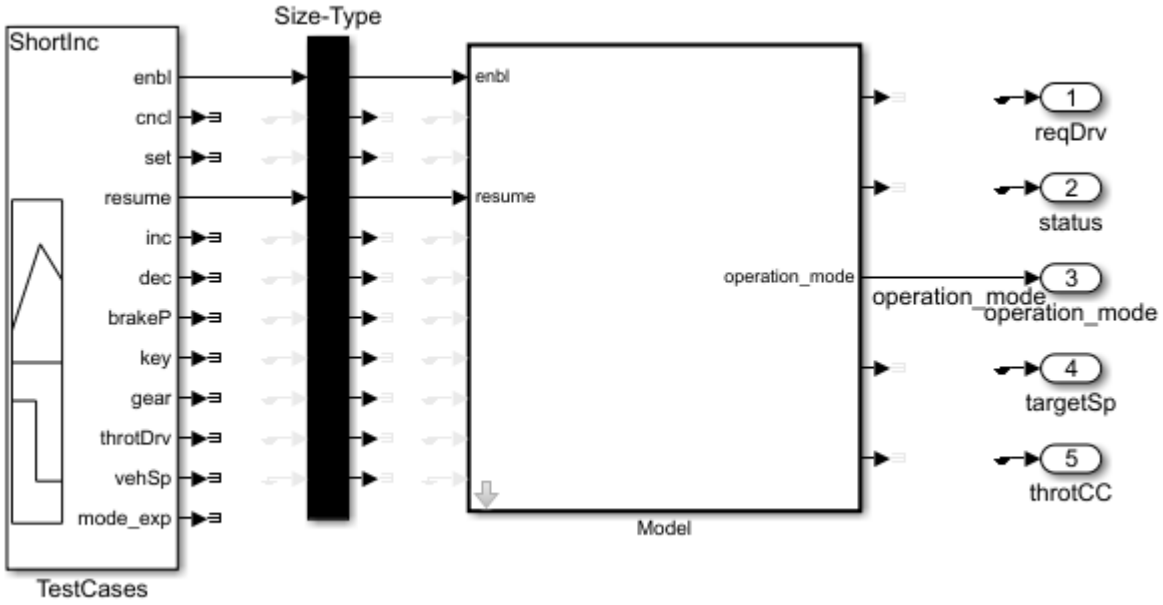
5 You can further narrow the simulation time window by changing the start time to 20 seconds.

```
setTimeWindow(obj,20,30)
```



6 Create a sliced model `sldvSliceCruiseControlHarness_sliced` containing only the area of interest.

```
slicedModel = slice(obj,'sldvSliceCruiseControlHarness_sliced')
open_system('sldvSliceCruiseControlHarness_sliced')
```



The sliced model `sldvSliceCruiseControlHarness_sliced` now contains a simplified version of the source model `sldvSliceCruiseControlHarness`. The simplified standalone model contains

only those parts of the model that are upstream of the specified starting point and active during the time window of interest.

Investigate the Sliced Model and Debug the Source Model

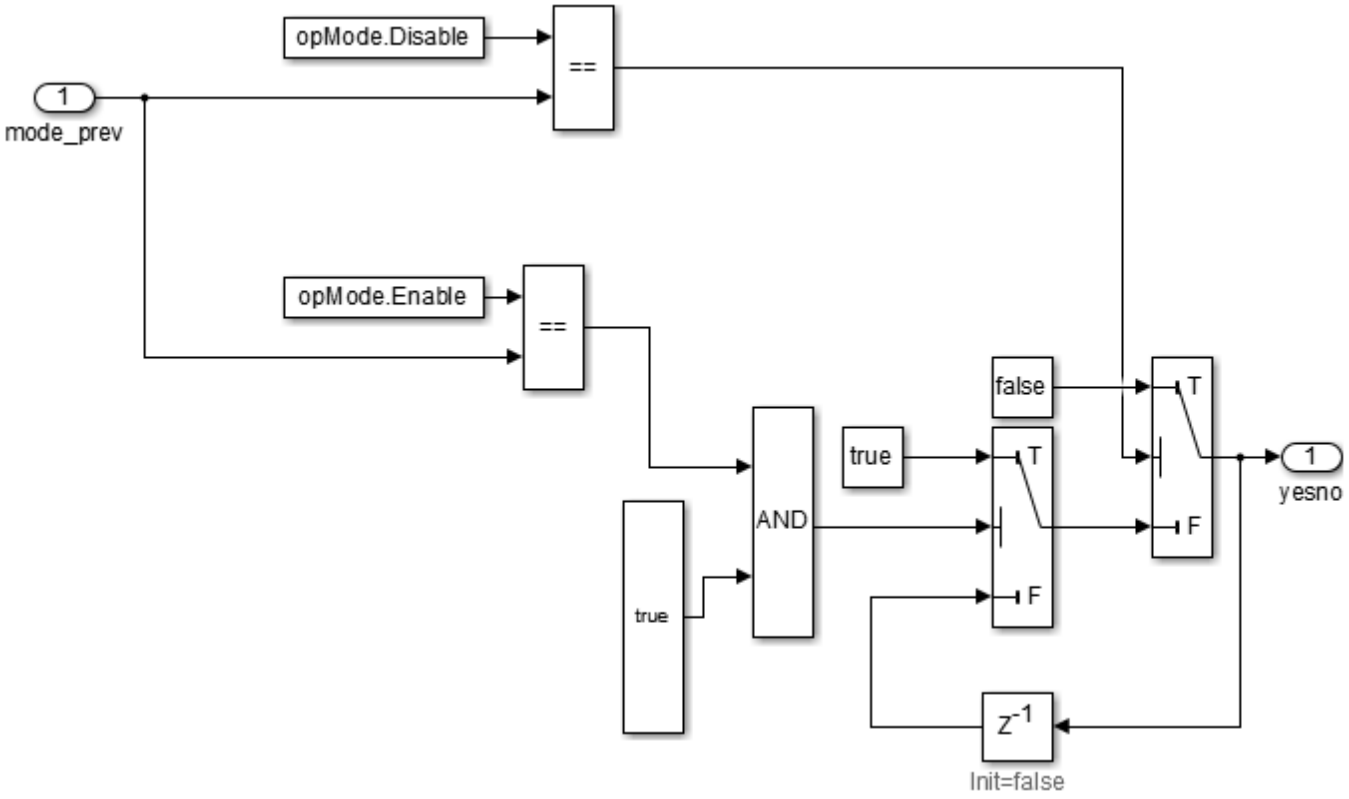
You can now debug the unexpected behavior in the simplified standalone model and then apply changes to the source model.

1. To enable editing the model again, terminate the Model Slicer mode.

   ```
   terminate(obj)
   ```

2. Navigate to the area of the sliced model that contains the unexpected behavior.

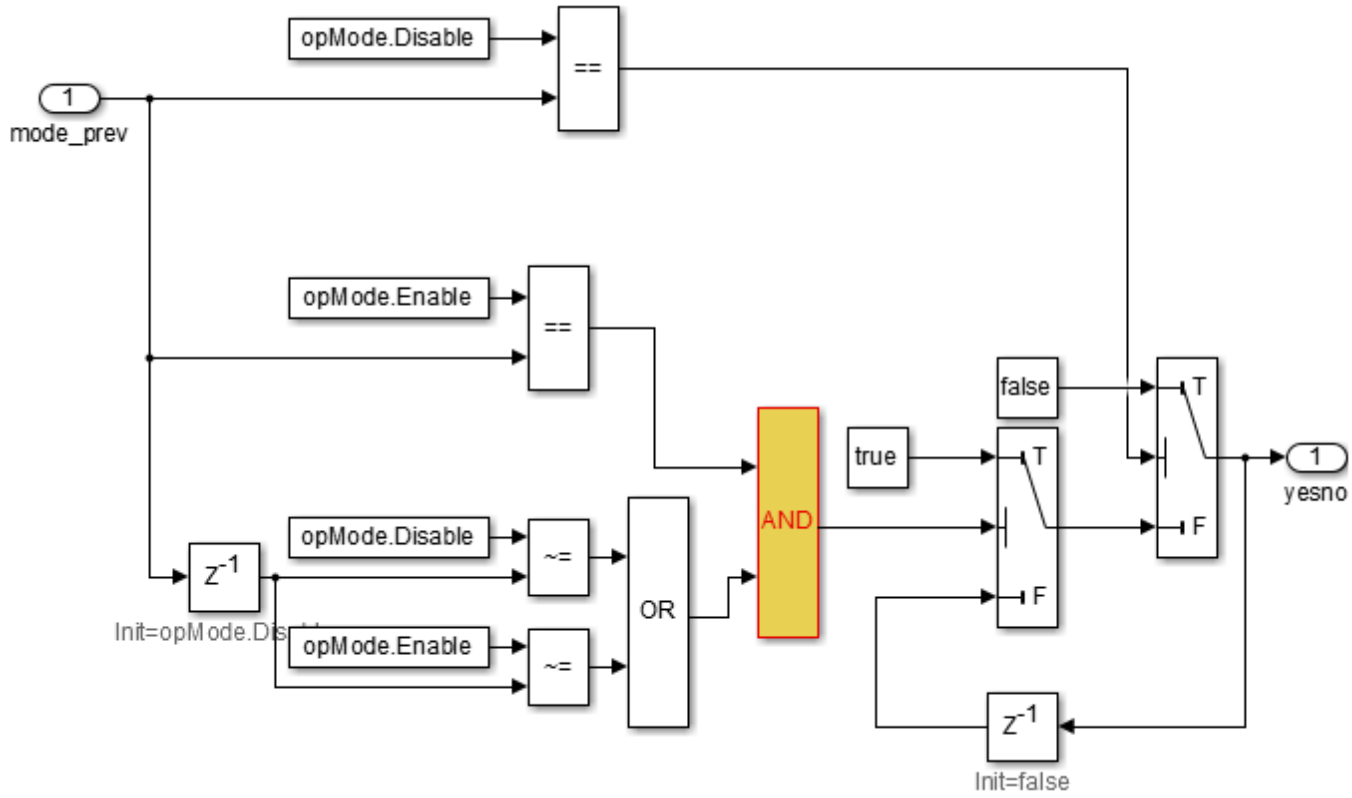
   ```
   open_system('sldvSliceCruiseControlHarness_sliced/Model/CruiseControlMode/opMode/resumeCondition/hasCanceled')
   ```



   The AND Logical Operator block in this subsystem has a truncated `true` constant attached to its second input port. This `true` constant indicates that the second input port is always `true` during the restricted time window for this sliced model, causing the cruise control system not to enter the "has canceled" state.

3. Navigate to the equivalent AND Logical Operator block in the source system by using `slslicertrace` to view the blocks connected to the second input port.

   ```
   h = slslicertrace('SOURCE',...
    'sldvSliceCruiseControlHarness_sliced/Model/CruiseControlMode/opMode/resumeCondition/hasCanceled/LogicOp1')
   hilite_system(h)
   ```



The OR Logical Operator block in this subsystem is always `true` in the current configuration. Changing the OR Logical Operator block to an AND Logical Operator block rectifies this error.

4 Before making edits, create new copies of the cruise control model and the test harness model.

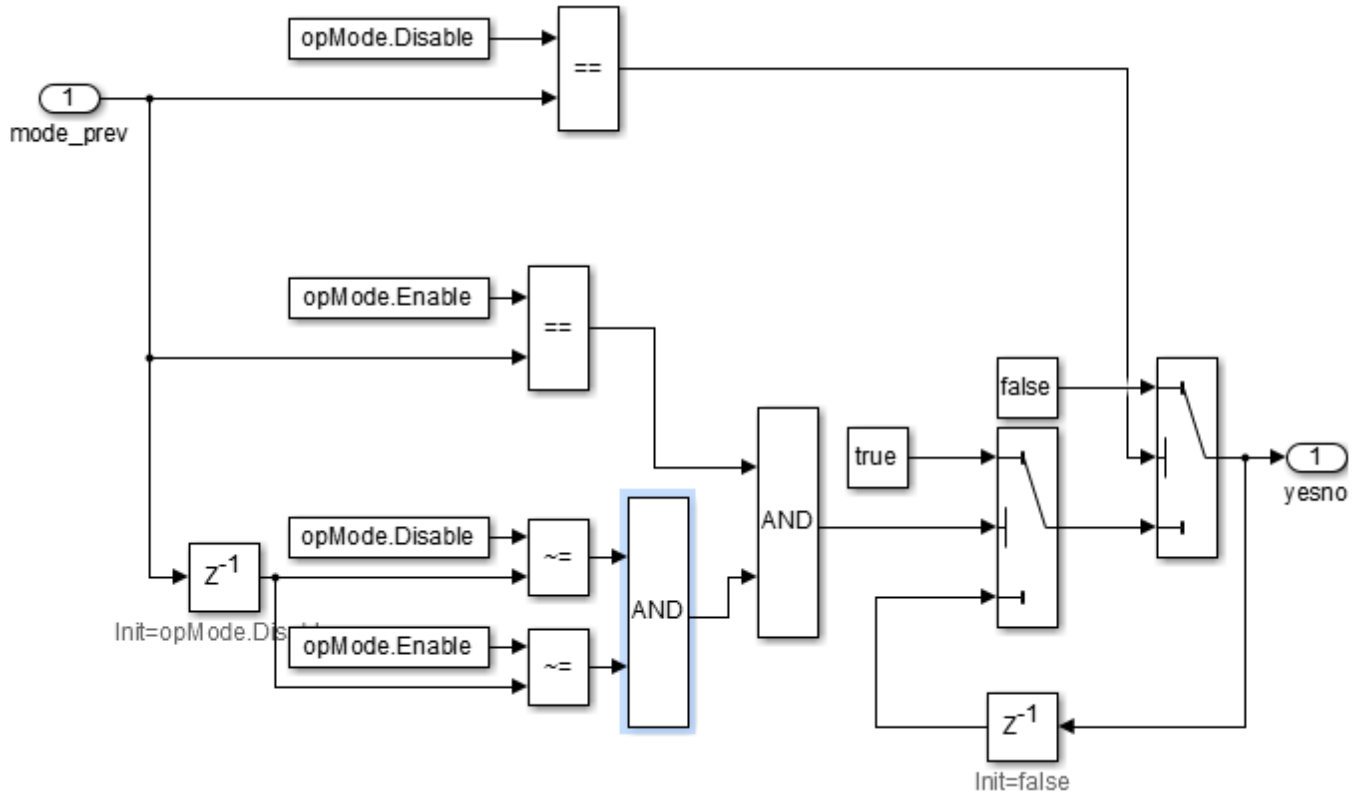
```
save_system('sldvSliceCruiseControl','sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness','sldvSliceCruiseControlHarness_fixed')
```

5 Update the model reference in the test harness to refer to the newly saved model.

```
set_param('sldvSliceCruiseControlHarness_fixed/Model',...
'ModelNameDialog','sldvSliceCruiseControl_fixed.slx')
```

6 Use the block path of the erroneous Logical Operator block to fix the error.

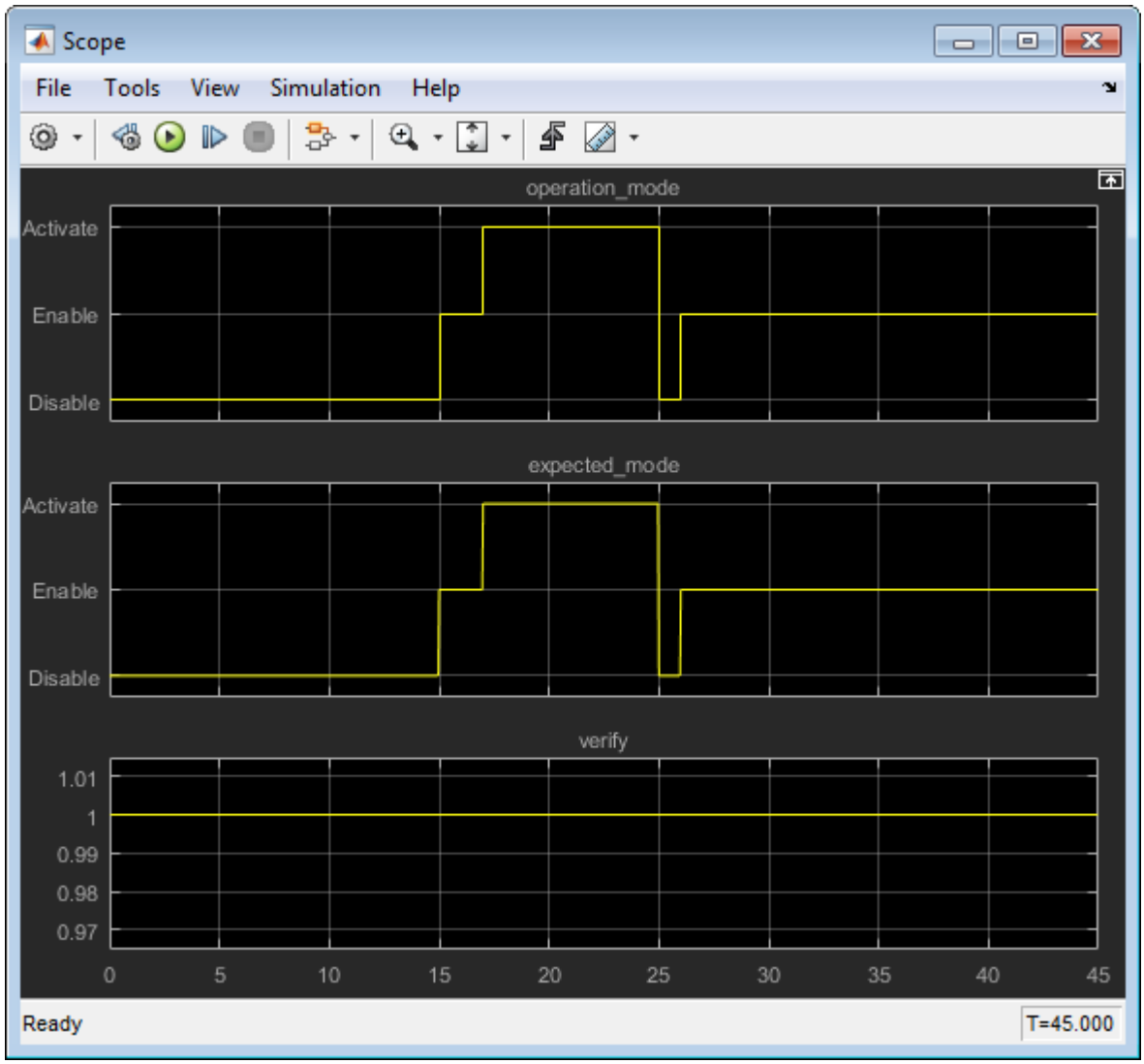
```
set_param('sldvSliceCruiseControl_fixed/CruiseControlMode/opMode/resumeCondition/hasCanceled/LogicOp2',..
'LogicOp','AND')
```



7 Simulate the test harness for 45 seconds with the fixed model to confirm the corrected behavior.

```
sim('sldvSliceCruiseControlHarness_fixed')
```

```
ans =

  Simulink.SimulationOutput:

                  tout: [4501x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
          ErrorMessage: [0x0 char]
```



The scope now shows that the expected operation mode is the same as the actual operation mode.

Clean Up

To complete the demo, save and close all models and remove the Model Slicer configuration object.

```
save_system('sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness_fixed')
close_system('sldvSliceCruiseControl_fixed')
close_system('sldvSliceCruiseControlHarness_fixed')
close_system('sldvSliceCruiseControlHarness_sliced')
clear obj
```

See Also

`slslicer` | `slsliceroptions` | `slslicertrace`

More About

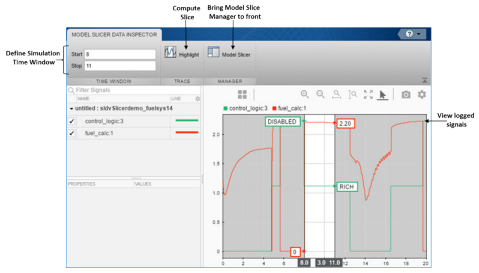
- “Highlight Functional Dependencies” on page 8-2

Refine Highlighted Model Slice by Using Model Slicer Data Inspector

Using the Model Slicer Data Inspector, you can inspect logged signals and refine the highlighted model slice. To refine the highlighted model slice, select the time window in the graphical plot by using data cursors.

In the Model Slicer Data Inspector, you can:

- View signals — Inspect logged signal data after model simulation. See “Inspect Simulation Data” (Simulink).
- Select simulation time window — Define simulation time window by using data cursors in the graphical plot or by defining the **Start** and **Stop** time in the Inspector.
- Highlight — Compute a slice for the defined simulation time window. See “Highlight Functional Dependencies” on page 8-2.



Investigate Highlighted Model Slice by Using Model Slicer Data Inspector

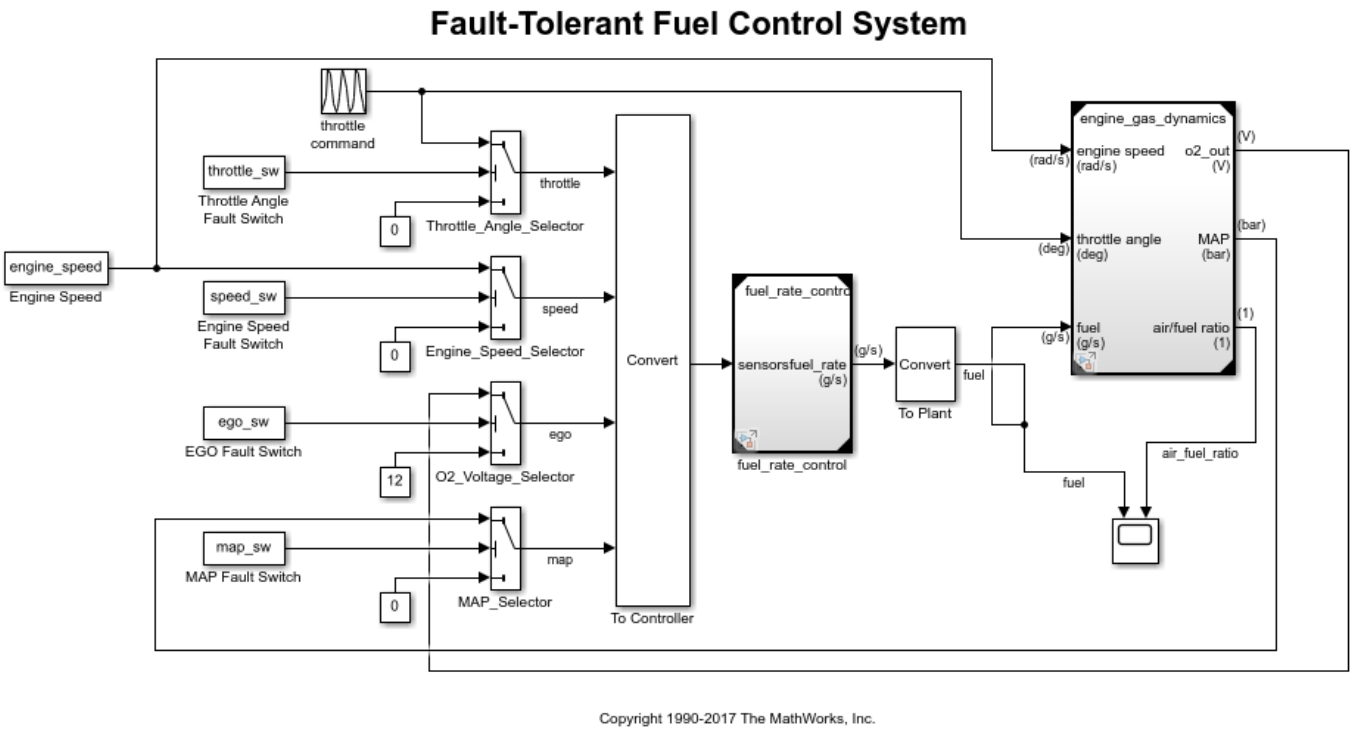
This example shows how to investigate and refine the highlighted model slice by using the Model Slicer Data Inspector.

In the fault-tolerant fuel control system, the `control_logic` controls the fueling mode of the engine. In this example, you slice the `fuel_rate_control` referenced model. Then, investigate the effect of `fuel_rate_ratio` on the `Fueling_mode` of the engine. For more information, see “Modeling a Fault-Tolerant Fuel Control System” (Simulink).

Step 1: Start the Model Slice Manager

To start the Model Slice Manager, open the `fuel_rate_control` model, and select **Apps** > **Model Verification, Validation, and Test** > **Model Slicer**.

```
open_system('sldvSlicerdemo_fuelsys');
```



To select the starting point, open the `fuel_rate_control` model, and add the `fuel-rate` port and the `fuel_mode` output signal as the starting point. To add a port or a signal as a starting point, right-click the port or signal, and select **Model Slicer** > **Add as Starting Point**.

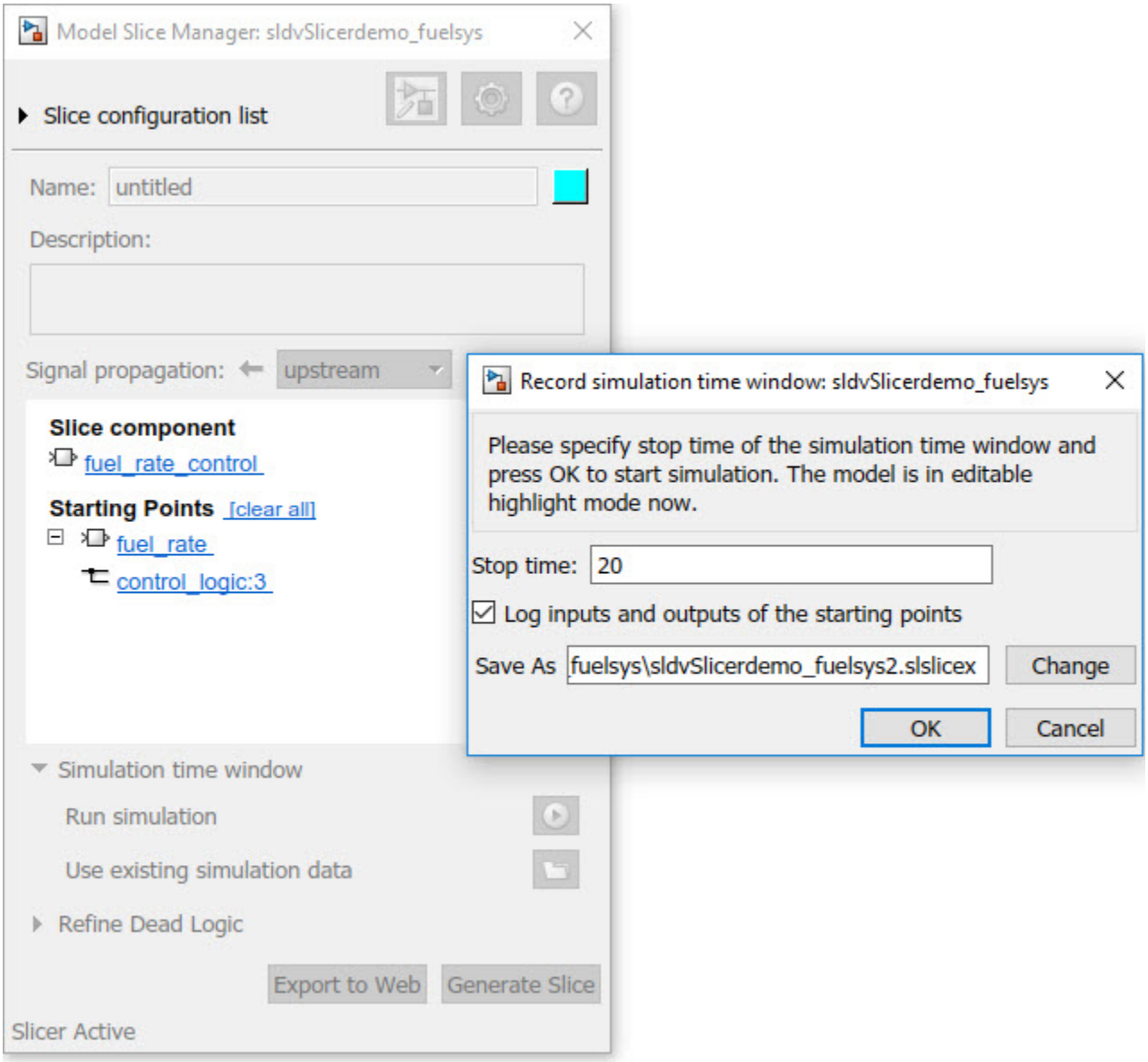
Step 2: Log input and output signals

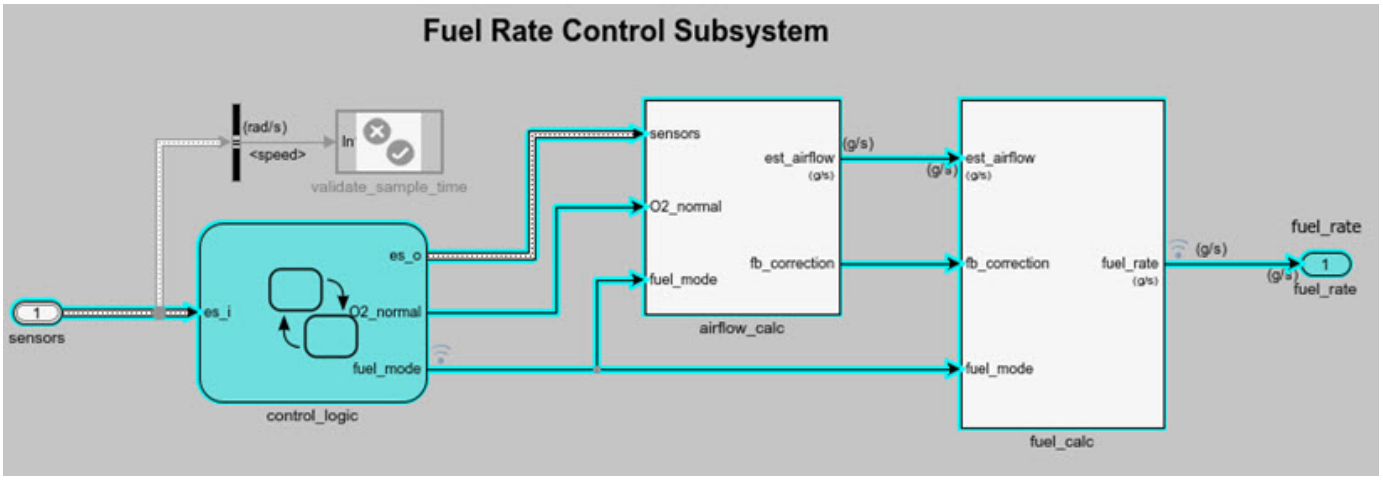
a. In the Model Slice Manager dialog box, select the **Simulation time window** and **Run simulation**.

b. In the Record simulation time window, for the **Stop time**, type `20`.

c. Select the **Log inputs and outputs of the starting points**.

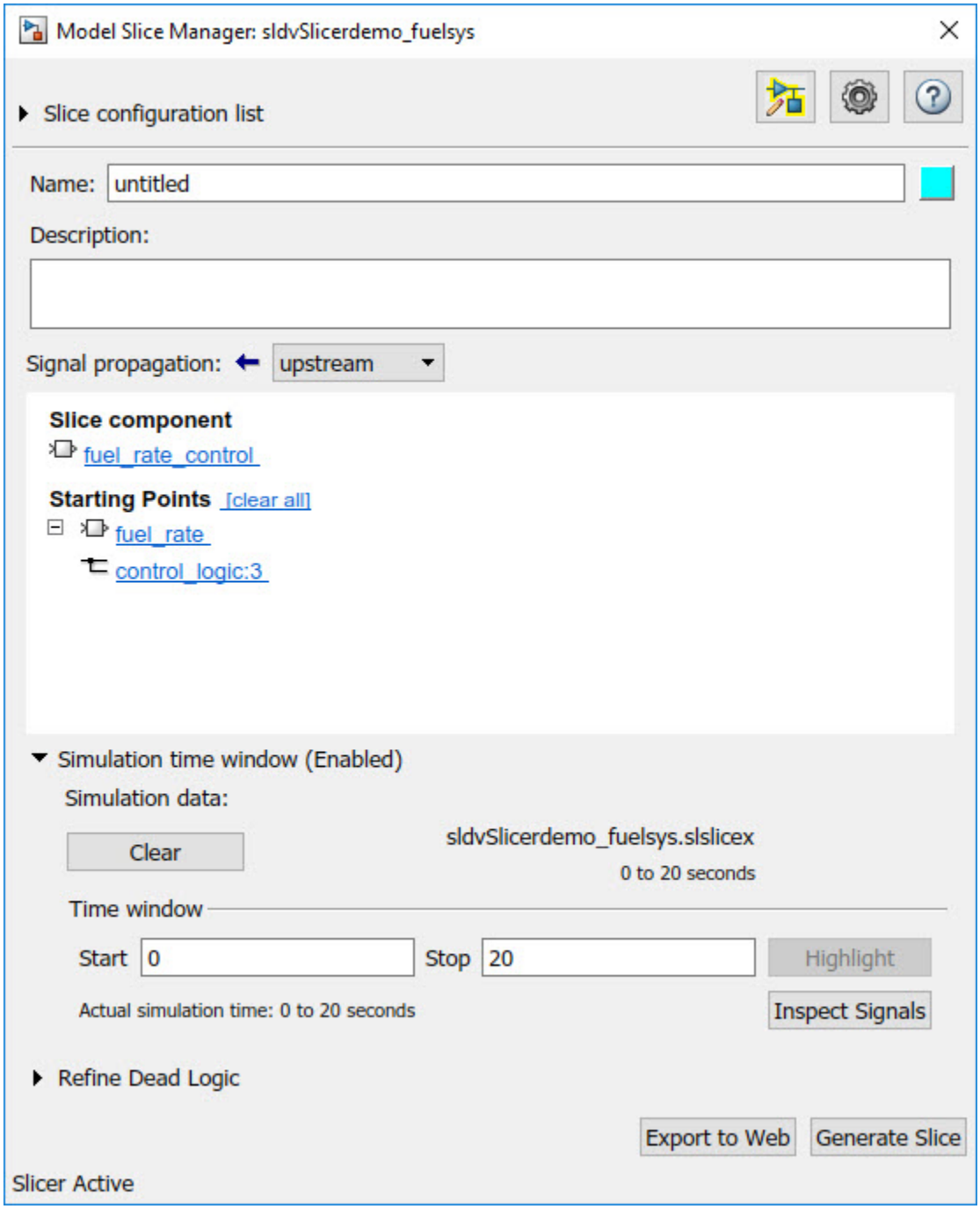
d. Click **OK**.


Model Slice Manager: sldvSlicerdemo_fuelsys
Slice configuration list
Name: untitled
Description:
Signal propagation: upstream
Slice component
fuel_rate_control
Starting Points [clear all]
fuel_rate
control_logic:3
Simulation time window
Run simulation
Use existing simulation data
Refine Dead Logic
Export to Web
Generate Slice
Slicer Active
Record simulation time window: sldvSlicerdemo_fuelsys
Please specify stop time of the simulation time window and press OK to start simulation. The model is in editable highlight mode now.
Stop time: 20
Log inputs and outputs of the starting points
Save As fuelsys\sldvSlicerdemo_fuelsys2.slslicex
Change
OK
Cancel



Step 3: Inspect signals

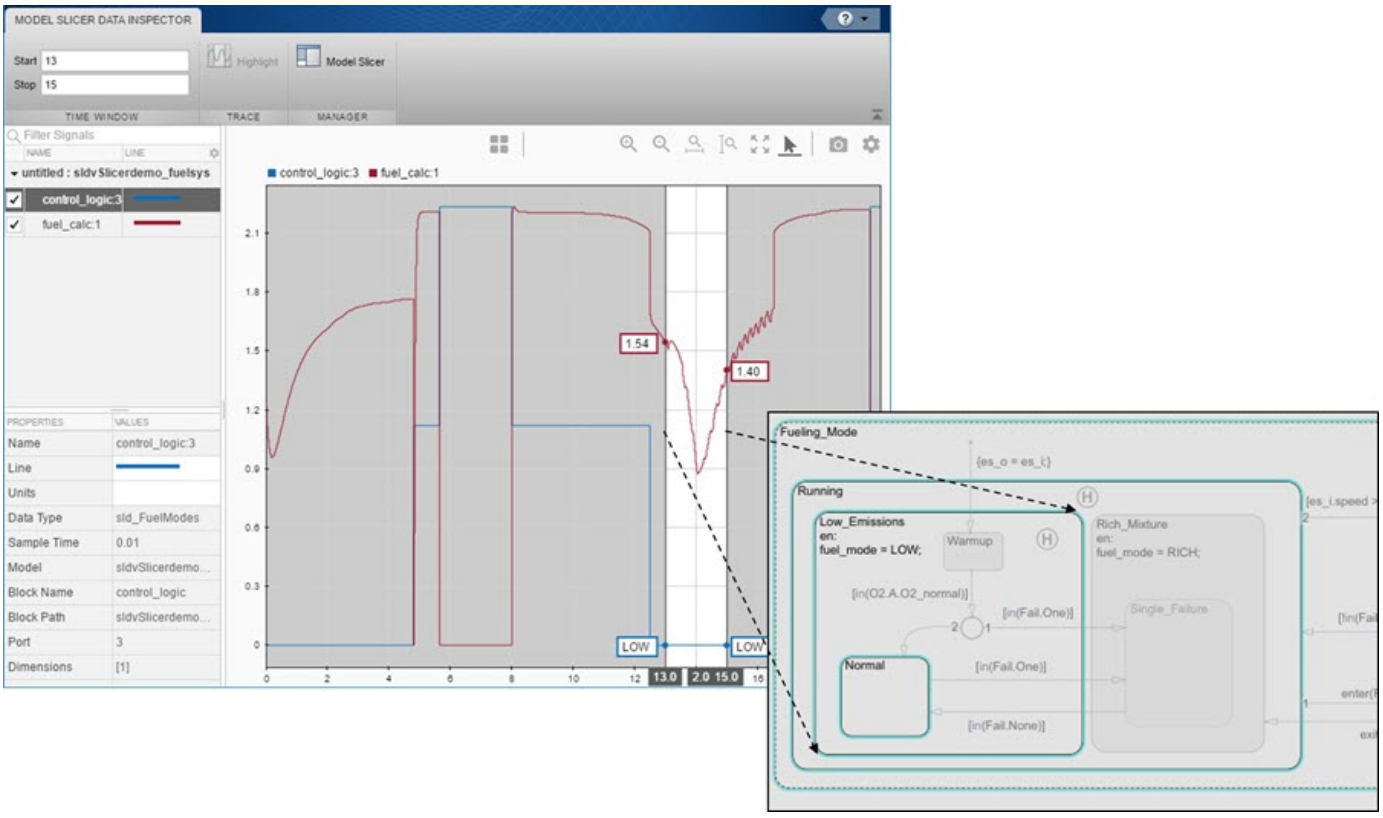
To open the Model Slicer Data Inspector, click **Inspect Signals**.



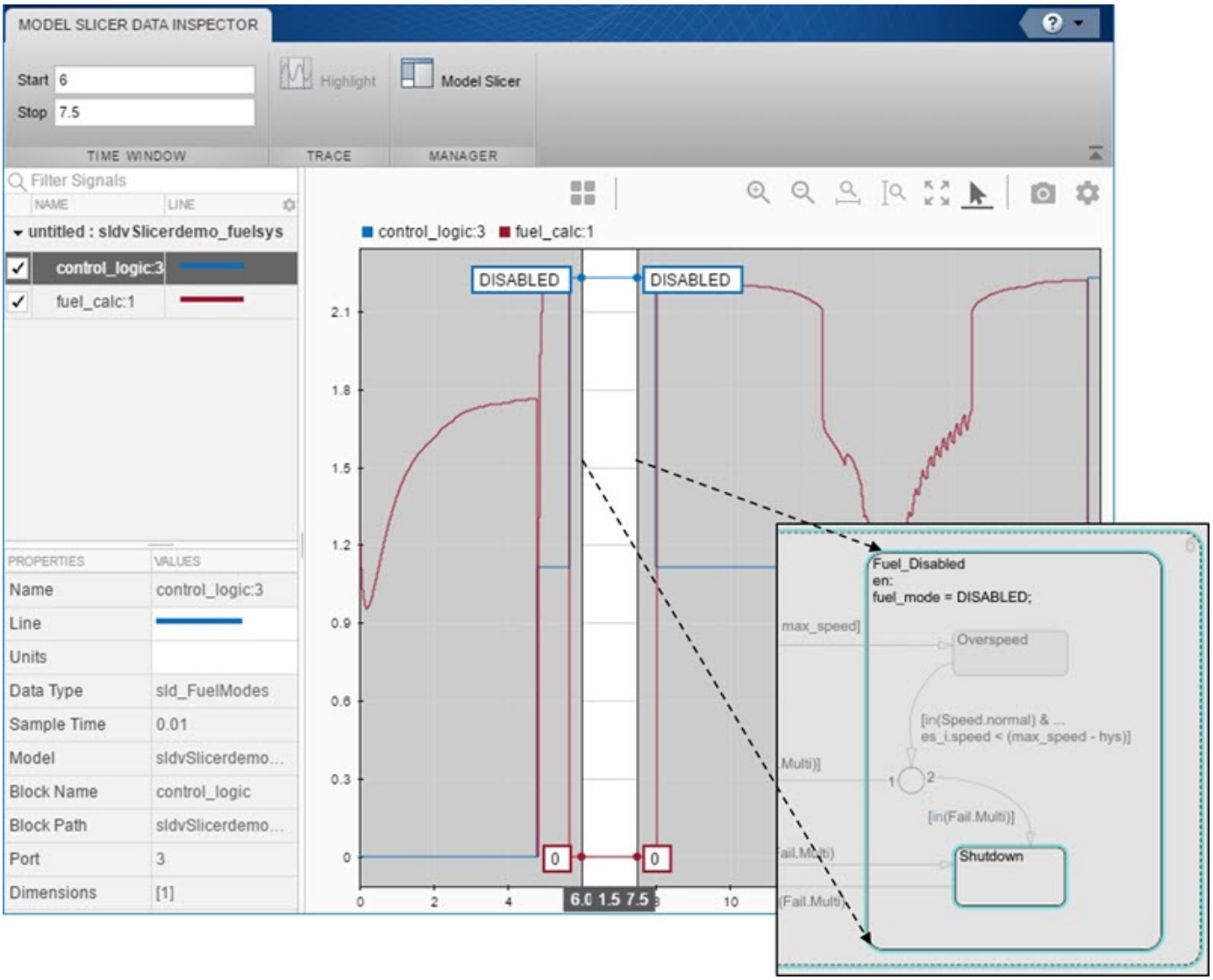
The logged input and output signals appear in the Model Slicer Data Inspector. When you open the Model Slicer Data Inspector, Model Slicer saves the existing Simulation Data Inspector session as MLDATX-file in the current working directory.

You can select the time window by dragging the data cursors to a specific location or by specifying the **Start** and **Stop** time in the navigation pane. To highlight the model for the defined simulation time window, Click **Highlight**.

To investigate the `Fueling_mode`, open the `control_logic` Stateflow™ chart, available in the `fuel_rate_control` referenced model. Select the time window for 13–15 seconds and click **Highlight**. For the defined simulation time window, the `Low_Emissions` fueling mode is active and highlighted.



Select the data cursor for the time window 6–7.5 seconds, with `0 fuel_cal:1`. Click **Highlight**. In the `control_logic` model, the `Fuel_Disabled` state is highlighted. The engine is in `Shutdown` mode.



See Also

“Highlight Functional Dependencies” on page 8-2 | “Refine Highlighted Model” on page 8-12

Debug Slice Simulation by Using Fast Restart Mode

Perform multiple slicer simulations and streamline model debugging workflows by using Model Slicer in fast restart mode. For more information, see “Get Started with Fast Restart” (Simulink).

If you enable fast restart mode, you can:

- Perform multiple slicer simulations efficiently with different inputs, without recompiling the model.
- Debug a simulation by stepping through the major time steps of a simulation and inspecting how a slice changes. For more information, see “Use Simulation Stepper” (Simulink).

Simulate and Debug a Test Case in a Model Slice

This example shows how the fast restart mode performs slicer simulations with different test case inputs, without recompiling the model. You can simulate a sliced harness model with a test case input and highlight the dependency analysis in the model.

Analyze the highlighted slice by stepping through the time steps. You use the simulation stepper to analyze how the slice changes at each time step.

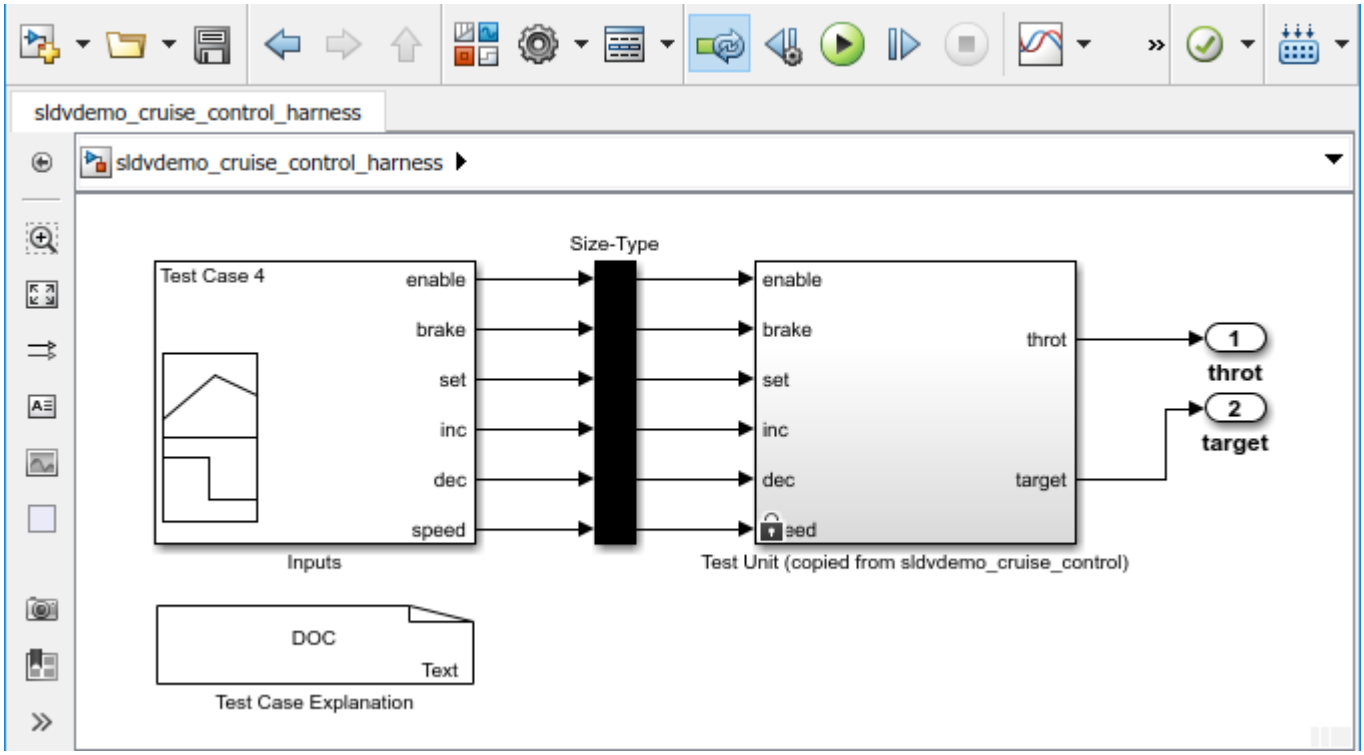
1. Open the `sldvdemo_cruise_control` model.

   ```
   open_system('sldvdemo_cruise_control');
   ```

2. Set `sldvoptions` parameters and analyze the model by using the specified options.

   ```
   opts = sldvoptions;
   opts.Mode = 'TestGeneration';             % Perform test-generation analysis
   opts.ModelCoverageObjectives = 'MCDC';    % Specify type of model coverage
   opts.SaveHarnessModel = 'on';             % Save harness as model file
   [ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
   ```

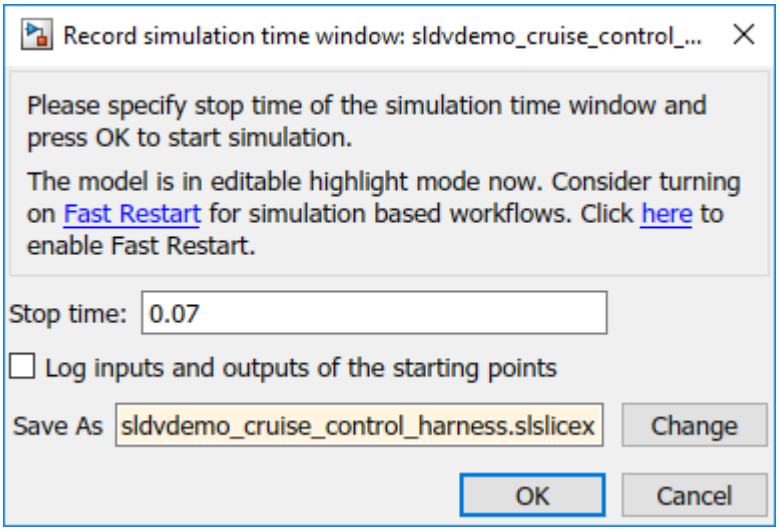
   After the analysis, the software opens a harness model `sldvdemo_cruise_control_harness` and saves it in the default location `<current_folder>\sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_harness.slx`. For more information, see “Simulink Design Verifier Harness Models” (Simulink Design Verifier).



3 To enable the fast restart mode, click **Enable Fast Restart** button .

4 On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**. Model Slicer compiles the model.

Optionally, you can enable fast restart after opening the Model Slice Manager. Select **Simulation time window** and click the run simulation button . To enable fast restart, in the Record simulation time window, click the **here** link.



5 To add **Starting Points**, in the Model Slice Manager, click **Add all outports.**.

The `throt` and `target` outports are added as the **Starting Points**.

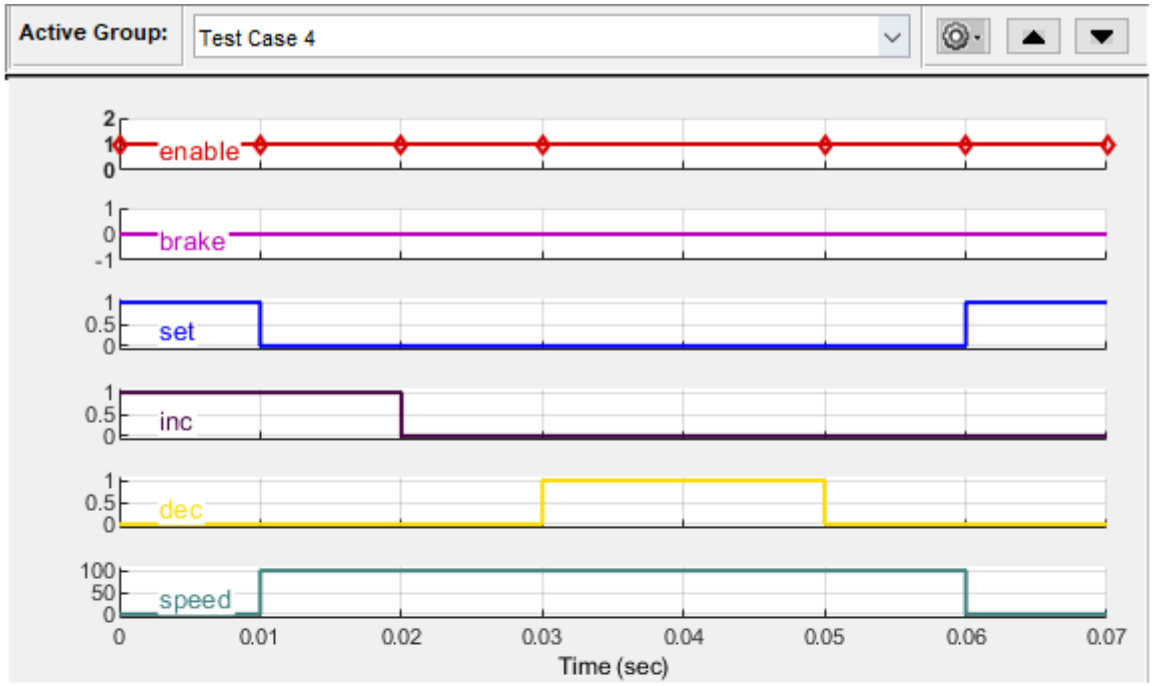
6 You can simulate a test case and analyze the highlighted dependencies in the slice.

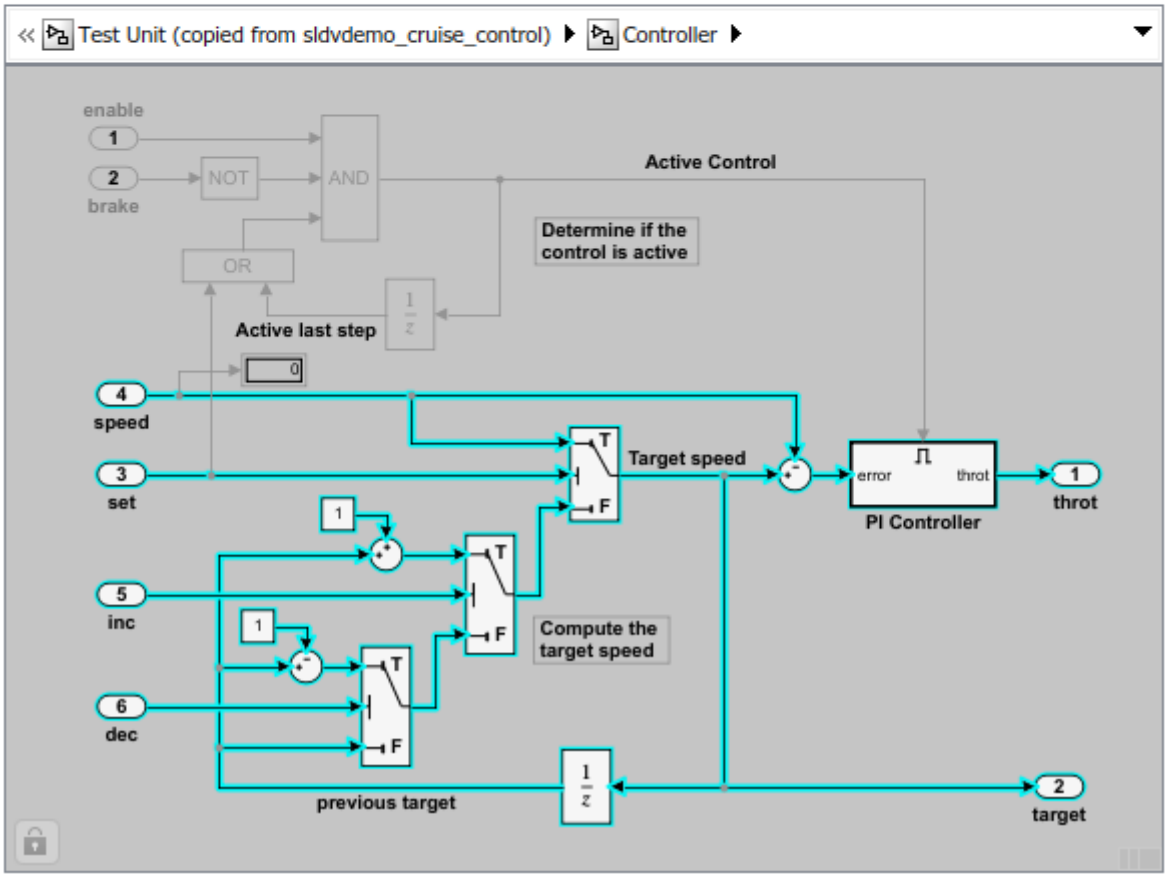
a In the Signal Builder block, select `Test Case 4`.

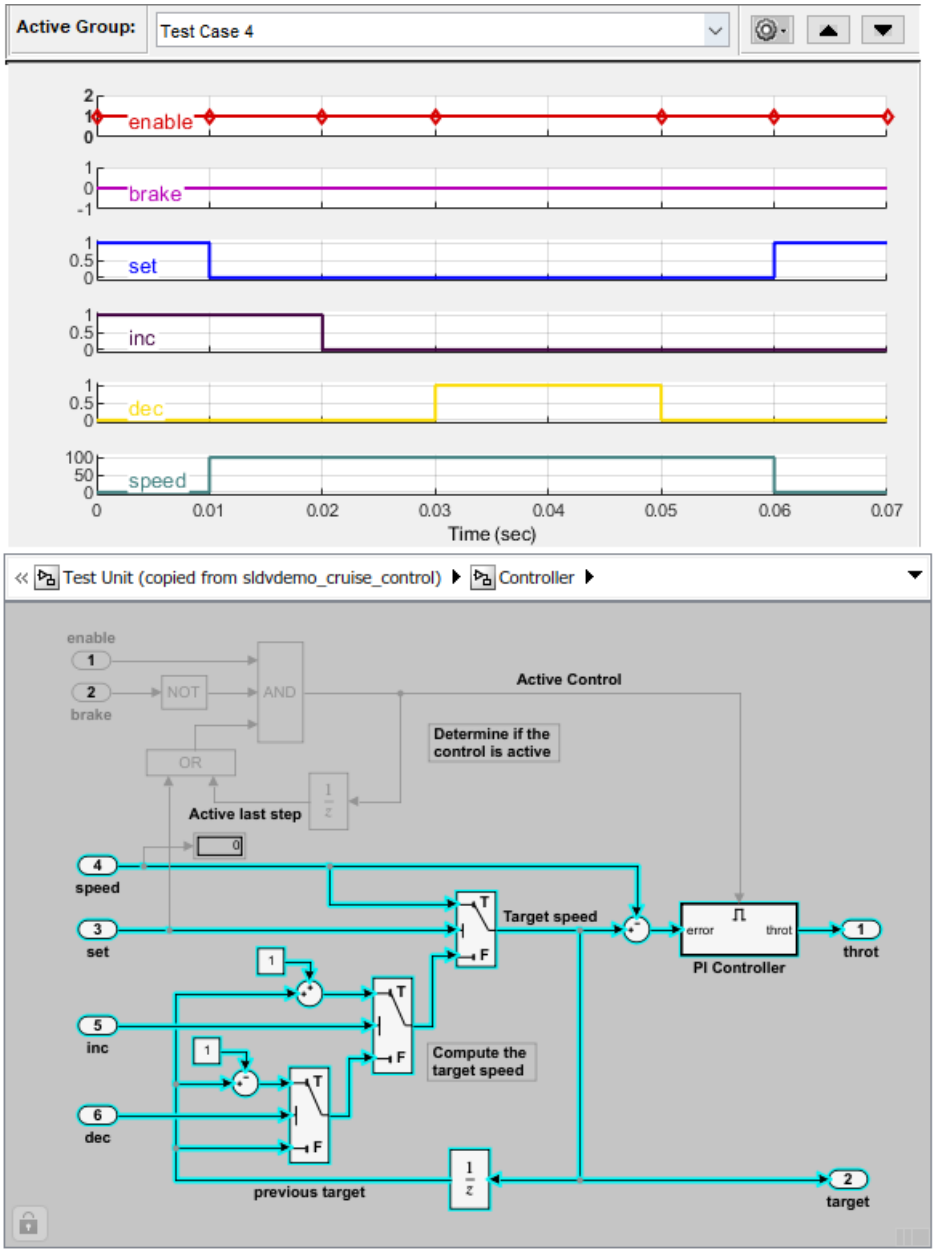
b To simulate the test case, click **Start simulation** button, .

Optionally, you can simulate the model by using the **Run** button in the Simulink editor. You can also simulate by using the **Simulation time window** in the Model Slice Manager.

The slice shows the highlighted dependencies for the `Test Case 4` inputs.

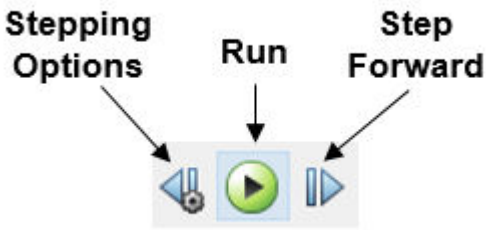

Active Group:
Test Case 4
enable
brake
set
inc
dec
speed
Time (sec)


Test Unit (copied from sldvdemo_cruise_control)
Controller
enable
brake
NOT
AND
OR
Active Control
Determine if the control is active
Active last step
speed
set
inc
dec
Target speed
Compute the target speed
PI Controller
error
throt
previous target
target

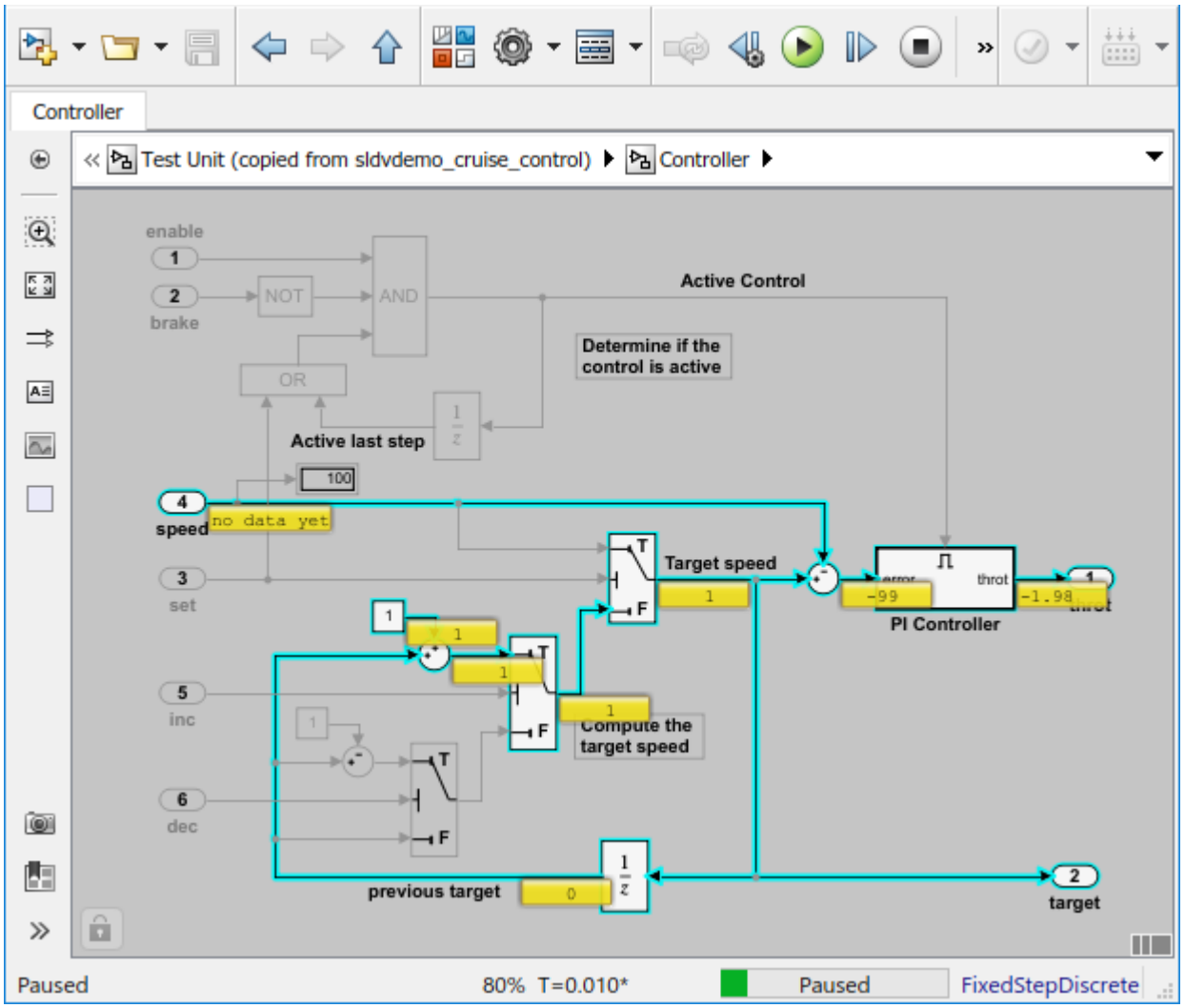


You can simulate a slice for different test case inputs and analyze the dependency analysis.

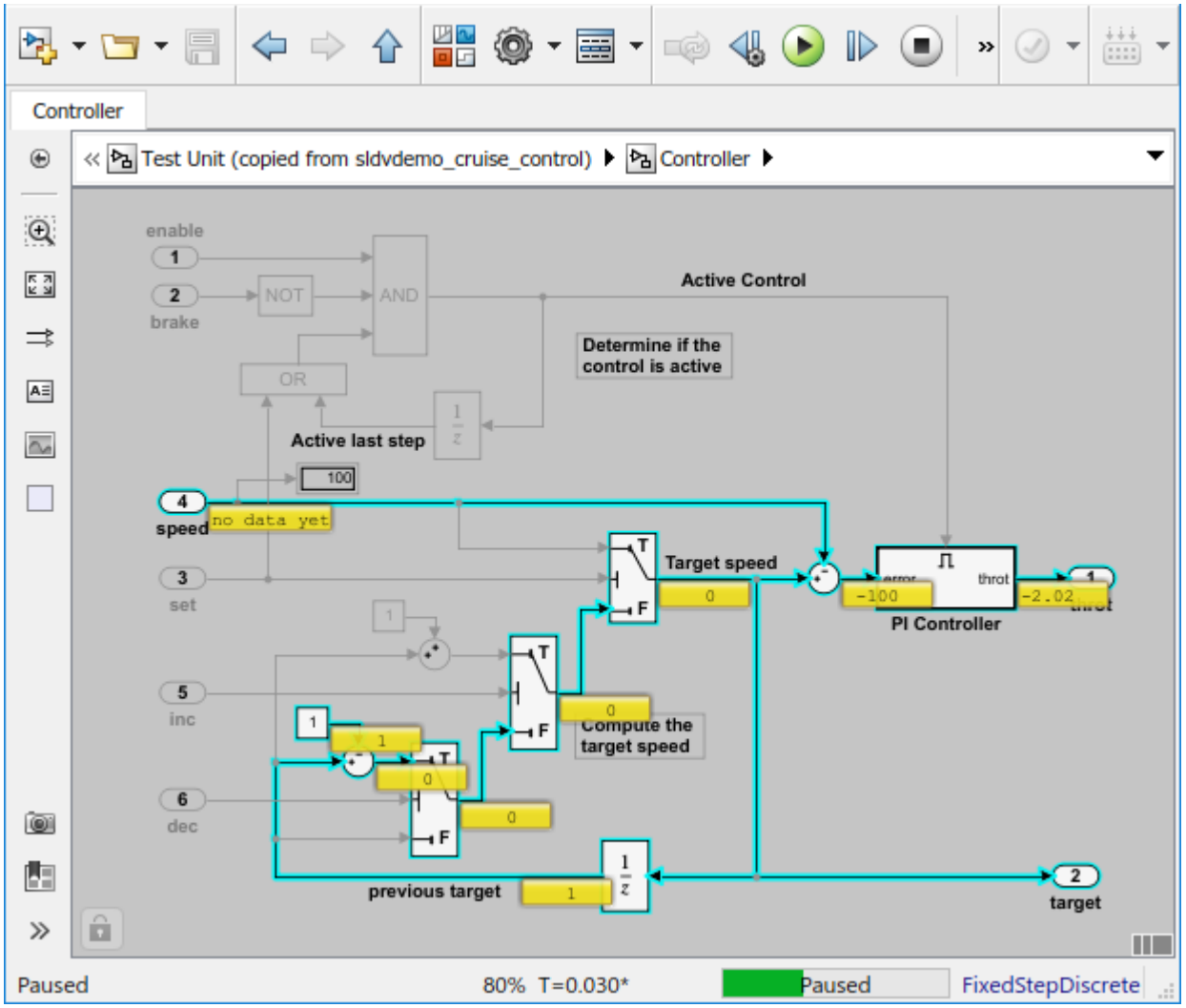
7 Debug a slicer simulation by using a simulation stepper. For more information see, “Simulation Stepper Access” (Simulink).



a To debug the simulation for the test case, in the Simulink Editor for the `sldvdemo_cruise_control_harness` model, click **Step Forward** button. You can view the signal values and the highlighted slice at each time step. For more information, see "Simulation Stepping Options" (Simulink). The signal values and the dependencies at `T=0.010` appears.



b To debug the slice at `T=0.030`, step forward and view the signal values and the highlighted slice.



8 To complete the simulation stepping, click the **Run** button.

See Also

More About

- “Highlight Functional Dependencies” on page 8-2
- “Simulation Stepper” (Simulink)
- “Get Started with Fast Restart” (Simulink)

Isolate Referenced Model for Functional Testing

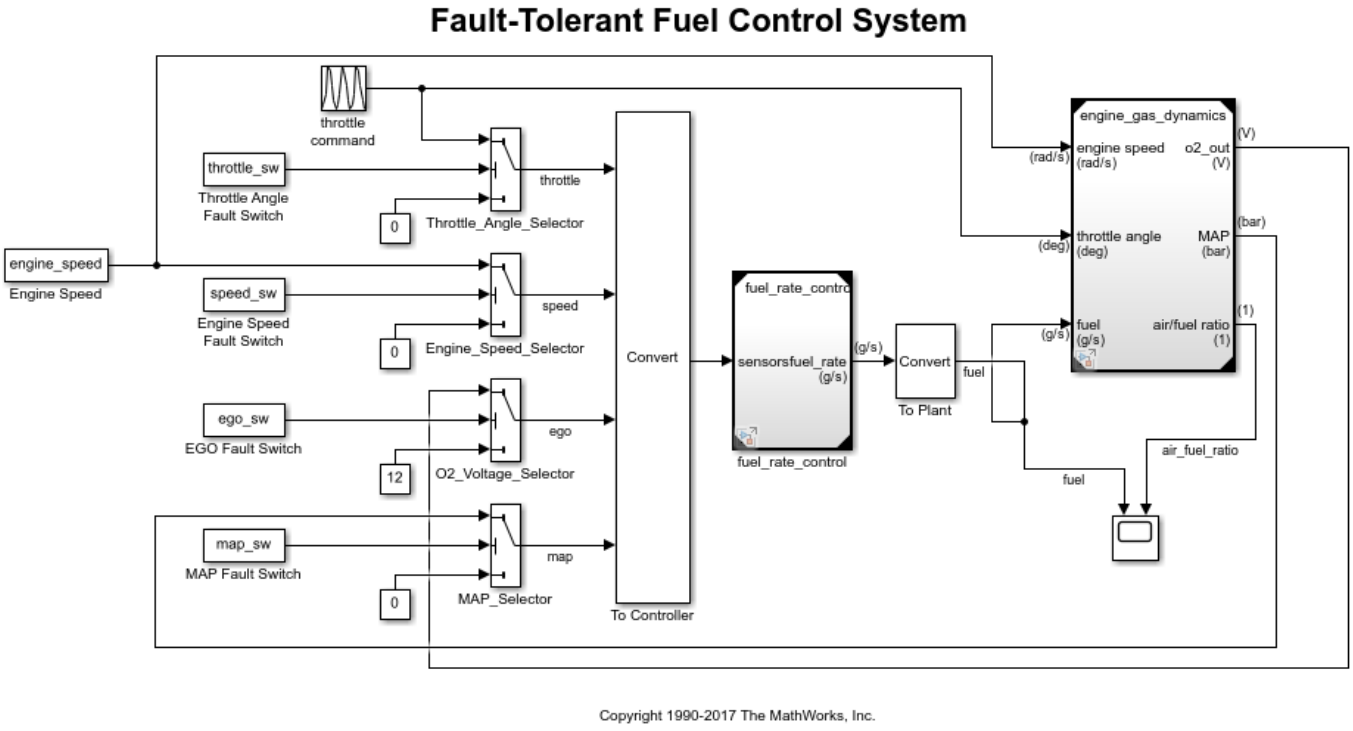
To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

This example shows how to slice the referenced model controller of a fault-tolerant fuel control system for functional testing. To create a simplified open-loop referenced model for debugging and refinement, you generate a slice of the referenced controller.

Step 1: Open the Model

The fault-tolerant fuel control system model contains a referenced model controller `fuel_rate_control`.

```
open_system('sldvSlicerdemo_fuelsys');
```



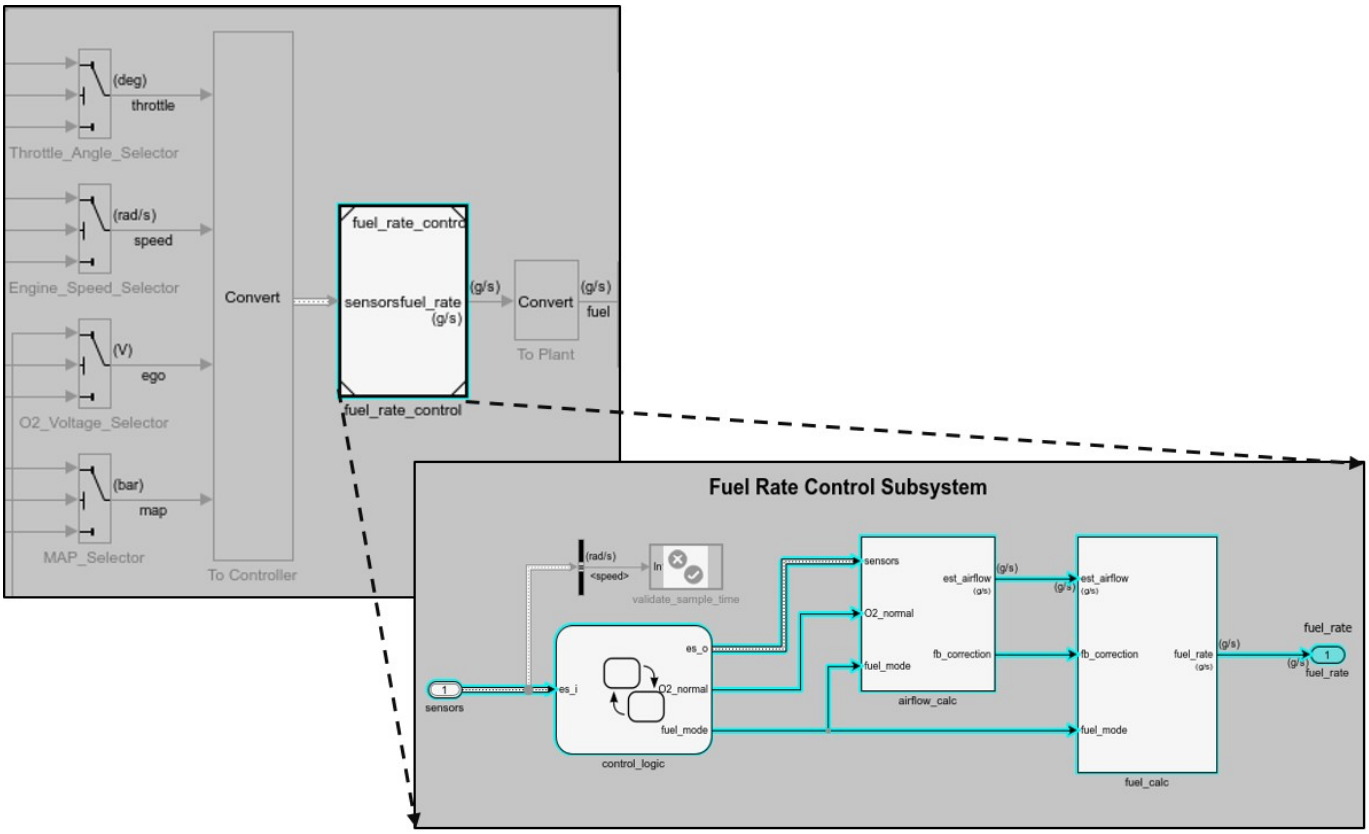
Step 2: Slice the Referenced Model

To analyze the `fuel_rate_control` referenced model, you slice it to create a standalone open-loop model. To open the Model Slice Manager, select **Apps** > **Model Verification, Validation, and Test** > **Model Slicer**, or right-click the `fuel_rate_control` model and select **Model Slicer** > **Slice component**. When you open the Model Slice Manager, the Model Slicer compiles the model. You then configure the model slice properties.

Note: The simulation mode of the `sldvSlicerdemo_fuelsys` model is `Accelerator` mode. When you slice the referenced model, the software configures the simulation mode to `Normal` mode and sets it back to its original simulation mode while exiting the Model Slicer.

Step 3: Select Starting Point

Open the `fuel_rate_control` model, right-click the `fuel-rate` port, and select **Model Slicer** > **Add as starting point**. The Model Slicer highlights the upstream constructs that affect the `fuel_rate`.



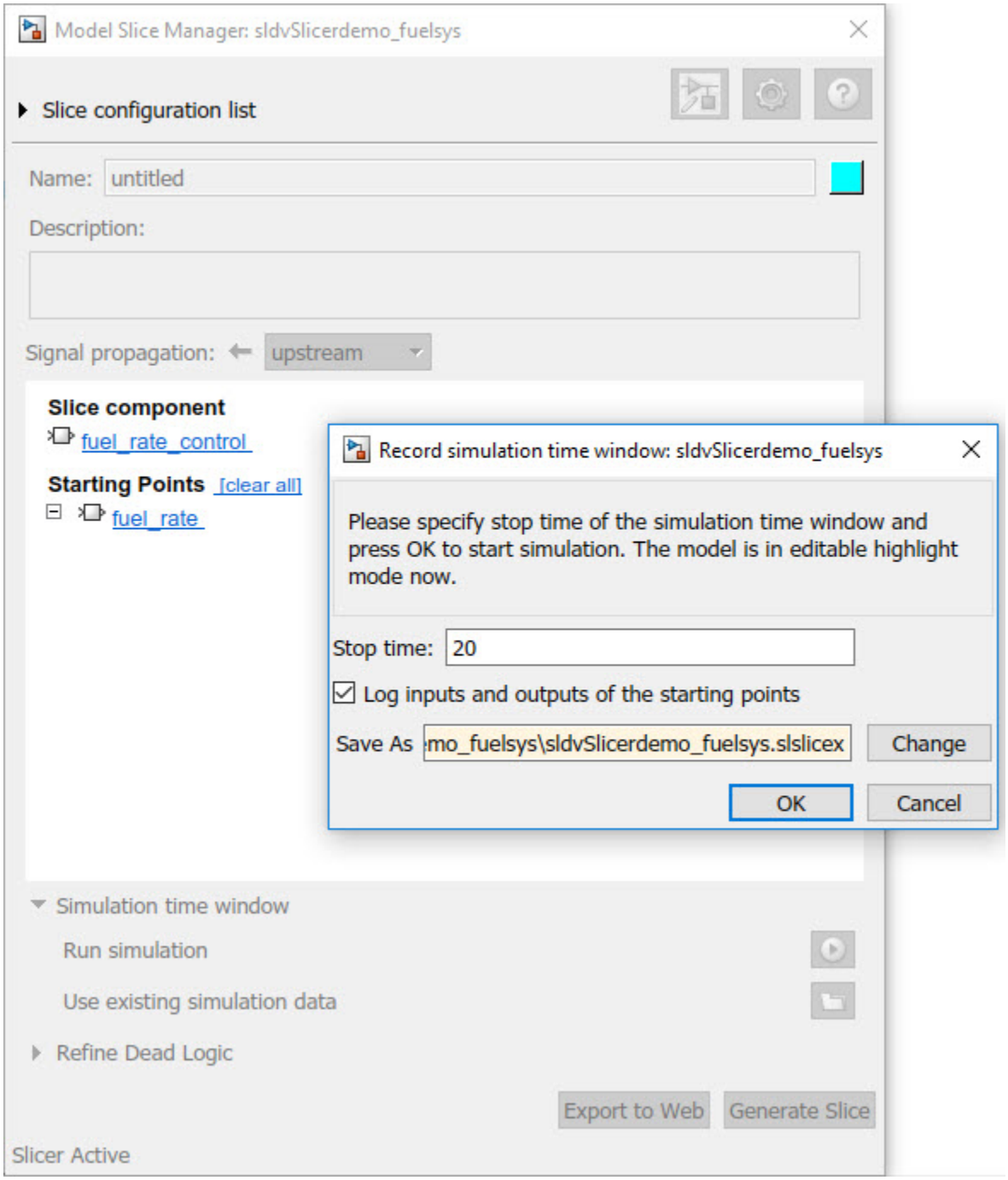
Step 4: Generate Slice

a. In the Model Slice Manager dialog box, select the **Simulation time window**.

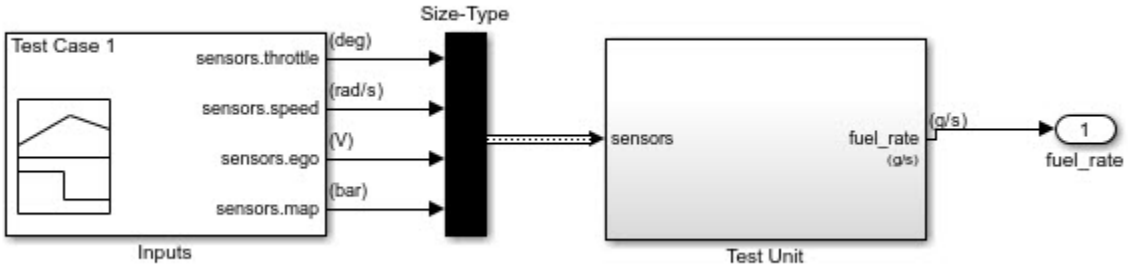
b. Click **Run simulation**.

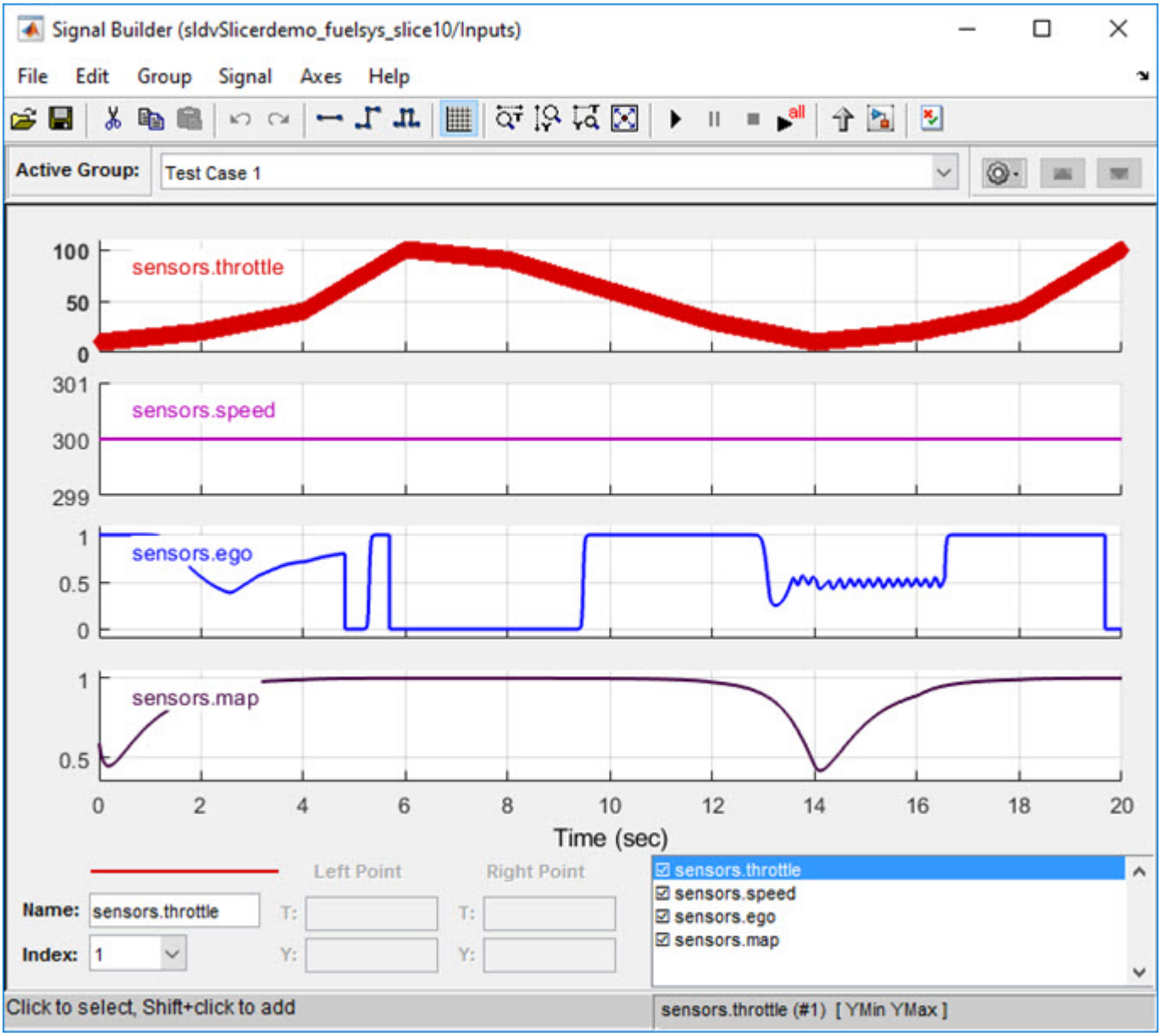
c. For the **Stop time**, enter `20`. Click **OK**.

d. Click **Generate Slice**. The software simulates the sliced referenced model by using the inputs of the close-loop `sldvSlicerdemo_fuelsys` model.



For the sliced model, in the Signal Builder window, one test case is displayed that represents the signals input to the referenced model for simulation time 0–20 seconds.


Size-Type
Test Case 1
sensors.throttle
(deg)
sensors.speed
(rad/s)
sensors.ego
(V)
sensors.map
(bar)
Inputs
sensors
fuel_rate
(g/s)
Test Unit
(g/s)
1
fuel_rate


Signal Builder (sldvSlicerdemo_fuelsys_slice10/Inputs)
File Edit Group Signal Axes Help
Active Group: Test Case 1
sensors.throttle
sensors.speed
sensors.ego
sensors.map
Time (sec)
Left Point
Right Point
Name: sensors.throttle
Index: 1
sensors.throttle
sensors.speed
sensors.ego
sensors.map
Click to select, Shift+click to add
sensors.throttle (#1) [YMin YMax]

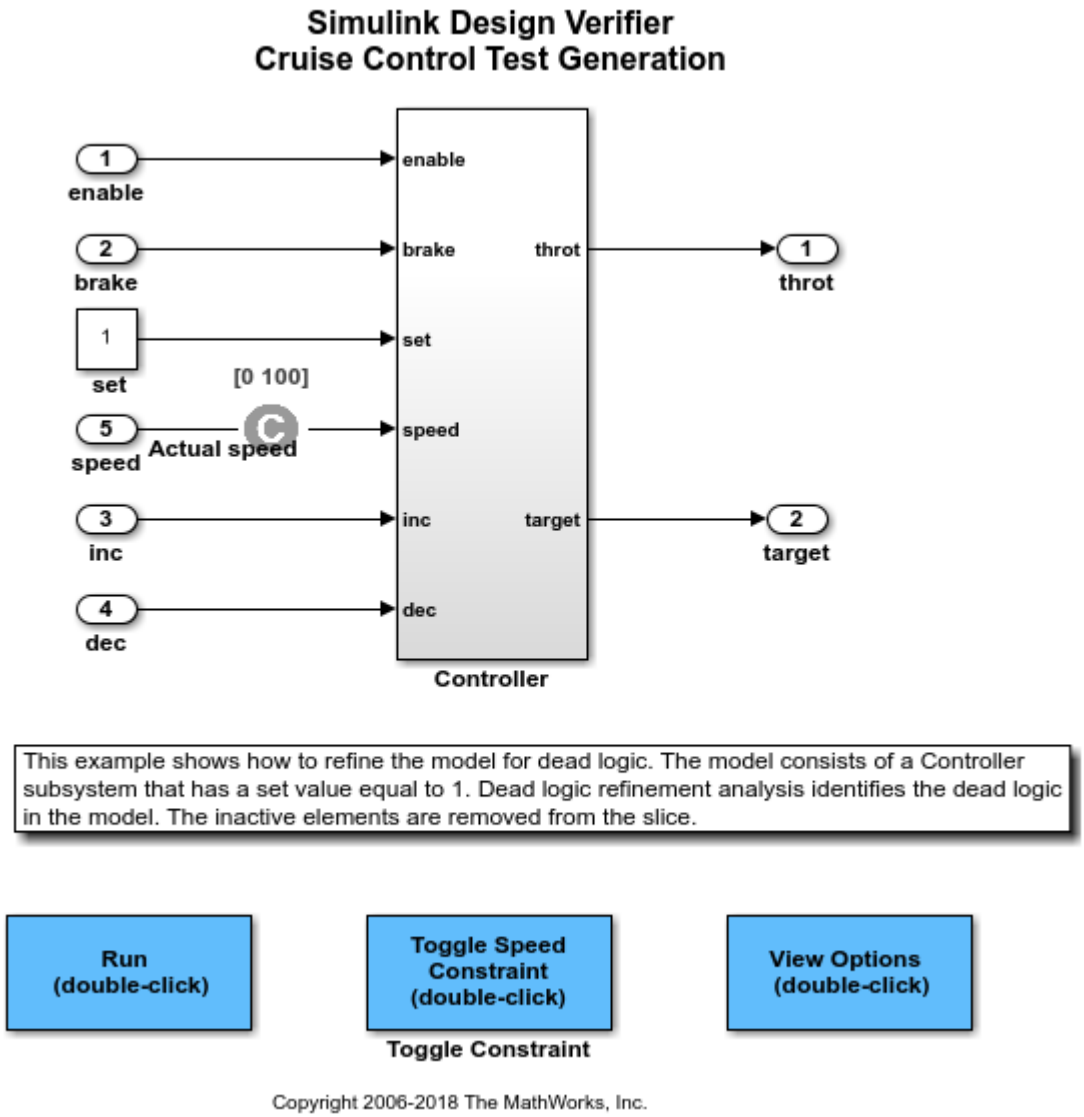
Analyze the Dead Logic

This example shows how to refine the model for dead logic. The `sldvSlicerdemo_dead_logic` model consists of dead logic paths that you refine for dependency analysis.

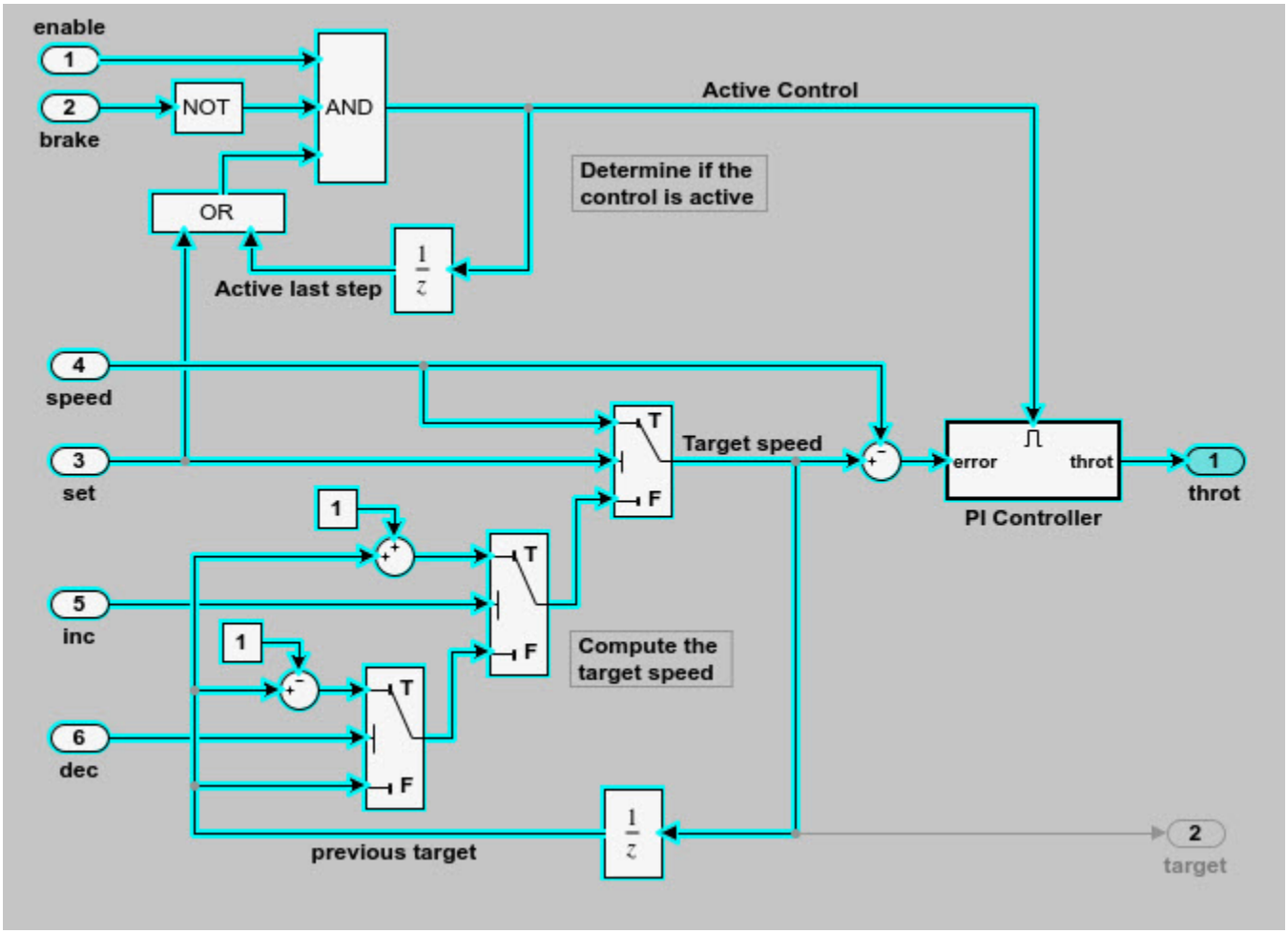
1. Open the `sldvSlicerdemo_dead_logic` model.

2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```



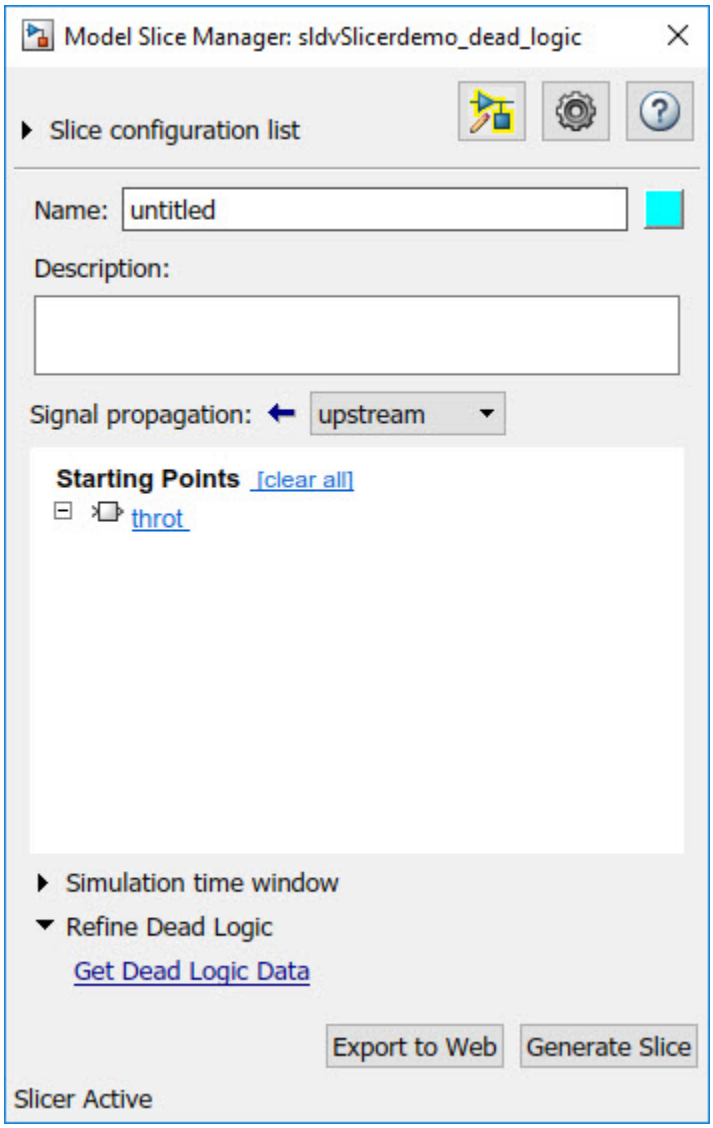
Open the `Controller` subsystem and add the outport `throt` as the starting point.



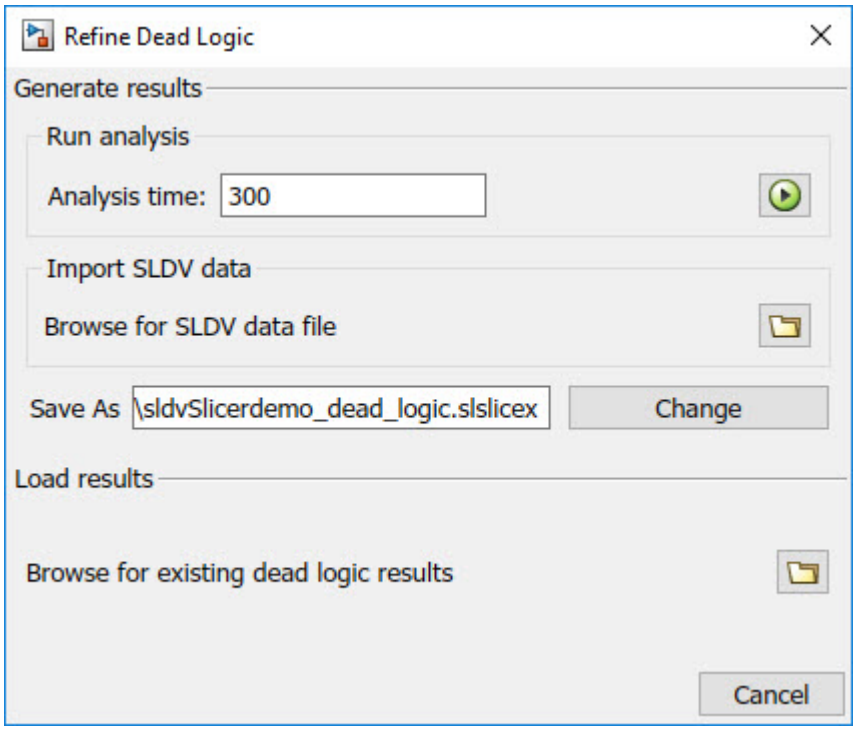
The Model Slicer highlights the upstream dependency of the `throt` outport.

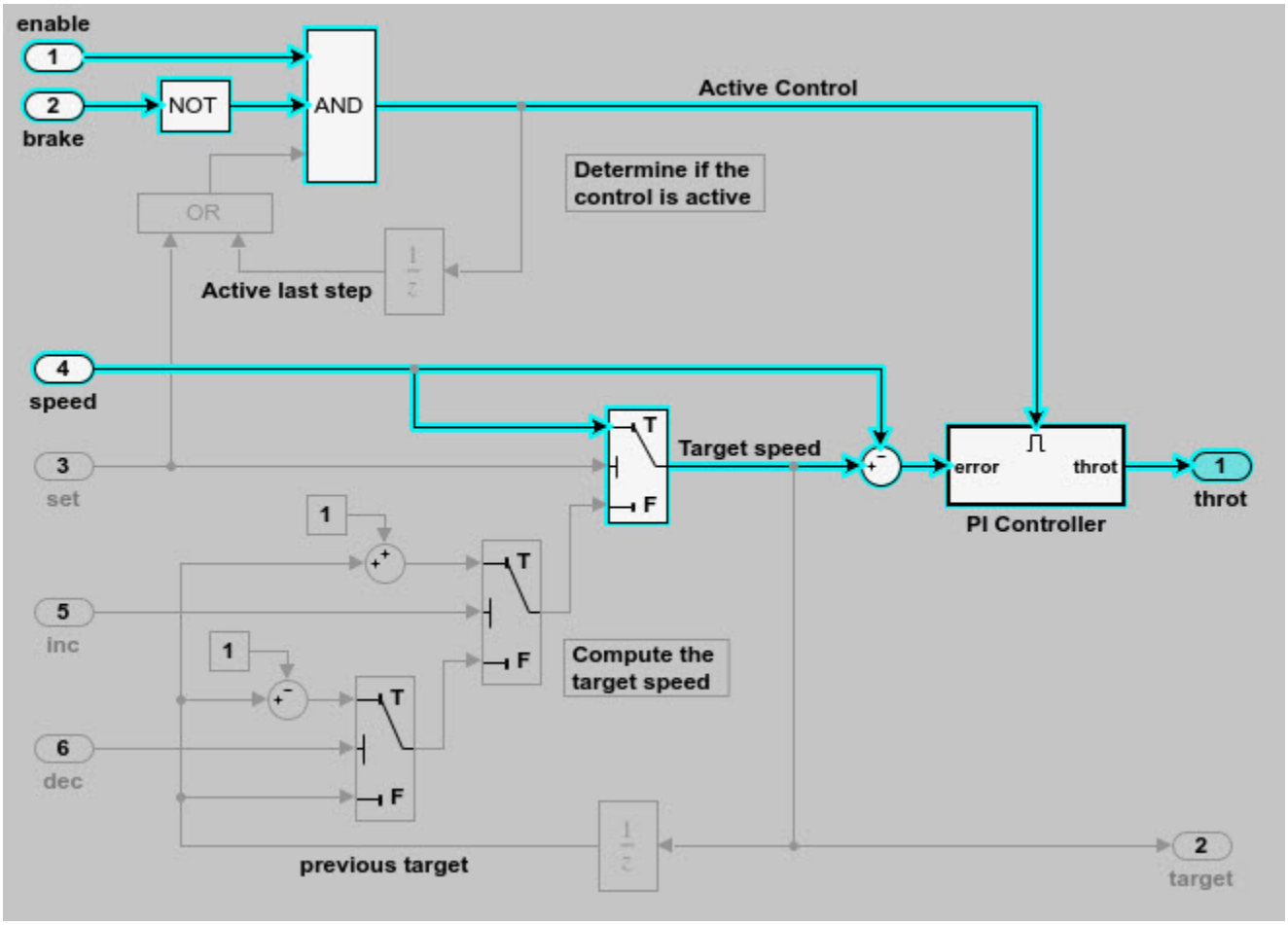
2. In the Model Slice Manager, select **Refine Dead Logic**.

3. Click **Get Dead Logic Data**.



4. Specify the **Analysis time** and run the analysis. You can import existing dead logic results from the `sldvData` file or load existing `.slslicex` data for analysis. For more information, see “Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results” on page 8-63.


Refine Dead Logic
Generate results
Run analysis
Analysis time:
300
Import SLDV data
Browse for SLDV data file
Save As
\sldvSlicerdemo_dead_logic.slslicex
Change
Load results
Browse for existing dead logic results
Cancel



As the `set` input is equal to `true`, the `False` input to switch is removed for dependency analysis. Similarly, the output of block `OR` is always `true` and removed from the model slice.

Investigate Highlighted Model Slice by Using Model Slicer Data Inspector

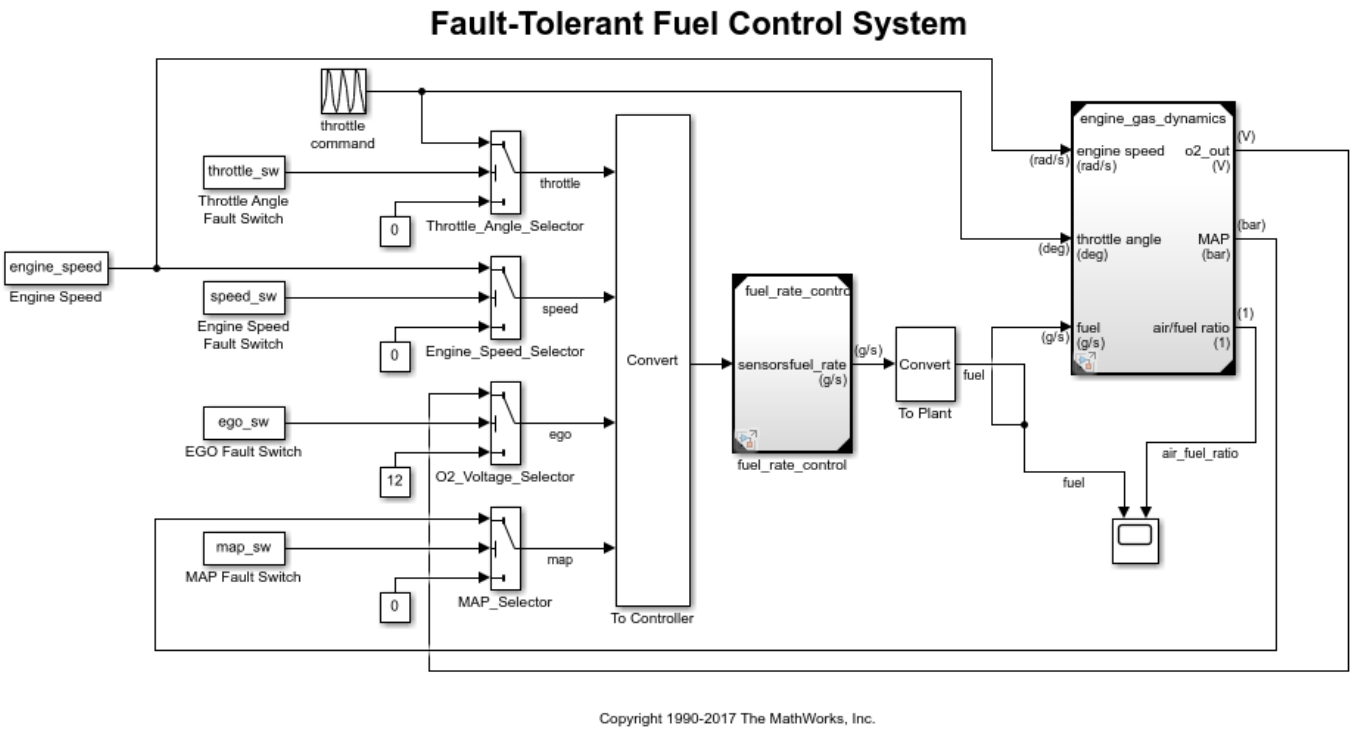
This example shows how to investigate and refine the highlighted model slice by using the Model Slicer Data Inspector.

In the fault-tolerant fuel control system, the `control_logic` controls the fueling mode of the engine. In this example, you slice the `fuel_rate_control` referenced model. Then, investigate the effect of `fuel_rate_ratio` on the `Fueling_mode` of the engine. For more information, see “Modeling a Fault-Tolerant Fuel Control System” (Simulink).

Step 1: Start the Model Slice Manager

To start the Model Slice Manager, open the `fuel_rate_control` model, and select **Apps** > **Model Verification, Validation, and Test** > **Model Slicer**.

```
open_system('sldvSlicerdemo_fuelsys');
```



To select the starting point, open the `fuel_rate_control` model, and add the `fuel-rate` port and the `fuel_mode` output signal as the starting point. To add a port or a signal as a starting point, right-click the port or signal, and select **Model Slicer** > **Add as Starting Point**.

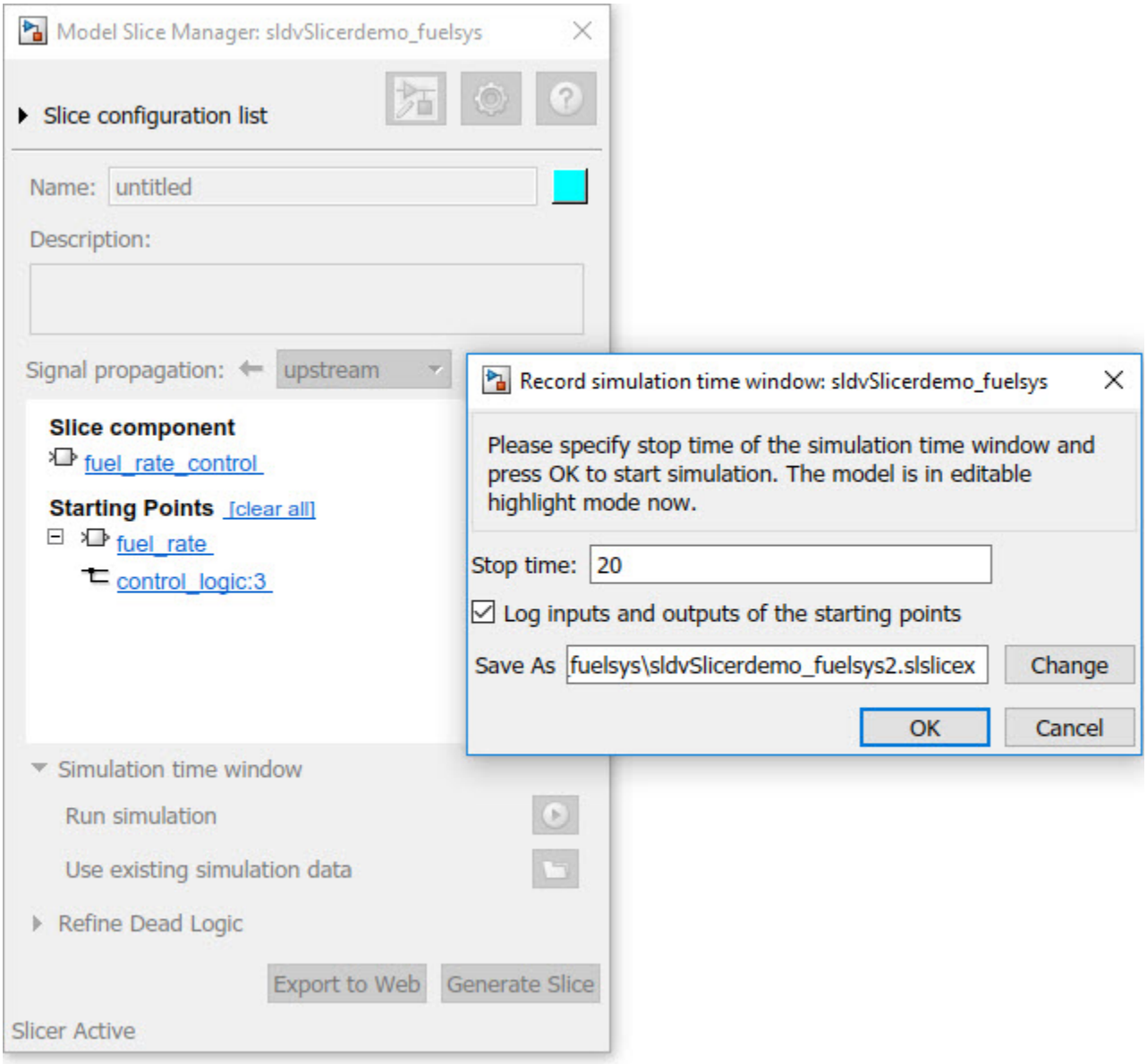
Step 2: Log input and output signals

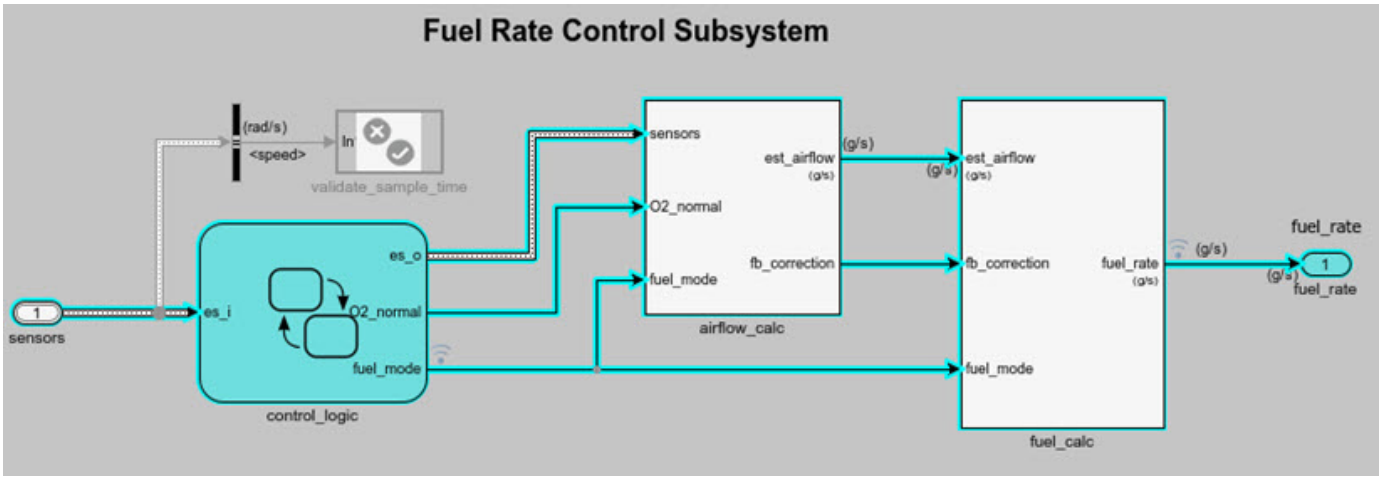
a. In the Model Slice Manager dialog box, select the **Simulation time window** and **Run simulation**.

b. In the Record simulation time window, for the **Stop time**, type `20`.

c. Select the **Log inputs and outputs of the starting points**.

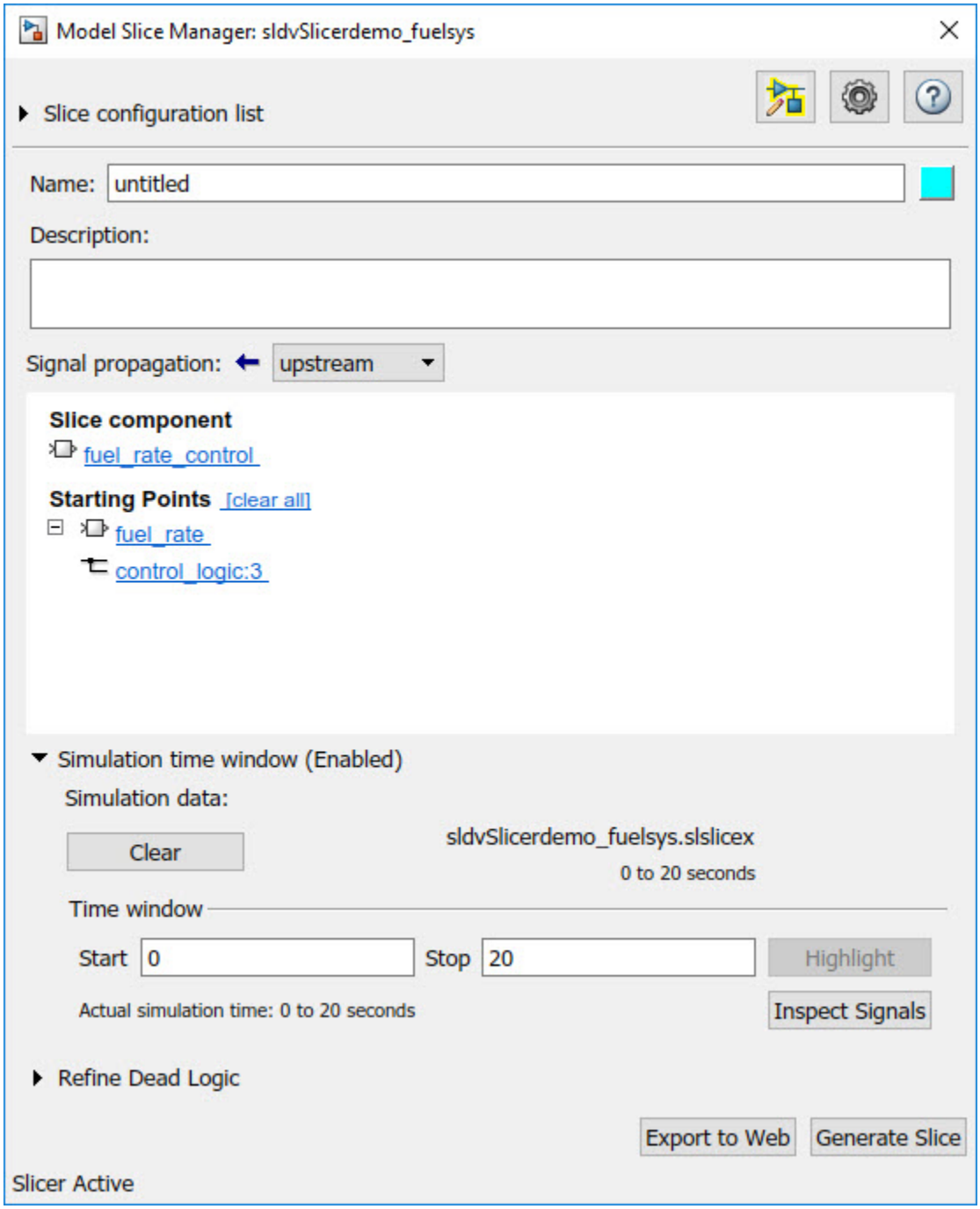
d. Click **OK**.


Model Slice Manager: sldvSlicerdemo_fuelsys
Slice configuration list
Name: untitled
Description:
Signal propagation: upstream
Slice component
fuel_rate_control
Starting Points [clear all]
fuel_rate
control_logic:3
Simulation time window
Run simulation
Use existing simulation data
Refine Dead Logic
Export to Web
Generate Slice
Slicer Active
Record simulation time window: sldvSlicerdemo_fuelsys
Please specify stop time of the simulation time window and press OK to start simulation. The model is in editable highlight mode now.
Stop time: 20
Log inputs and outputs of the starting points
Save As fuelsys\sldvSlicerdemo_fuelsys2.slslicex
Change
OK
Cancel



Step 3: Inspect signals

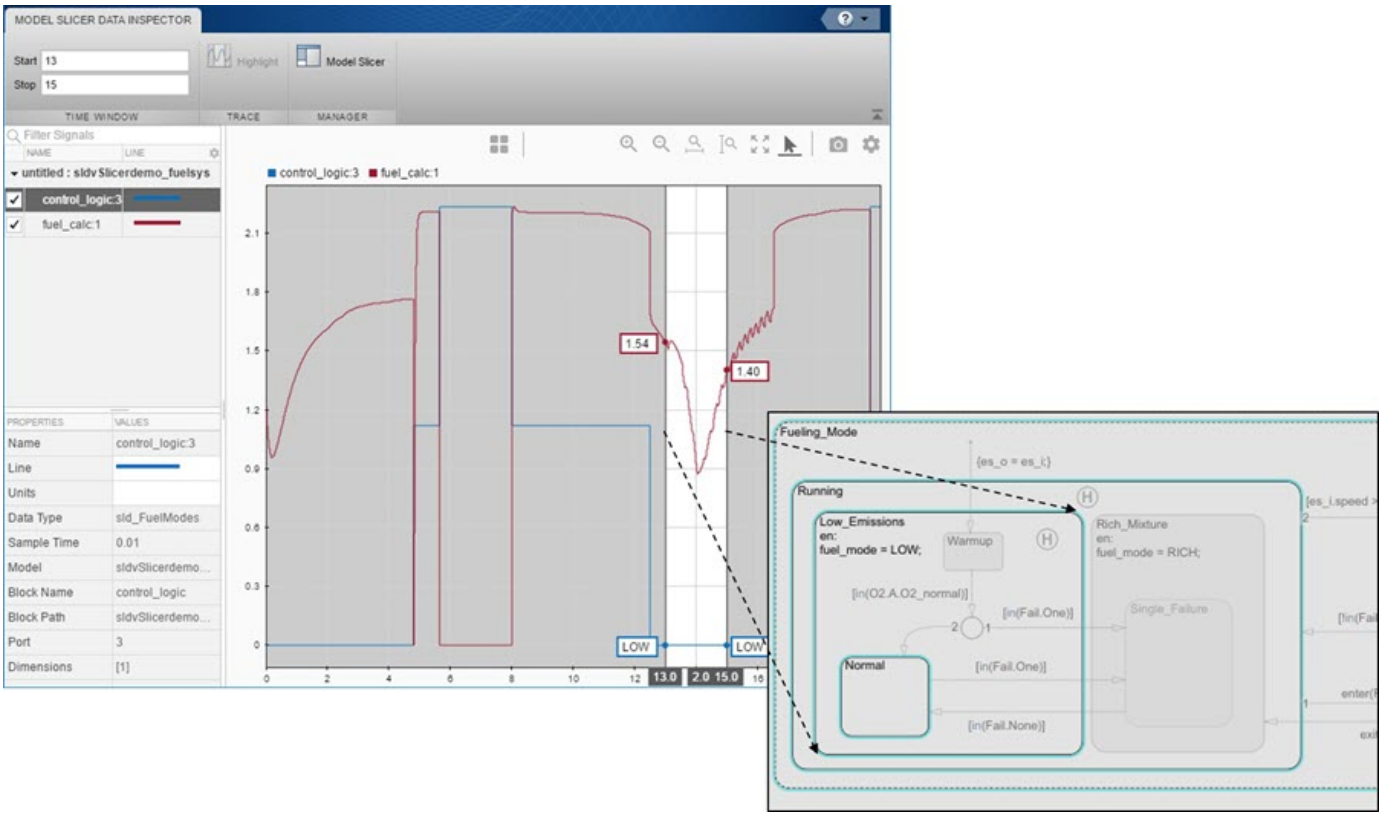
To open the Model Slicer Data Inspector, click **Inspect Signals**.



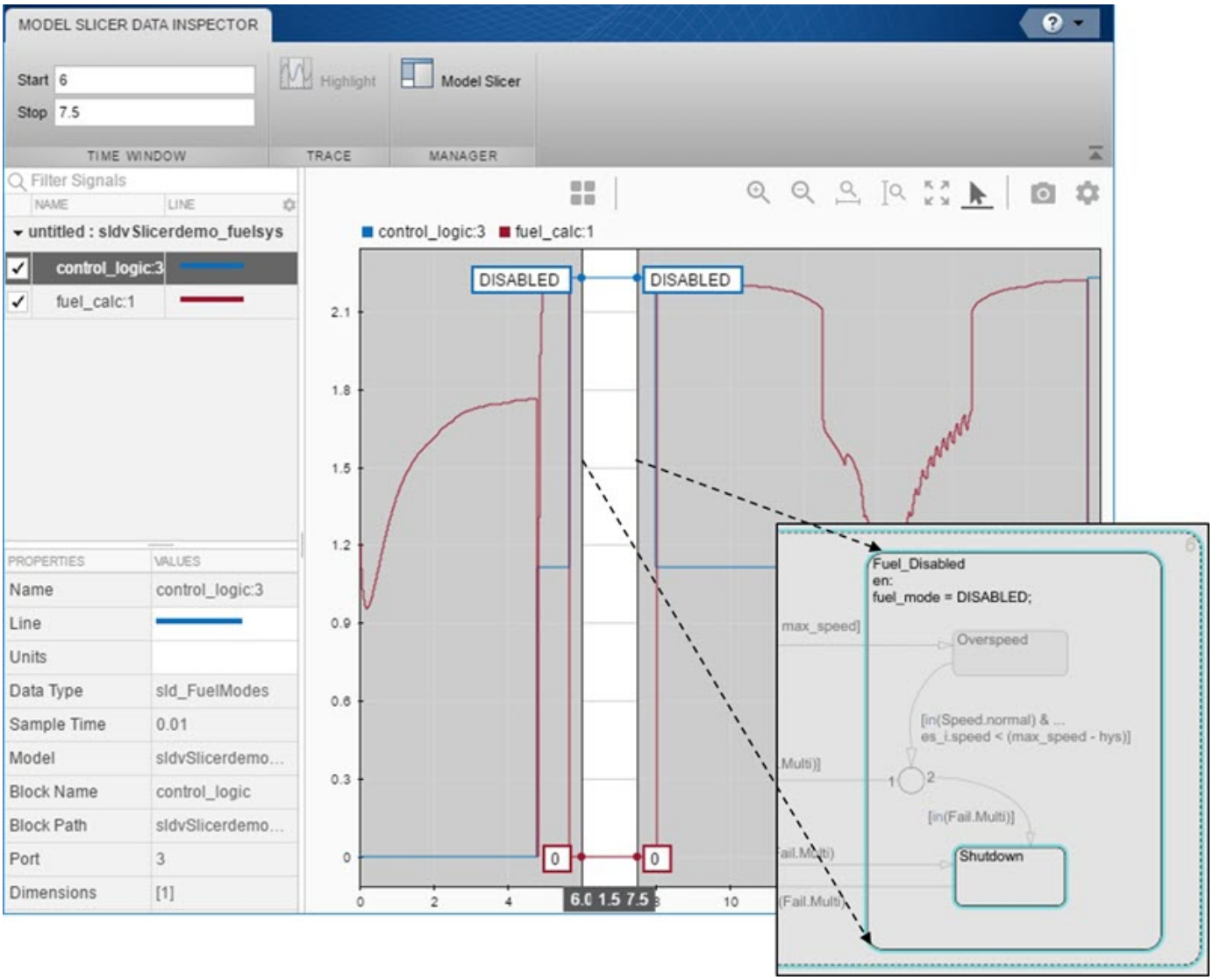
The logged input and output signals appear in the Model Slicer Data Inspector. When you open the Model Slicer Data Inspector, Model Slicer saves the existing Simulation Data Inspector session as MLDATX-file in the current working directory.

You can select the time window by dragging the data cursors to a specific location or by specifying the **Start** and **Stop** time in the navigation pane. To highlight the model for the defined simulation time window, Click **Highlight**.

To investigate the `Fueling_mode`, open the `control_logic` Stateflow™ chart, available in the `fuel_rate_control` referenced model. Select the time window for 13–15 seconds and click **Highlight**. For the defined simulation time window, the `Low_Emissions` fueling mode is active and highlighted.



Select the data cursor for the time window 6–7.5 seconds, with `0 fuel_cal:1`. Click **Highlight**. In the `control_logic` model, the `Fuel_Disabled` state is highlighted. The engine is in `Shutdown` mode.


MODEL SLICER DATA INSPECTOR
Start 6
Stop 7.5
TIME WINDOW
Highlight
TRACE
Model Slicer
MANAGER
Filter Signals
NAME
LINE
untitled : sldvSlicerdemo_fuelsys
control_logic:3
fuel_calc:1
PROPERTIES
VALUES
Name control_logic:3
Line
Units
Data Type sld_FuelModes
Sample Time 0.01
Model sldvSlicerdemo...
Block Name control_logic
Block Path sldvSlicerdemo...
Port 3
Dimensions [1]
control_logic:3
fuel_calc:1
DISABLED
DISABLED
2.1
1.8
1.5
1.2
0.9
0.6
0.3
0
0
0
0
2
4
6.0 1.5 7.5
10
Fuel_Disabled
en:
fuel_mode = DISABLED;
max_speed]
Overspeed
[in(Speed.normal) & ...
es_i.speed < (max_speed - hys)]
Multi)]
1
2
[in(Fail.Multi)]
Shutdown
(Fail.Multi)