Simulink® Check™

User's Guide

MATLAB®&SIMULINK®

R2019a

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

Trademarks

Patents

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)

Contents

1 Getting Started

2 Verification and Validation

3 Checking Systems Interactively

4 Check Systems Programmatically

5 Model Metrics

6 Overview of Customizing the Model Advisor

7 Create Model Advisor Checks

8 Create Custom Configurations by Organizing Checks and Folders

9 Create Procedural-Based Model Advisor Configurations

10 Deploy Custom Configurations

11 Model Slicer

Getting Started

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 Automotive Advisory Board (MAAB) 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 MAAB 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

Check for Standards Compliance in Your Model

With Simulink Check, the Model Advisor can check for model conditions that cause generation of inefficient code or code unsuitable for safety-critical applications.

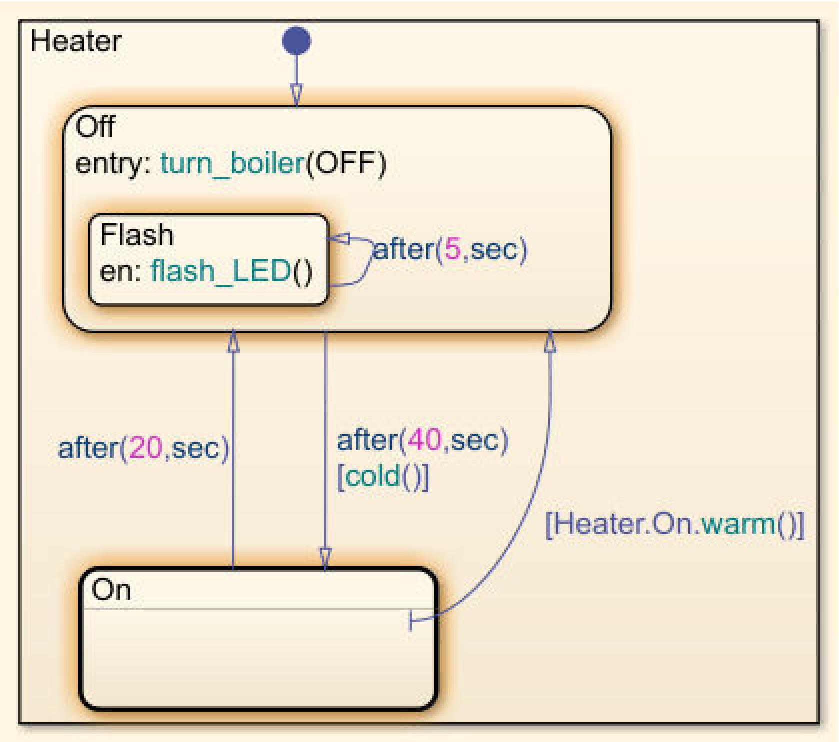
The Model Advisor produces a report that lists the suboptimal conditions or settings that it finds. The Model Advisor proposes better model configuration settings.

Detect and Fix Model Advisor Check Violations by Using Edit-Time Checking

In the Model Advisor, you can check that your model complies with certain guidelines while you edit.

1. Open model `sf_boiler`.
2. To use edit-time checking, go to **Analysis** > **Model Advisor** > **Display Advisor Checks in Editor**.

   The highlighted blocks and subsystems indicate a compliance issues. Place your cursor over the 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. In this case, the warning indicates that the subsystem block name contains incorrect characters.
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 review the issue.



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

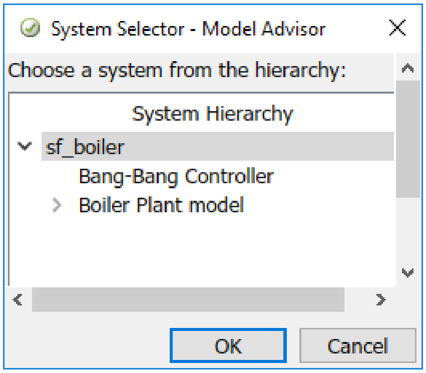
Detect Model Advisor Check Violations Interactively

You can interactively check that your model complies with DO-178C/DO-331 guidelines by using the Model Advisor.

1 Open model `sf_boiler`.

2 In the model window, select **Analysis > Model Advisor > Model Advisor**.

3 Select the top-level model `sf_boiler` from the System Hierarchy and click **OK**.



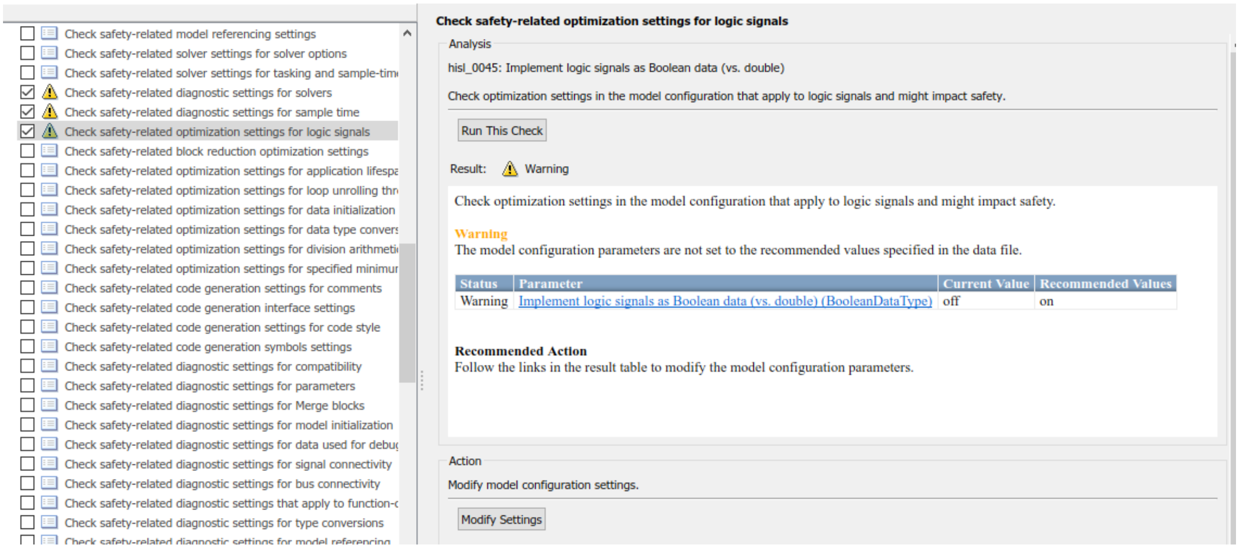
4 In the left pane, in the **By Product > Simulink Check > Modeling Standards >DO-178C/DO-331 Checks** folder, select:

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

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

Update Model to Reach Compliance

1 To review the configuration parameters that are not set to the recommended values, click **Check safety-related optimization settings for logical signals**.



- □ Check safety-related model referencing settings
- □ Check safety-related solver settings for solver options
- □ Check safety-related solver settings for tasking and sample-tim
- ☑ ⚠ Check safety-related diagnostic settings for solvers
- ☑ ⚠ Check safety-related diagnostic settings for sample time
- ☑ ⚠ Check safety-related optimization settings for logic signals
- □ Check safety-related block reduction optimization settings
- □ Check safety-related optimization settings for application lifesp
- □ Check safety-related optimization settings for loop unrolling thr
- □ Check safety-related optimization settings for data initialization
- □ Check safety-related optimization settings for data type conver
- □ Check safety-related optimization settings for division arithmeti
- □ Check safety-related optimization settings for specified minimu
- □ Check safety-related code generation settings for comments
- □ Check safety-related code generation interface settings
- □ Check safety-related code generation settings for code style
- □ Check safety-related code generation symbols settings
- □ Check safety-related diagnostic settings for compatibility
- □ Check safety-related diagnostic settings for parameters
- □ Check safety-related diagnostic settings for Merge blocks
- □ Check safety-related diagnostic settings for model initialization
- □ Check safety-related diagnostic settings for data used for debu
- □ Check safety-related diagnostic settings for signal connectivity
- □ Check safety-related diagnostic settings for bus connectivity
- □ Check safety-related diagnostic settings that apply to function-
- □ Check safety-related diagnostic settings for type conversions
- □ Check safety-related diagnostic settings for model referencing

Check safety-related optimization settings for logic signals

Analysis

hisl_0045: Implement logic signals as Boolean data (vs. double)

Check optimization settings in the model configuration that apply to logic signals and might impact safety.

Result: ⚠ Warning

Check optimization settings in the model configuration that apply to logic signals and might impact safety.

Warning

The model configuration parameters are not set to the recommended values specified in the data file.

Status	Parameter	Current Value	Recommended Values
Warning	Implement logic signals as Boolean data (vs. double) (BooleanDataType)	off	on

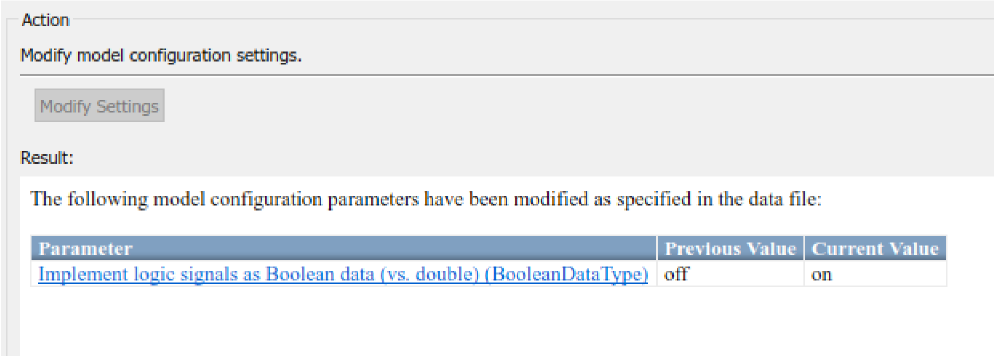
Recommended Action

Follow the links in the result table to modify the model configuration parameters.

Action

Modify model configuration settings.

2 To update the optimization parameters to the recommended values, click the **Modify Settings** button in the **Action** section of the right pane. The Model Advisor updates the parameters to the recommended value and details the results.



Action

Modify model configuration settings.

Result:

The following model configuration parameters have been modified as specified in the data file:

Parameter	Previous Value	Current Value
Implement logic signals as Boolean data (vs. double) (BooleanDataType)	off	on

3 Repeat steps 1 and 2 for the other two checks: **Check safety-related diagnostic settings for solvers** and **Check safety-related diagnostic settings for sample time**.

4 To verify that your model now passes, rerun the selected checks.

Display an HTML Report of Check Results

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**.

See Also

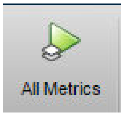
More About

- “Check Model Compliance by Using the Model Advisor” on page 3-2
- “Create Model Advisor Checks Workflow” on page 7-2

Collect Model Metric Data by Using the Metrics Dashboard

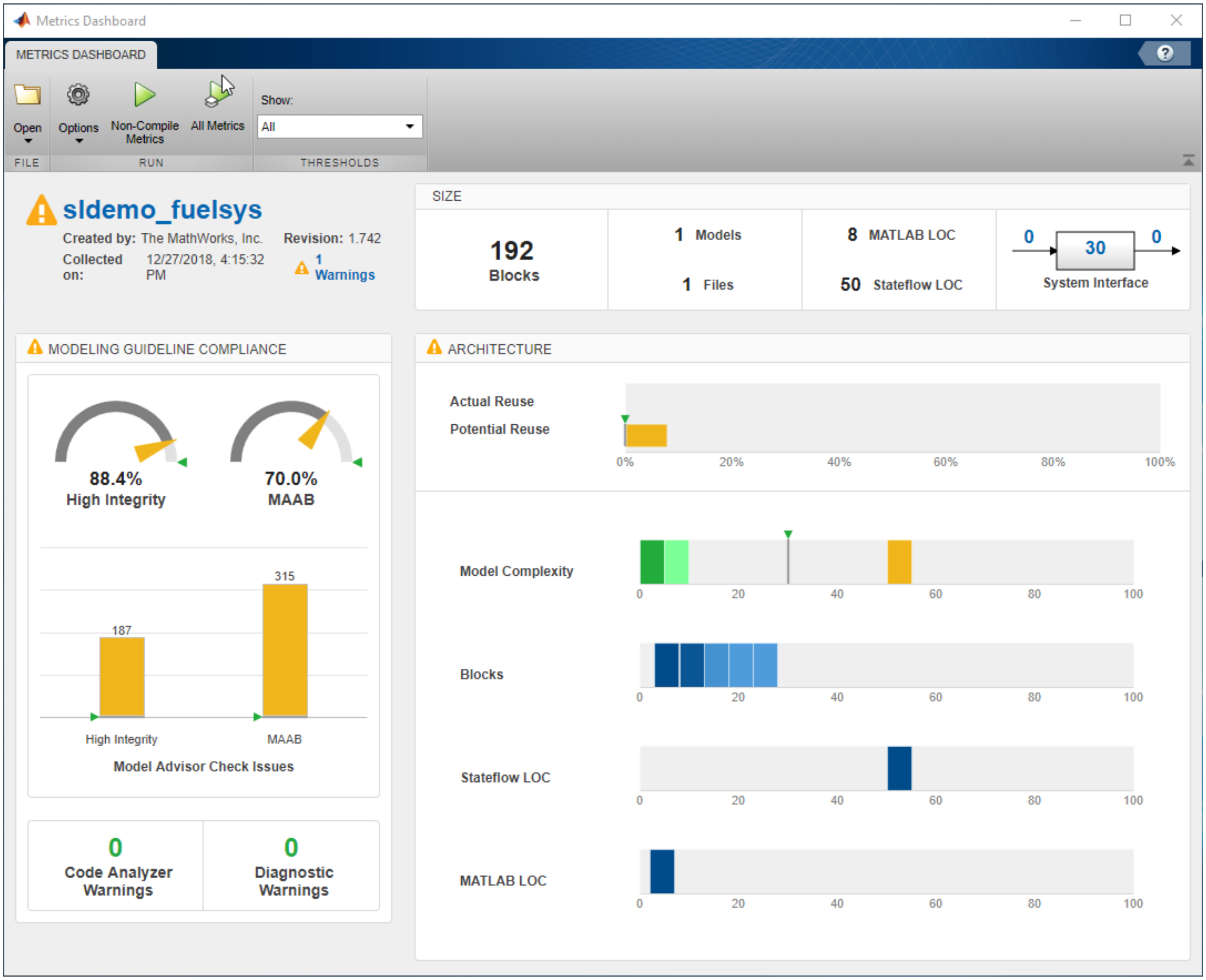
To collect model metric data and assess the design status and quality of your model, use the Metrics Dashboard. The Metrics Dashboard provides a view into the size, architecture, and guideline compliance for your model.

1. Open the model by typing `sldemo_fuelsys`.
2. In the model window, open the Metrics Dashboard by selecting **Analysis > Metrics Dashboard**.
3. To collect metric data for this model, click the **All Metrics** icon.



Analyze Metric Data

The Metrics Dashboard contains widgets that provide visualization of 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 (appears green in the widget) or produces a warning (appears yellow in the widget). Metrics that do not have threshold values appear blue in the widget. You can specify noncompliant ranges and apply other Metrics Dashboard customizations. For more information, see “Customize Metrics Dashboard Layout and Functionality” on page 5-49.



Show: All

Created by: The MathWorks, Inc.

Revision: 1.742

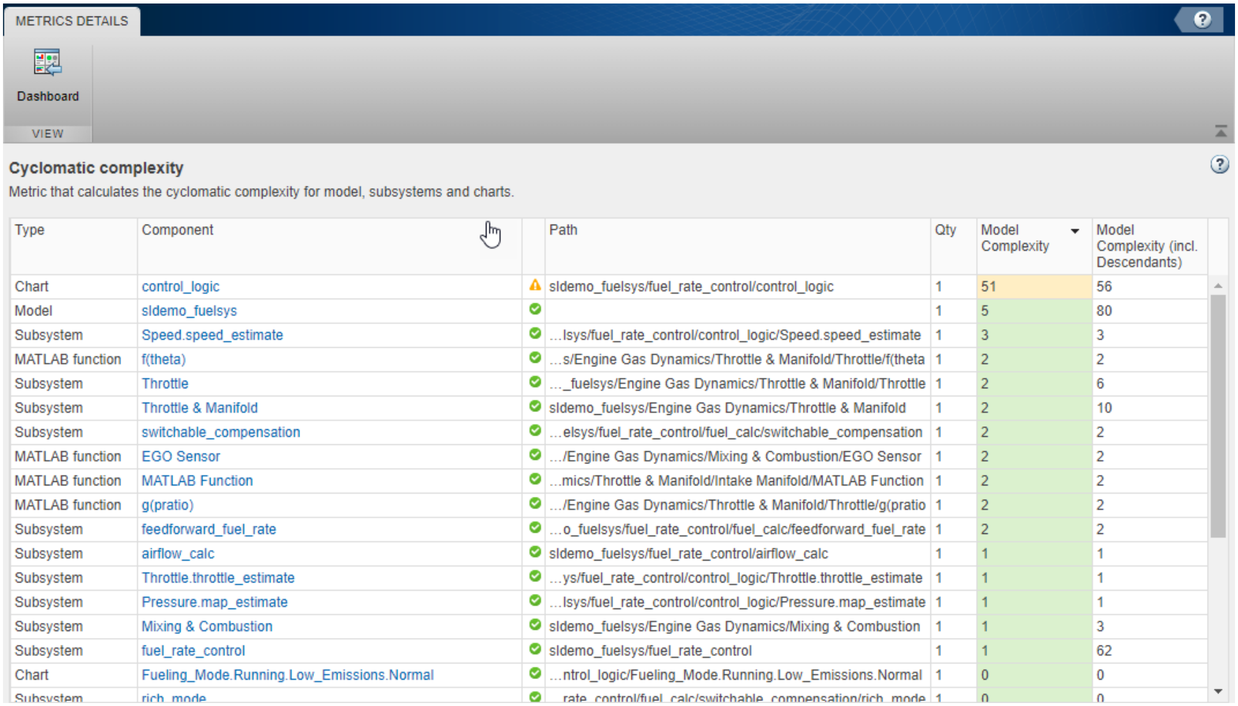
Collected on: 12/27/2018, 4:15:32 PM

In the `ARCHITECTURE` section of the dashboard, locate the **Model Complexity** widget. 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, which is the default threshold between compliant and warning.

Drill-In to 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 drill into model complexity details at the model, subsystem, and chart level, click anywhere 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.

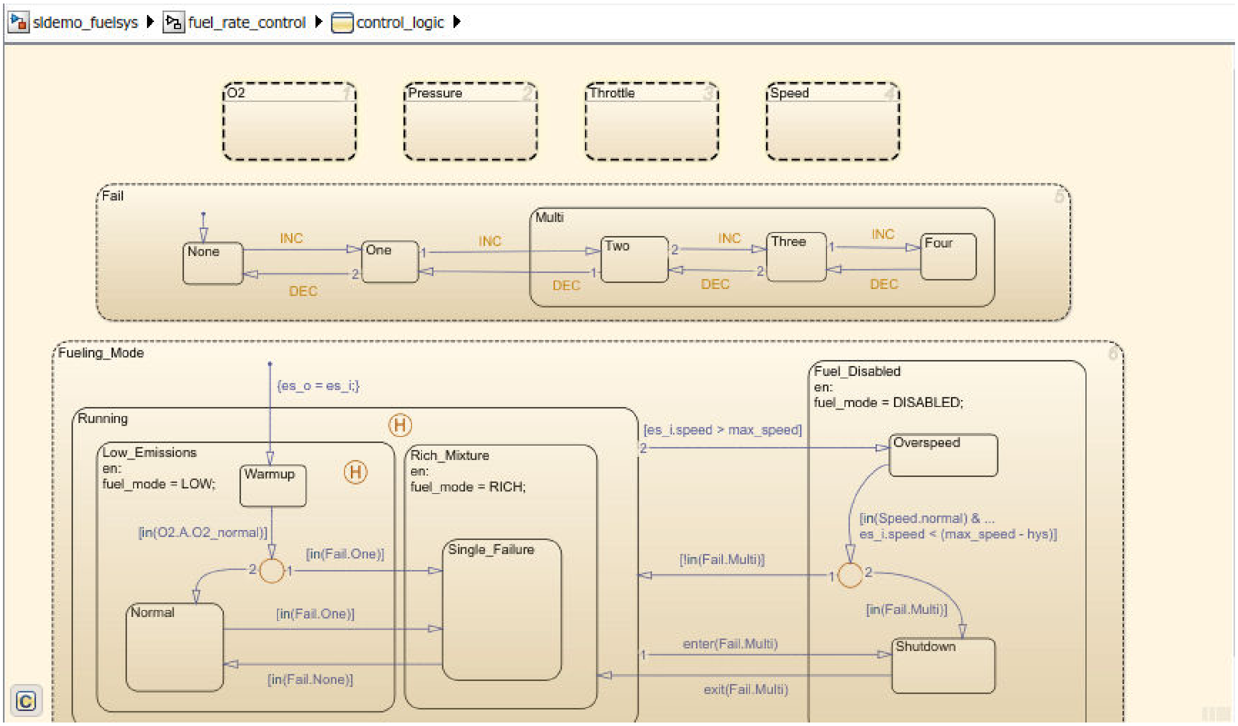


Cyclomatic complexity

Metric that calculates the cyclomatic complexity for model, subsystems and charts.

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 the **Modeling Guideline Compliance** widgets, to fix issues, open the Model Advisor. For the **Potential Reuse** widget, to create and link to library blocks, open the Clone Detection tool. Open the Model Advisor and the Clone Detection tool by clicking respective buttons on the drill-in details.

For this example, refactoring the `control_logic` chart by moving logic into atomic subcharts reduces the complexity for that component.

See Also

More About

- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2
- “Model Metrics”
- “Collect Model Metrics Programmatically” on page 5-20

Refactor Model with Clone Detection and Model Transformer Tools

With Simulink Check, you can use the Model Transformer and Identify Modeling Clones tools to refactor a model to improve model componentization and readability and enable reuse.

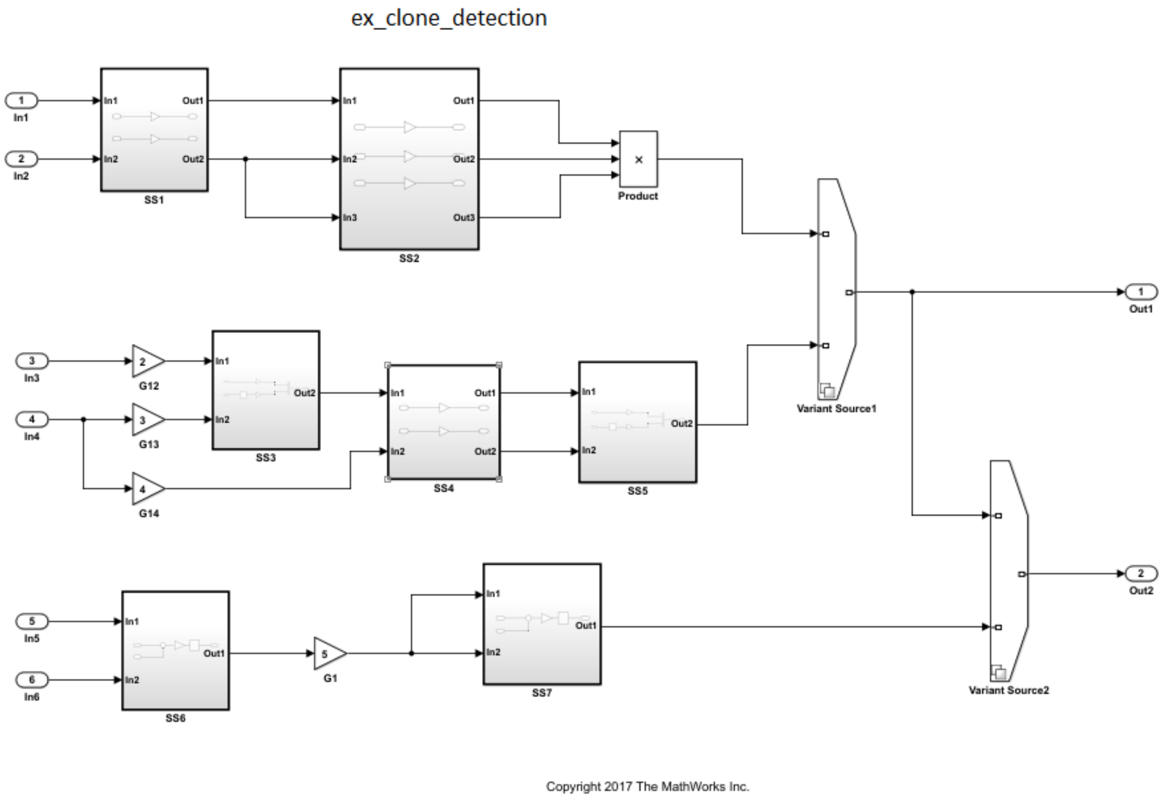
Identify and Replace Clones with Links to Library Blocks

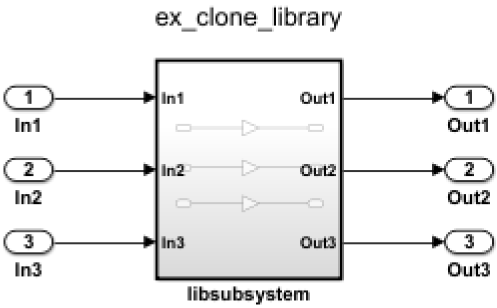
You can use the Identify Modeling Clones tool to enable component reuse by completing these tasks:

- Identify subsystem clones.
- Create library blocks from clones.
- Create a model that replaces clones with links to library blocks.
- Identify similar clones.

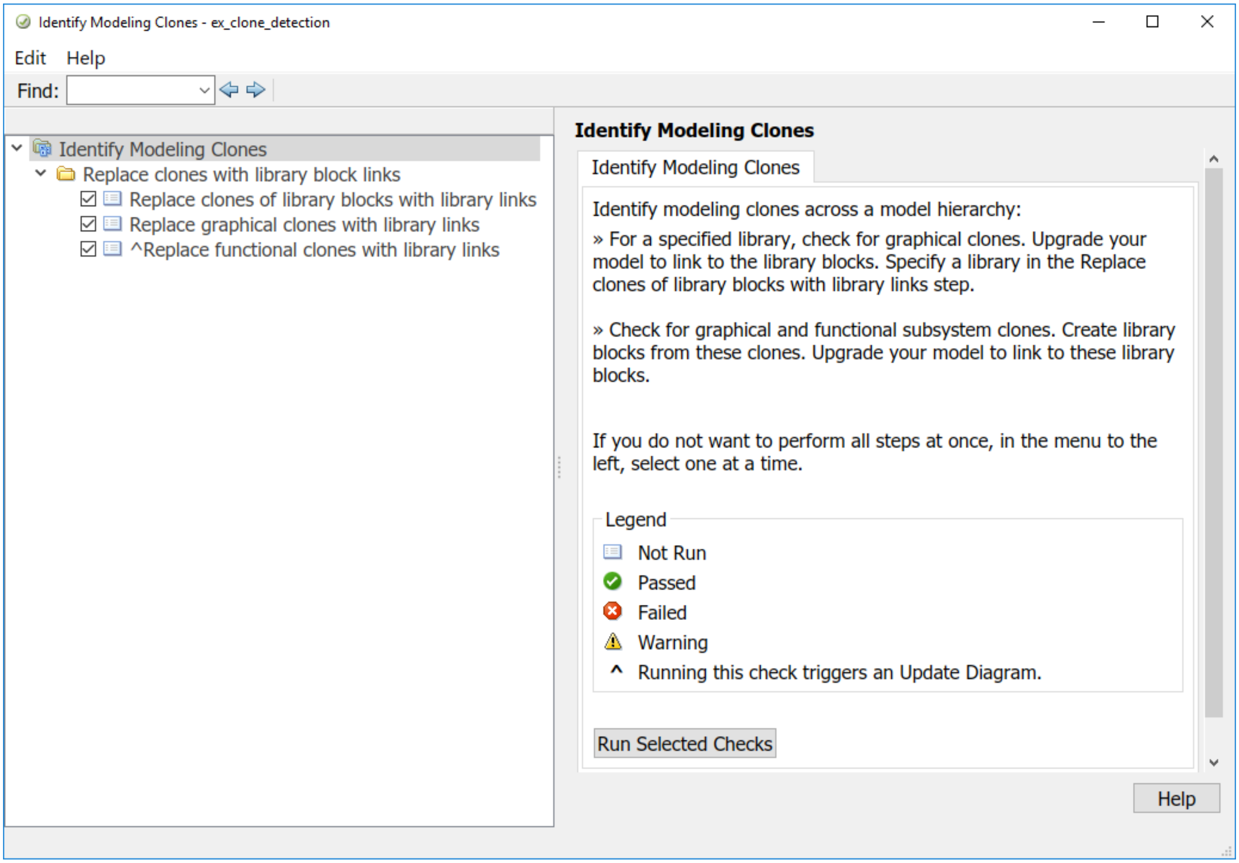
1. Open the example model `ex_clone_detection` and the corresponding library `ex_clone_library`.

   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   ex_clone_detection
   ex_clone_library
   ```
2. Save the model and library to the current folder on the MATLAB path.


ex_clone_detection
1
In1
2
In2
In1
Out1
In2
Out2
SS1
In1
Out1
In2
Out2
In3
Out3
SS2
×
Product
Variant Source1
1
Out1
3
In3
2
G12
4
In4
3
G13
4
G14
In1
In2
Out2
SS3
In1
Out1
In2
Out2
SS4
In1
In2
Out2
SS5
Variant Source2
2
Out2
5
In5
6
In6
In1
In2
Out1
SS6
5
G1
In1
In2
Out1
SS7
Copyright 2017 The MathWorks Inc.


ex_clone_library
1
In1
2
In2
3
In3
In1
In2
In3
Out1
Out2
Out3
libsubsystem
1
Out1
2
Out2
3
Out3

3 In the Simulink Editor, from the **Analysis** menu, select **Refactor Model** > **Identify Modeling Clones**. To open the Identify Modeling Clones tool programmatically, at the MATLAB command prompt type: `clonedetection('ex_clone_detection')`.


Identify Modeling Clones - ex_clone_detection
Edit
Help
Find:
Identify Modeling Clones
Replace clones with library block links
☑ Replace clones of library blocks with library links
☑ Replace graphical clones with library links
☑ ^Replace functional clones with library links
Identify Modeling Clones
Identify Modeling Clones
Run Selected Checks
Help

Identify modeling clones across a model hierarchy:

» For a specified library, check for graphical clones. Upgrade your model to link to the library blocks. Specify a library in the Replace clones of library blocks with library links step.

» Check for graphical and functional subsystem clones. Create library blocks from these clones. Upgrade your model to link to these library blocks.

If you do not want to perform all steps at once, in the menu to the left, select one at a time.

Legend

- ▤ Not Run
- ✓ Passed
- ✗ Failed
- ⚠ Warning
- ^ Running this check triggers an Update Diagram.

4 Open the **Identify Modeling Clones** folder.

5 Select **Replace clones of library blocks with library links**. In the **Library file name** field, insert the library name, `ex_clone_library`.

6 Select the **Identify Modeling Clones** folder. Then, click **Run Selected Checks**. Because every check is selected by default, the tool identifies all possible clones in the model.

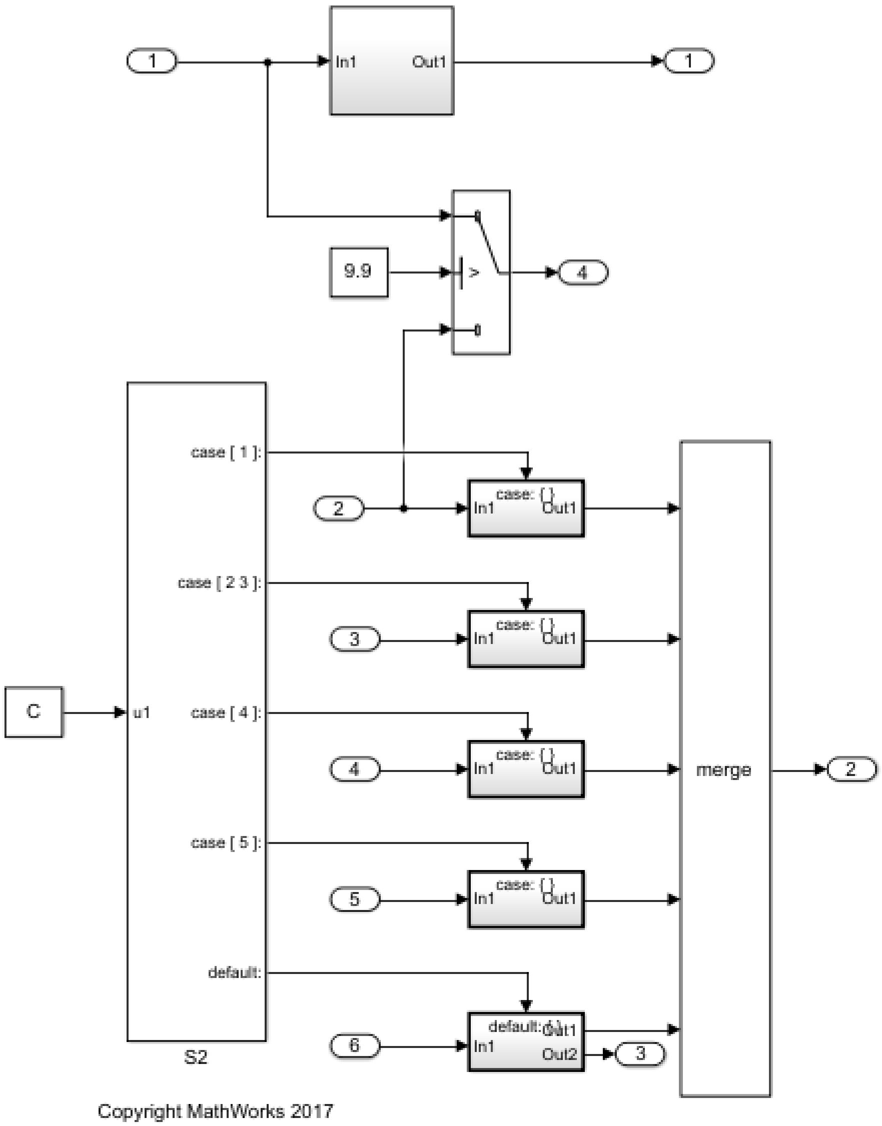
7 Select each check. The checks contain hyperlinks to the clones in the model.

Note Each check contains a **Refactor Model** button. To replace clones with links to library blocks, you must complete each check and click **Refactor Model**. You cannot simultaneously run selected checks and refactor the model.

Replace Qualifying Modeling Patterns with Variant Blocks

To improve model componentization by replacing qualifying modeling patterns with Variant Source and Variant Subsystem blocks, use the Model Transformer tool.

The `ex_variants_transformer` model contains several modeling patterns that qualify for transformation into variants blocks.



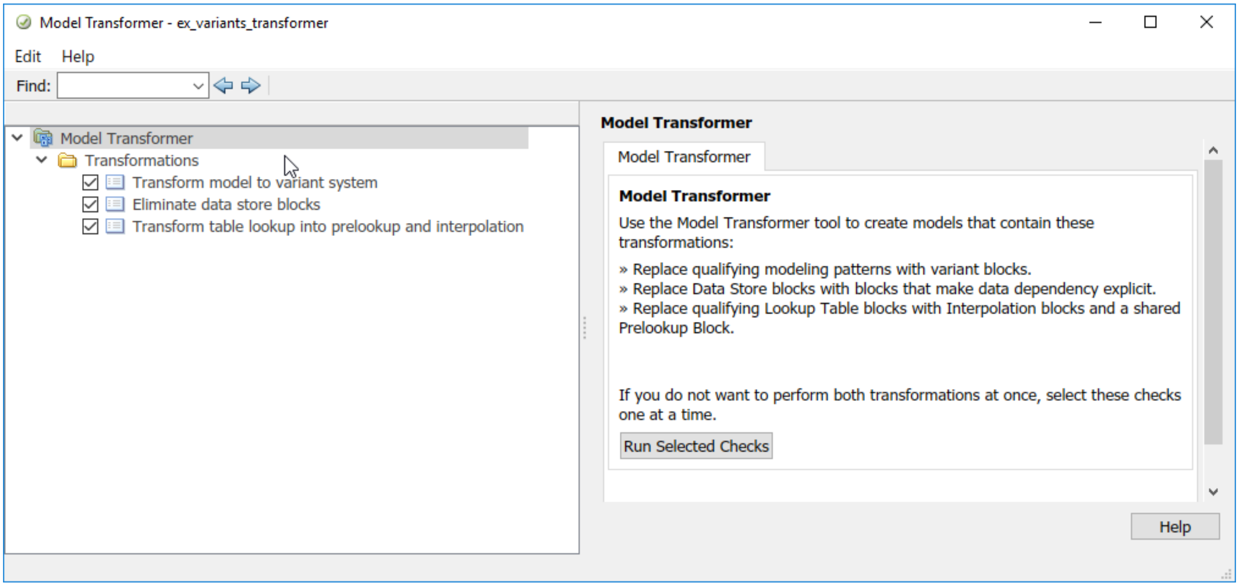
1 Open the example model `ex_variants_transformer` by entering these commands at the MATLAB command line:

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

2 Save the model to your working folder.

3 From the Simulink Editor, open the Model Transformer tool by selecting **Refactor Model** > **Model Transformer**. Or, in the Command Window, type:

```
mdltransformer(‘ex_variants_transformer’)
```


Model Transformer - ex_variants_transformer
Edit
Help
Find:
Model Transformer
Transformations
☑ Transform model to variant system
☑ Eliminate data store blocks
☑ Transform table lookup into prelookup and interpolation
Model Transformer
Model Transformer
Run Selected Checks
Help

Model Transformer

Use the Model Transformer tool to create models that contain these transformations:

» Replace qualifying modeling patterns with variant blocks.
» Replace Data Store blocks with blocks that make data dependency explicit.
» Replace qualifying Lookup Table blocks with Interpolation blocks and a shared Prelookup Block.

If you do not want to perform both transformations at once, select these checks one at a time.

4 Select the **Transform model to variant system** check.

5 Click **Run This Check**. The top **Result** table contains a list of system constants that qualify to be part of condition expressions in Variant Source or Variant Subsystem blocks.

6 Click **Refactor Model**.

7 Your working folder contains a folder called `m2m_ex_variants_transformer`. This folder contains the transformed model `gen0_ex_variants_transformer`.

8 The bottom **Results** table contains hyperlinks to the original and transformed models.

9 Select the **Eliminate data store blocks** check. You can use this check to replace data stores with blocks that improve model readability by making data dependency

explicit. For an example, see “Improve Model Readability by Eliminating Local Data Store Blocks” on page 3-22.

10 Select the **Transform table lookup into prelookup and interpolation** check. You can use this check to replace Lookup Table blocks into a single shared Prelookup block and multiple Interpolation blocks. For an example, see “Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks” on page 3-40.

See Also

More About

- “Refactor Models”
- “Enable Component Reuse by Using Clone Detection” on page 3-15
- “Transform Model to Variant System” on page 3-9

Create a Simplified, Standalone Model Using the Model Slicer Tool

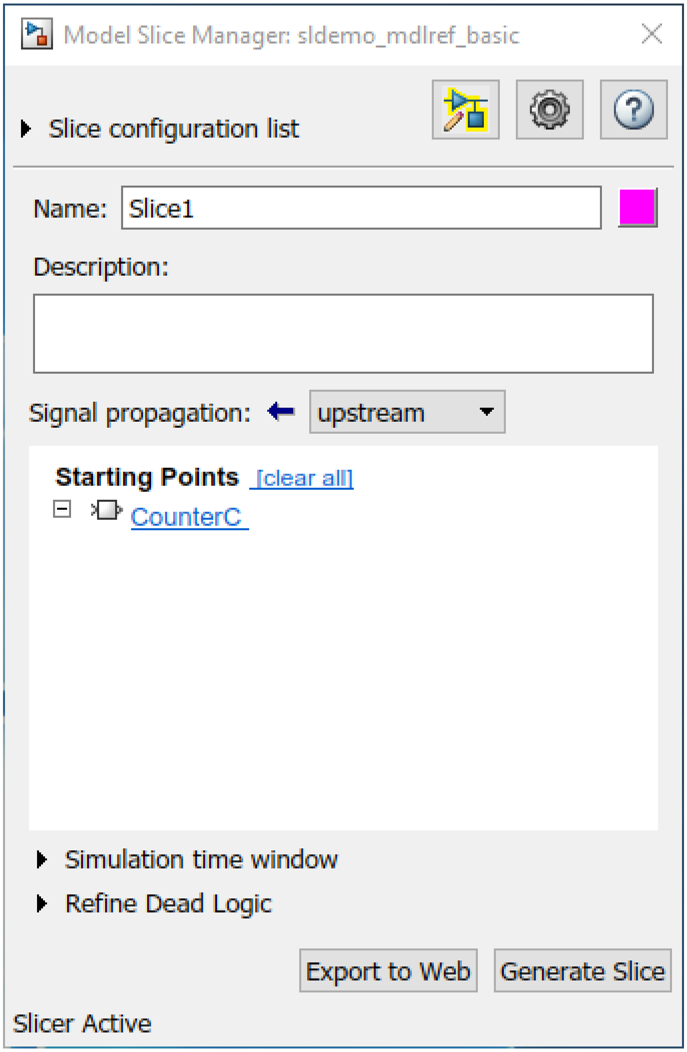
You can simplify simulation, debugging, and formal analysis of large, complex models by focusing on areas of interest in your model. After highlighting a portion of your model using the Model Slicer, you can generate a simplified standalone model. The simplified model contains the blocks and dependency 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:

   ```
   sldemo_mdlref_basic
   ```
2. Select **Analysis** > **Model Slicer** to open the Model Slice Manager.
3. In the Model Slice Manager, click the arrow to expand the **Slice 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**.



Model Slice Manager: sldemo_mdlref_basic

Name: Slice1

Description:

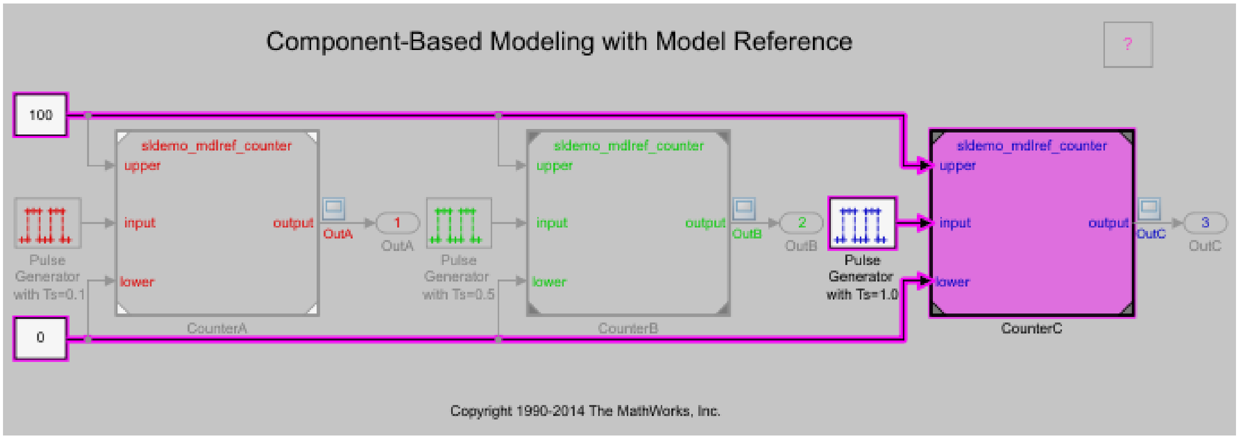
Signal propagation: upstream

Starting Points [clear all]

CounterC

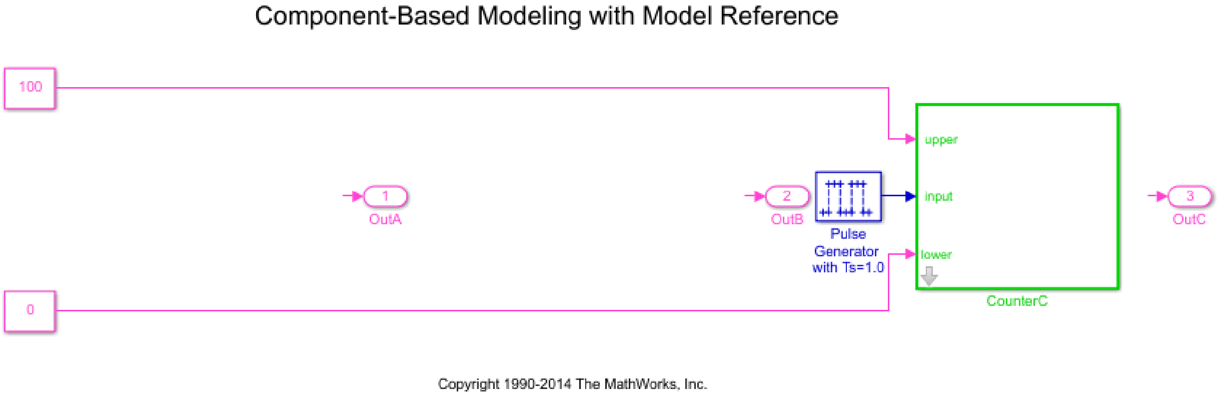
Slicer Active

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.


Component-Based Modeling with Model Reference
100
0
1
OutA
2
OutB
Pulse Generator with Ts=1.0
upper
input
lower
CounterC
3
OutC
Copyright 1990-2014 The MathWorks, Inc.

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

See Also

More About

- “Model Slicer Considerations and Limitations” on page 11-53
- “Highlight Functional Dependencies” on page 11-2
- “Refine Highlighted Model” on page 11-13

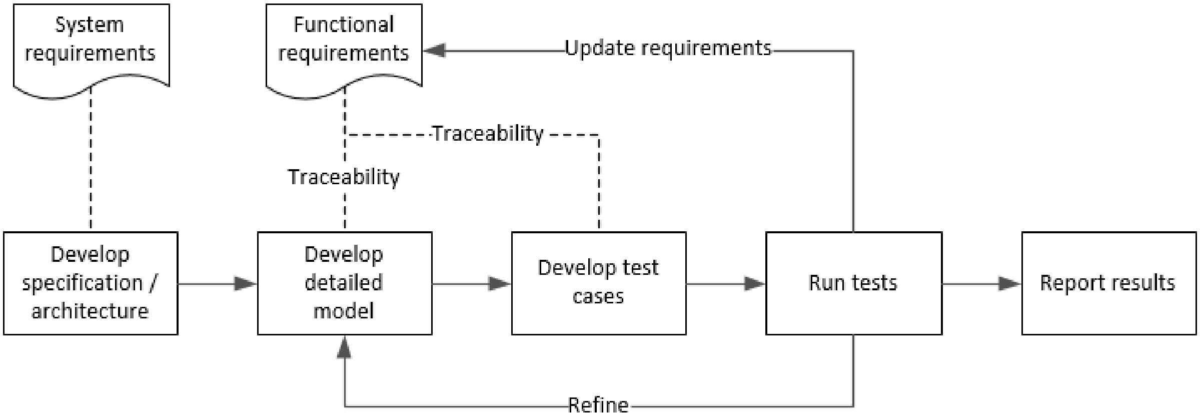
Verification and Validation

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:

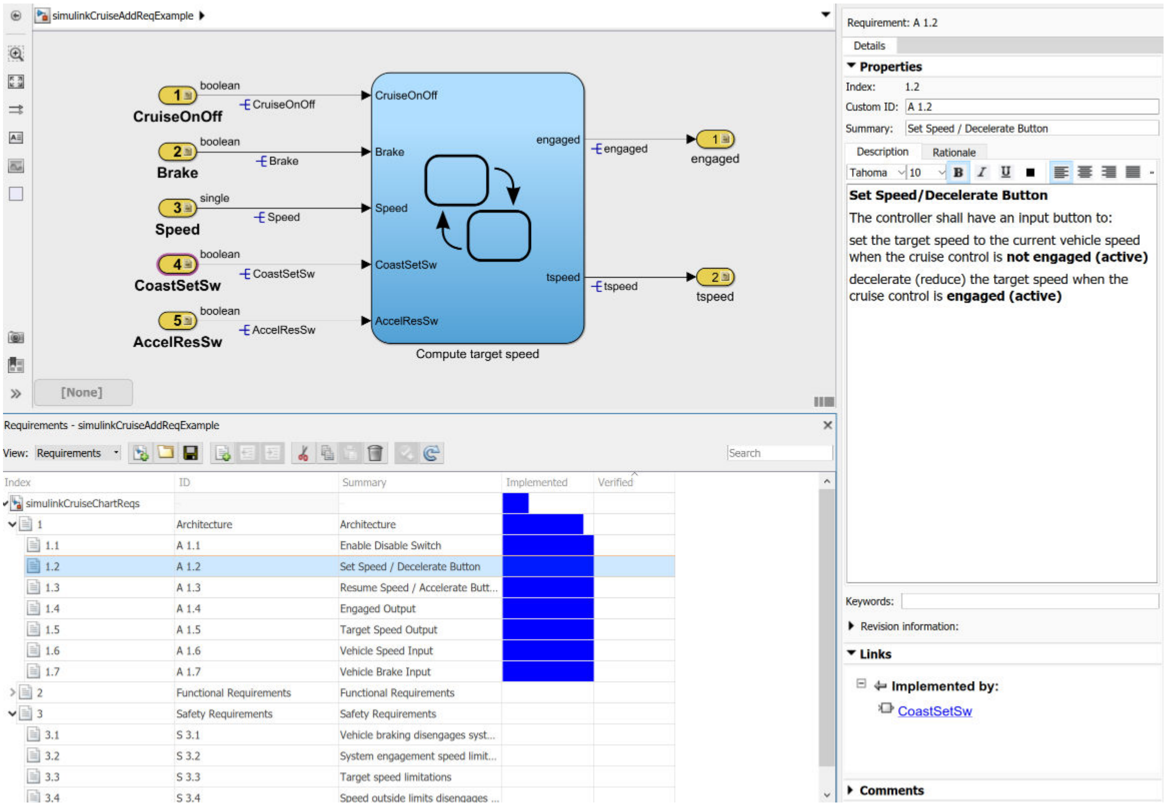
- That the cruise control system transitions to disengaged from engaged when a braking event has occurred
- That the cruise control system transitions to disengaged from engaged when the current vehicle speed is outside the range of 20 mph to 90 mph.

Display the Requirements and Test Case

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**.



Requirements - simulinkCruiseAddReqExample

View: Requirements

Index	ID	Summary	Implemented	Verified
simulinkCruiseChartReqs			■	
1	Architecture	Architecture	■	
1.1	A 1.1	Enable Disable Switch	■	
1.2	A 1.2	Set Speed / Decelerate Button	■	
1.3	A 1.3	Resume Speed / Accelerate Butt...	■	
1.4	A 1.4	Engaged Output	■	
1.5	A 1.5	Target Speed Output	■	
1.6	A 1.6	Vehicle Speed Input	■	
1.7	A 1.7	Vehicle Brake Input	■	
2	Functional Requirements	Functional Requirements		
3	Safety Requirements	Safety Requirements		
3.1	S 3.1	Vehicle braking disengages syst...		
3.2	S 3.2	System engagement speed limit...		
3.3	S 3.3	Target speed limitations		
3.4	S 3.4	Speed outside limits disengages ...		

Requirement: A 1.2

Details

Properties

Index: 1.2

Custom ID: A 1.2

Summary: Set Speed / Decelerate Button

Description | Rationale

Set Speed/Decelerate Button

The controller shall have an input button to:

set the target speed to the current vehicle speed when the cruise control is **not engaged (active)**

decelerate (reduce) the target speed when the cruise control is **engaged (active)**

Keywords:

Revision information:

Links

Implemented by:

CoastSetSw

Comments

5 Open the Simulink Test file `slReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

Link Requirements to Tests

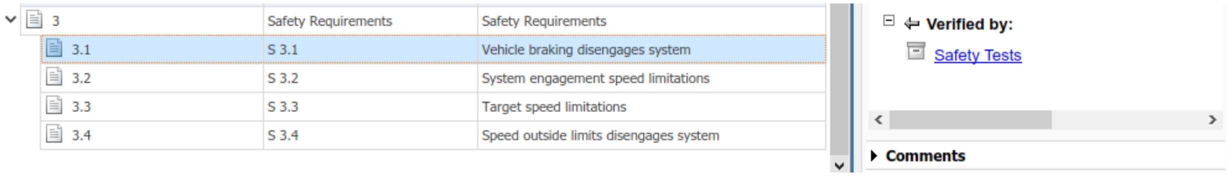
Link the requirements to the test case.

1 In the Requirements Browser, select requirement `S 3.1`.

2 In the Test Manager, expand the test file and select the **Safety Tests** test case. Expand the **Requirements** section.

3 In the **Requirements** section, select **Add > Link to Selected Requirement**.

The requirements browser displays the verification-type link.



3	Safety Requirements	Safety Requirements
3.1	S 3.1	Vehicle braking disengages system
3.2	S 3.2	System engagement speed limitations
3.3	S 3.3	Target speed limitations
3.4	S 3.4	Speed outside limits disengages system

Verified by:

Safety Tests

Comments

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

Run the Test

1 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

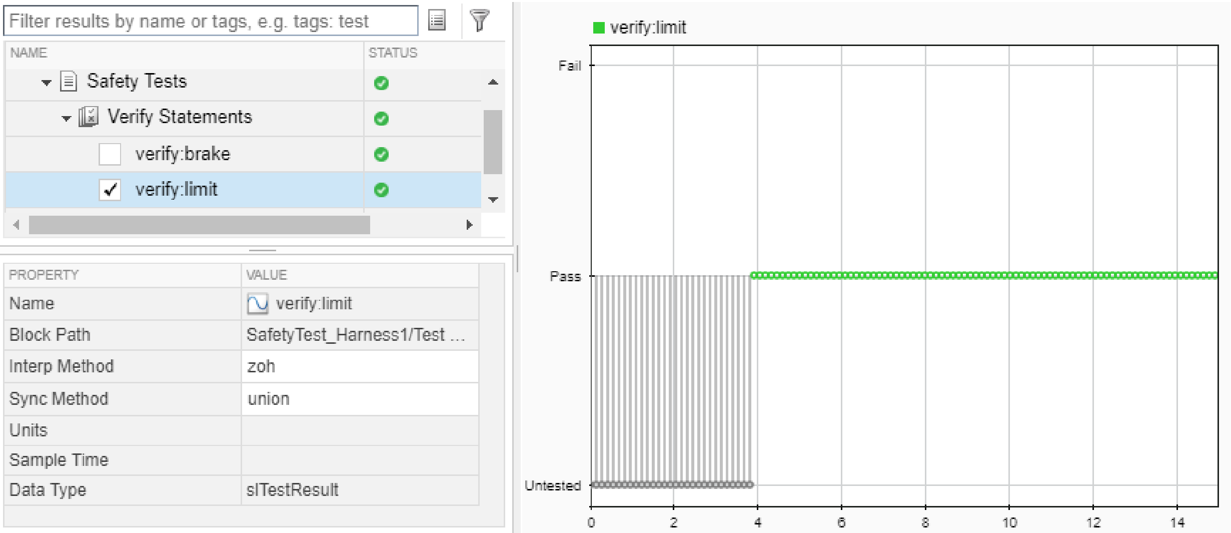
  ```
  verify(engaged == false,...
      'verify:brake',...
      'system must disengage when brake applied')
  ```

- 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')
  ```

2 Run the test case. In the Test Manager toolstrip, click **Run**.

3 When the test finishes, expand the **Verify Statements** results. The Test Manager results show that both assessments pass, and the plot shows the detailed results of each `verify` statement.

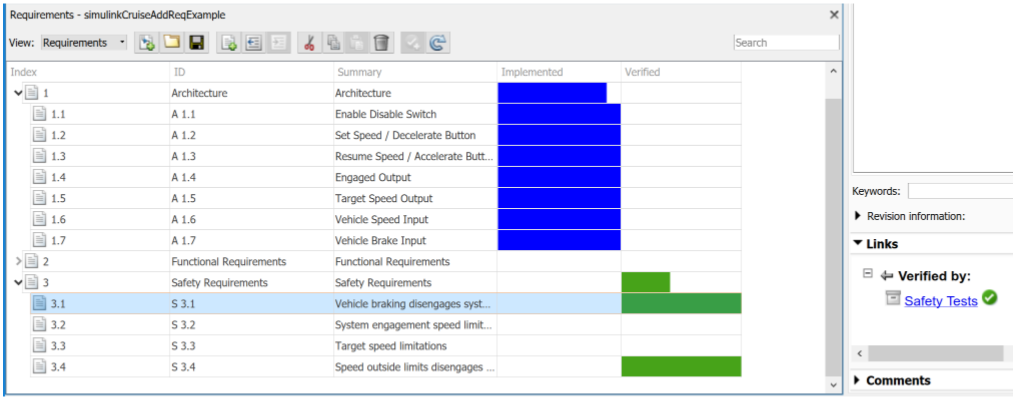


Filter results by name or tags, e.g. tags: test

NAME	STATUS
Safety Tests	✓
Verify Statements	✓
□ verify:brake	✓
☑ verify:limit	✓

PROPERTY	VALUE
Name	verify:limit
Block Path	SafetyTest_Harness1/Test ...
Interp Method	zoh
Sync Method	union
Units	
Sample Time	
Data Type	slTestResult

4 In the Requirements Browser, right-click a requirement and select **Refresh Verification Status** to show the passing test results for each requirement.



Requirements - simulinkCruiseAddReqExample

View: Requirements

Search

Index	ID	Summary	Implemented	Verified
1	Architecture	Architecture	🟦	
1.1	A 1.1	Enable Disable Switch	🟦	
1.2	A 1.2	Set Speed / Decelerate Button	🟦	
1.3	A 1.3	Resume Speed / Accelerate Butt...	🟦	
1.4	A 1.4	Engaged Output	🟦	
1.5	A 1.5	Target Speed Output	🟦	
1.6	A 1.6	Vehicle Speed Input	🟦	
1.7	A 1.7	Vehicle Brake Input	🟦	
2	Functional Requirements	Functional Requirements		
3	Safety Requirements	Safety Requirements		🟩
3.1	S 3.1	Vehicle braking disengages syst...		🟩
3.2	S 3.2	System engagement speed limit...		
3.3	S 3.3	Target speed limitations		
3.4	S 3.4	Speed outside limits disengages ...		🟩

Keywords:

Revision information:

Links

Verified by:

Safety Tests ✓

Comments

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 For the **Template File**, select the `ReportTemplate.dotx` file in the **documents** project folder.

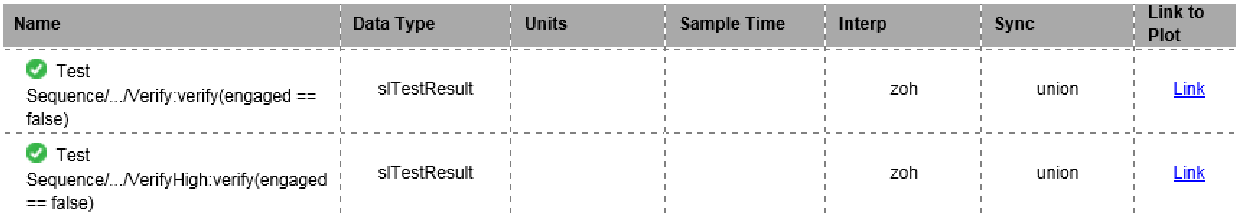
d Enter a file name and select a location for the report.

e Click **Create**.

2 Review the report.

a In the **Test Case Requirements** section, click the link to trace to the requirements document.

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



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

See Also

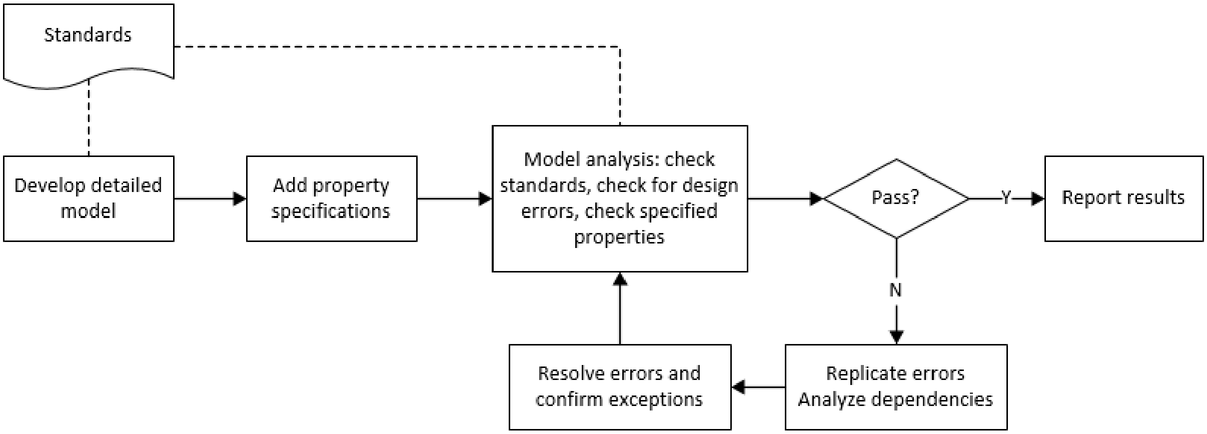
Related Examples

- “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 MAAB style guidelines and high-integrity system design guidelines such as DO-178 and ISO 26262. Analyze your model for errors, dead logic, and conditions that violate required properties. Using the analysis results, update your model and document exceptions. Report the results using customizable templates.


Standards
Develop detailed model
Add property specifications
Model analysis: check standards, check for design errors, check specified properties
Pass?
Y
Report results
N
Replicate errors Analyze dependencies
Resolve errors and confirm exceptions

Check Model for Style Guideline Violations and Design Errors

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

Check Model for MAAB Style Guideline Violations

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

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

   ```
   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 model window, select **Analysis** > **Model Advisor** > **Model Advisor**.
4. Click OK to choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
5. Check your model for MAAB style guideline violations using Simulink Check.

   a. In the left pane, in the **By Product > Simulink Check > Modeling Standards > MathWorks Automotive Advisory Board Checks** folder, select:

      - **Check for indexing in blocks**
      - **Check for prohibited blocks in discrete controllers**
      - **Check model diagnostic parameters**

   b. Right-click the **MathWorks Automotive Advisory Board Checks** node, and then select `Run Selected Checks`.

   c. Click **Check model diagnostic parameters** to review the configuration parameter settings that violate MAAB style guidelines.

   d. In the right pane, click the parameter links to update the values in the Configuration Parameters dialog box.

   e. To verify that your model passes, rerun the check. Repeat steps `c` and `d`, if necessary, to reach compliance.

   f. To generate a results report of the Simulink Check checks, select the **MathWorks Automotive Advisory Board Checks** node, and then, in the right pane click **Generate Report...**.

Check Model for Design Errors

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

1. In the left pane, in the **By Product > Simulink Design Verifier** folder, select **Design Error Detection**.
2. In the right pane, click **Run Selected Checks**.
3. After the analysis is complete, expand the **Design Error Detection** folder, then select checks to review warnings or errors.
4. In the right pane, click **Simulink Design Verifier Results Summary**. The dialog box provides tools to help you diagnose errors and warnings in your model.

   a. Review the results on the model. Click **Highlight analysis results on model**. Click the `Compute target speed` subsystem, outlined in red. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.

   b. Review the harness model. The Simulink Design Verifier Results Inspector window displays information that an overflow error occurred. To see the test cases that demonstrate the errors, click **View test case**.

   c. Review the analysis report. In the Simulink Design Verifier Results Inspector window, click **Back to summary**. To see a detailed analysis report, click **HTML** or **PDF**.

See Also

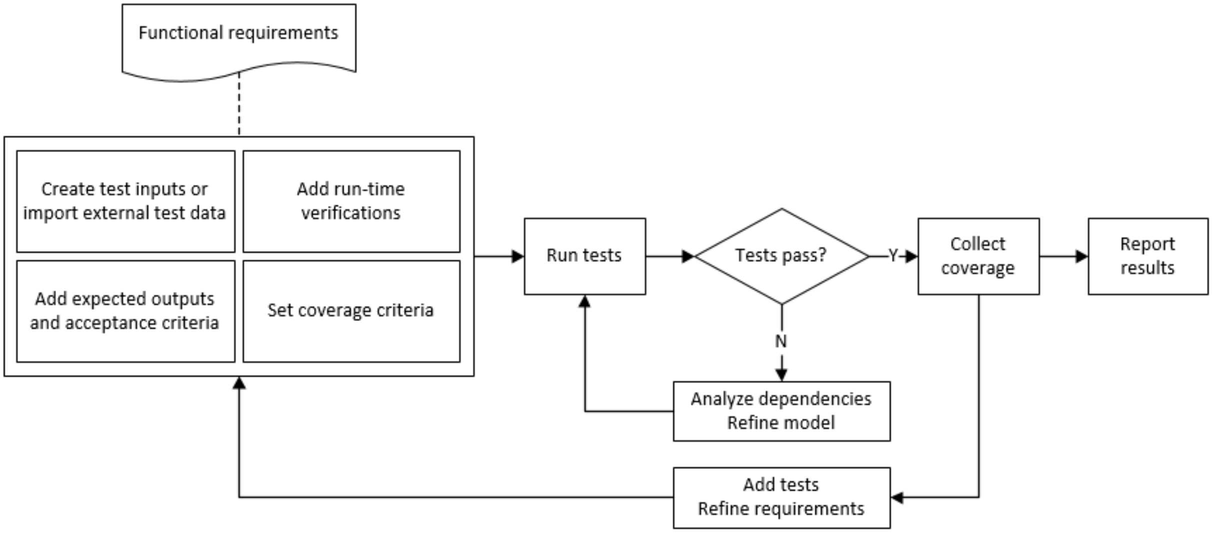
Related Examples

- “Check Model Compliance by Using the Model Advisor” on page 3-2
- “Collect Model Metrics Using the Model Advisor” on page 5-10
- “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). 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
   - That the system disengages when the speed exceeds a limit

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

Measure Model Coverage

1. In the Test Manager, enable coverage collection for the test case.

   a. Open the Test Manager. In the Simulink menu, click **Analysis > Test Manager**.

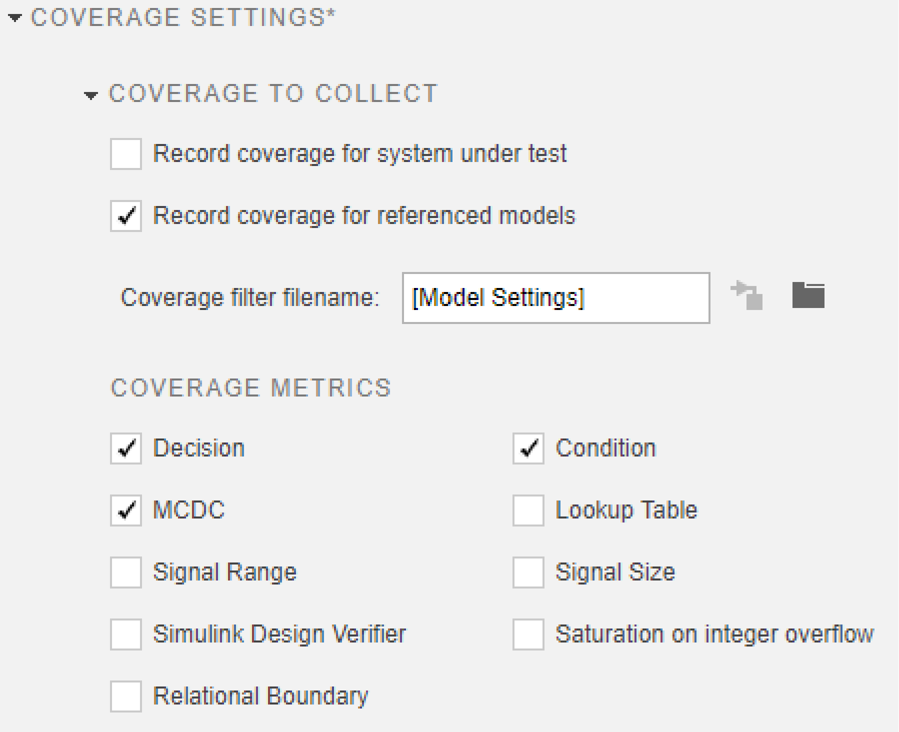
   b. In the **Test Browser**, click the `slReqTests` test file.

   c. Expand **Coverage Settings**.

   d. Under **Coverage to Collect**, select **Record coverage for referenced models**.

      You specify a coverage filter to use for coverage analysis by using the **Coverage filter filename** field. The default setting honors the model configuration parameter settings. Leaving the **Coverage filter filename** field empty attaches no coverage filter.

   e. Under **Coverage Metrics**, select **Decision**, **Condition**, and **MCDC**.



COVERAGE SETTINGS*

COVERAGE TO COLLECT

☐ Record coverage for system under test

☑ Record coverage for referenced models

Coverage filter filename: [Model Settings]

COVERAGE METRICS

☑ Decision ☑ Condition

☑ MCDC ☐ Lookup Table

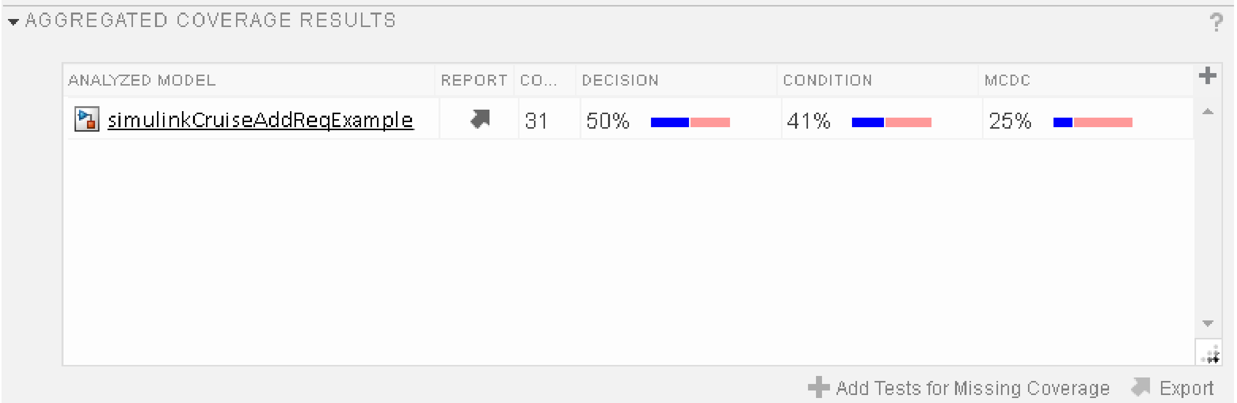
☐ Signal Range ☐ Signal Size

☐ Simulink Design Verifier ☐ Saturation on integer overflow

☐ Relational Boundary

2 Run the test. On the Test Manager toolstrip, click **Run**.

3 When the test finishes, in the Test Manager, navigate to the test case. The aggregated coverage results show that the example model achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.



ANALYZED MODEL	REPORT	CO...	DECISION	CONDITION	MCDC
simulinkCruiseAddReqExample		31	50%	41%	25%

Generate Tests to Increase Model Coverage

1. Use Simulink Design Verifier to generate additional tests to increase model coverage. Select the test case in the **Results and Artifacts** and open the aggregated coverage results section.
2. Select the test results from the previous section and then click **Add Tests for Missing Coverage**.

   The **Add Tests for Missing Coverage** options open.
3. Under **Harness**, choose `Create a new harness`.
4. Click **OK** to add tests to the test suite using Simulink Design Verifier.
5. Run the updated test suite. On the Test Manager toolstrip, click **Run**. The test results include coverage for the combined test case inputs, achieving increased model coverage.

See Also

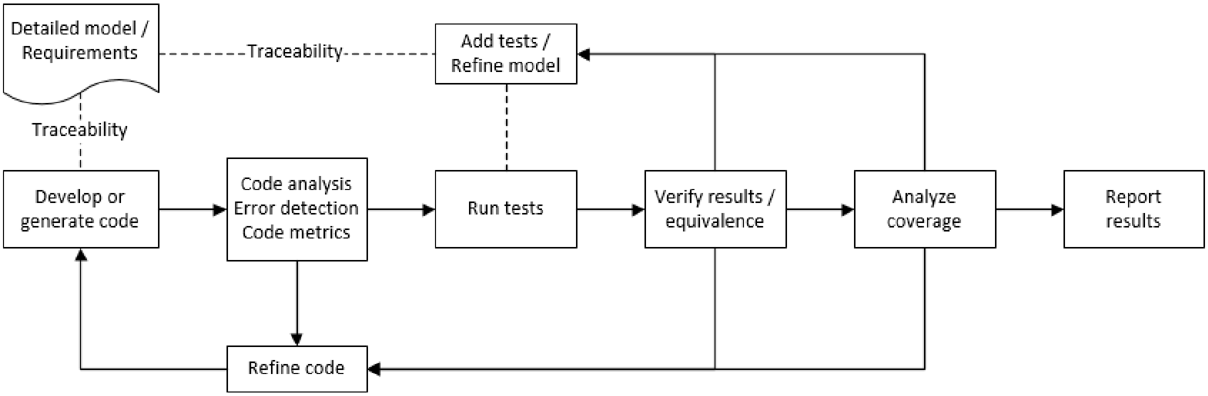
Related Examples

- “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.



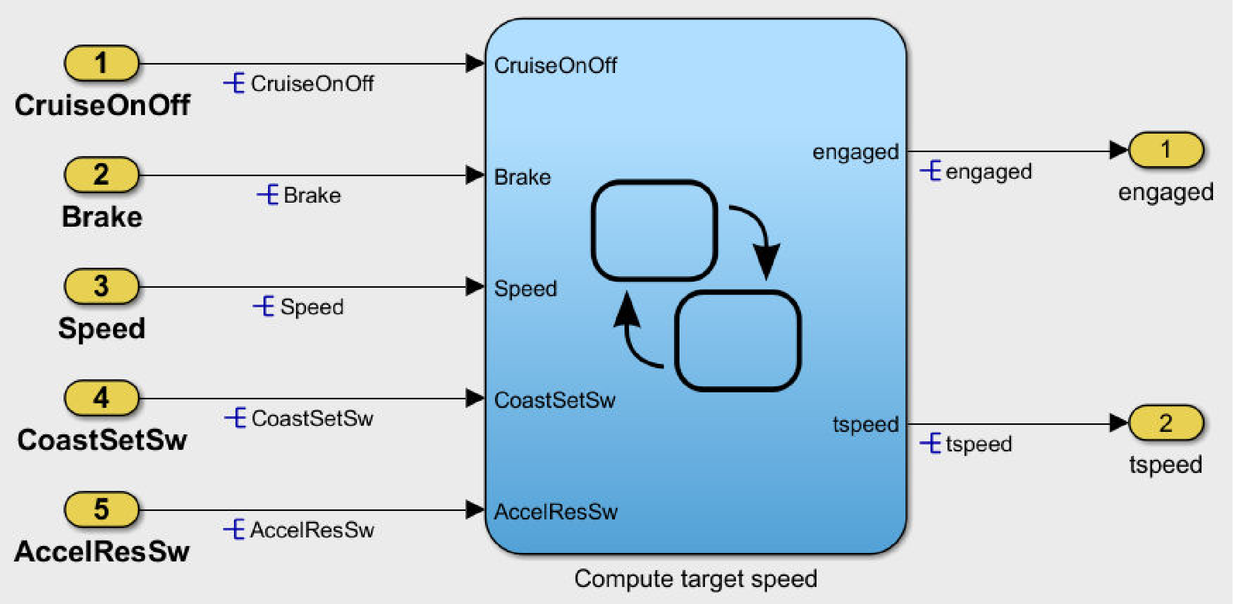
Analyze Code for Defects, Metrics, and MISRA C:2012

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`.

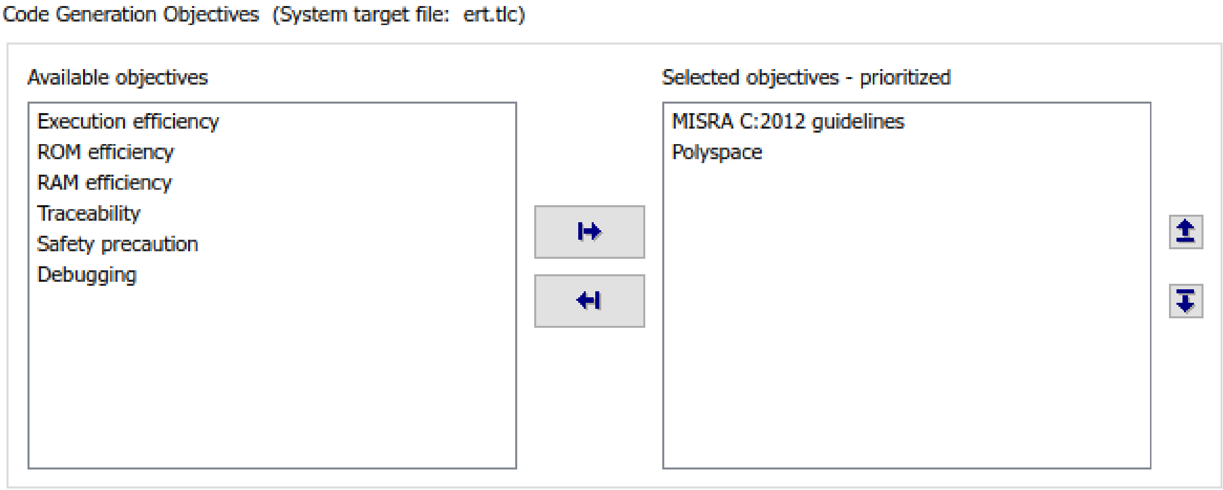


Run Code Generator Checks

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 Generation Advisor**.

2 Select the Code Generation Advisor folder. Add the `Polyspace` objective. The `MISRA C:2012 guidelines` objective is already selected.



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

Available objectives

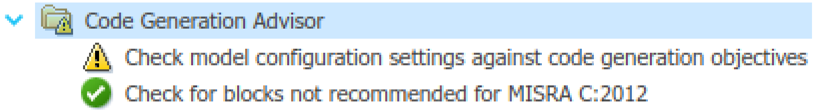
- Execution efficiency
- ROM efficiency
- RAM efficiency
- Traceability
- Safety precaution
- Debugging

Selected objectives - prioritized

- MISRA C:2012 guidelines
- Polyspace

3 Click **Run Selected Checks**.

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



Code Generation Advisor

- ⚠ Check model configuration settings against code generation objectives
- ✓ Check for blocks not recommended for MISRA C:2012

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

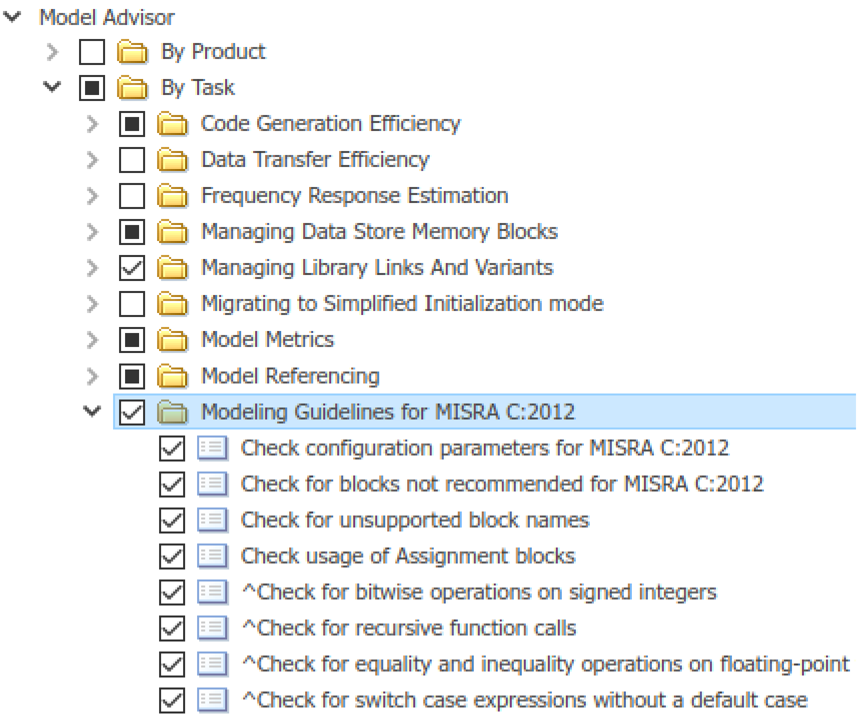
5 Rerun the check by selecting **Run This Check**.

Run Model Advisor Checks

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

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

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



- Model Advisor
  - □ By Product
  - ▣ By Task
    - ▣ Code Generation Efficiency
    - □ Data Transfer Efficiency
    - □ Frequency Response Estimation
    - ▣ Managing Data Store Memory Blocks
    - ☑ Managing Library Links And Variants
    - □ Migrating to Simplified Initialization mode
    - ▣ Model Metrics
    - ▣ Model Referencing
    - ☑ Modeling Guidelines for MISRA C:2012
      - ☑ Check configuration parameters for MISRA C:2012
      - ☑ Check for blocks not recommended for MISRA C:2012
      - ☑ Check for unsupported block names
      - ☑ Check usage of Assignment blocks
      - ☑ ^Check for bitwise operations on signed integers
      - ☑ ^Check for recursive function calls
      - ☑ ^Check for equality and inequality operations on floating-point
      - ☑ ^Check for switch case expressions without a default case

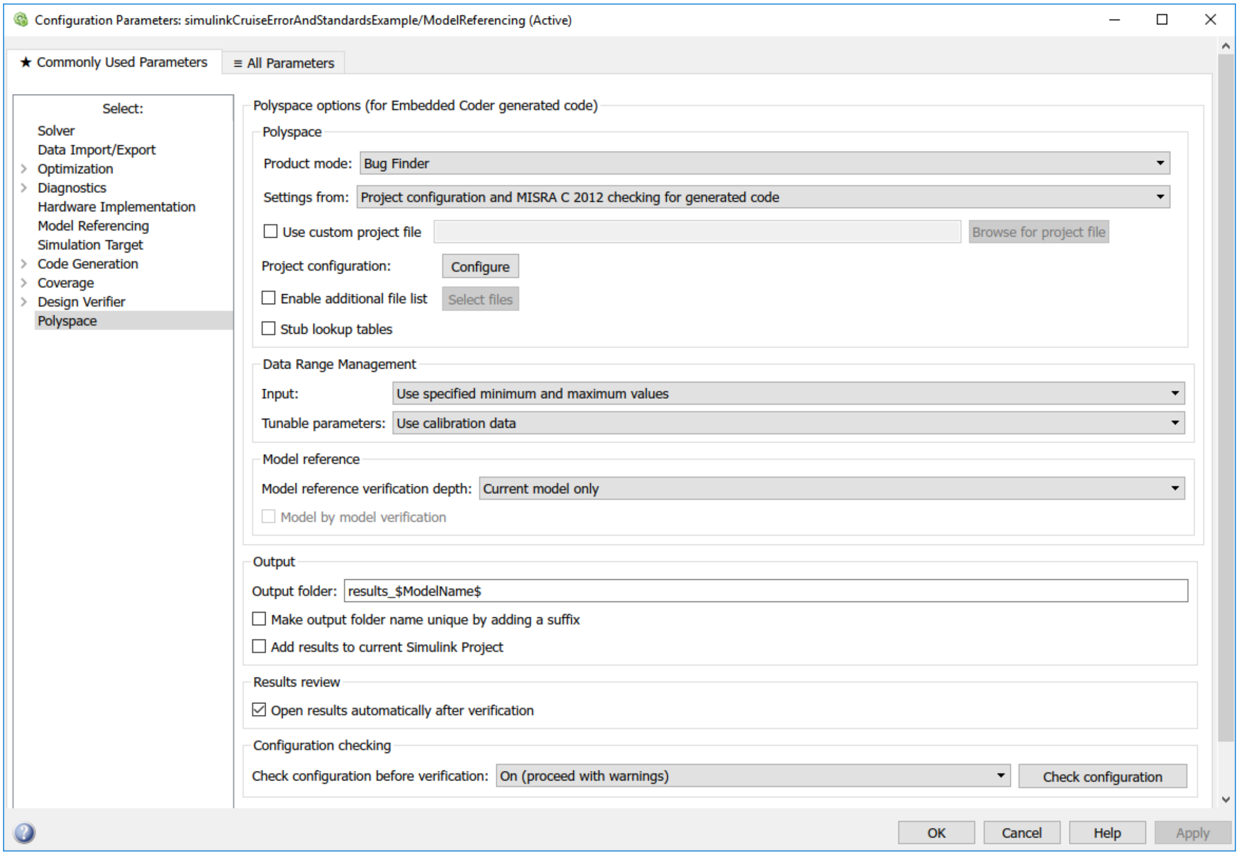
3. Click **Run Selected Checks** and review the results.
4. If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

Generate and Analyze Code

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

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

3 After the code is generated, right-click Compute target speed and select **Polyspace** > **Options**.



Configuration Parameters: simulinkCruiseErrorAndStandardsExample/ModelReferencing (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration and MISRA C 2012 checking for generated code

□ Use custom project file

Project configuration: Configure

□ Enable additional file list

□ Stub lookup tables

Data Range Management

Input: Use specified minimum and maximum values

Tunable parameters: Use calibration data

Model reference

Model reference verification depth: Current model only

□ Model by model verification

Output

Output folder: results_$ModelName$

□ Make output folder name unique by adding a suffix

□ Add results to current Simulink Project

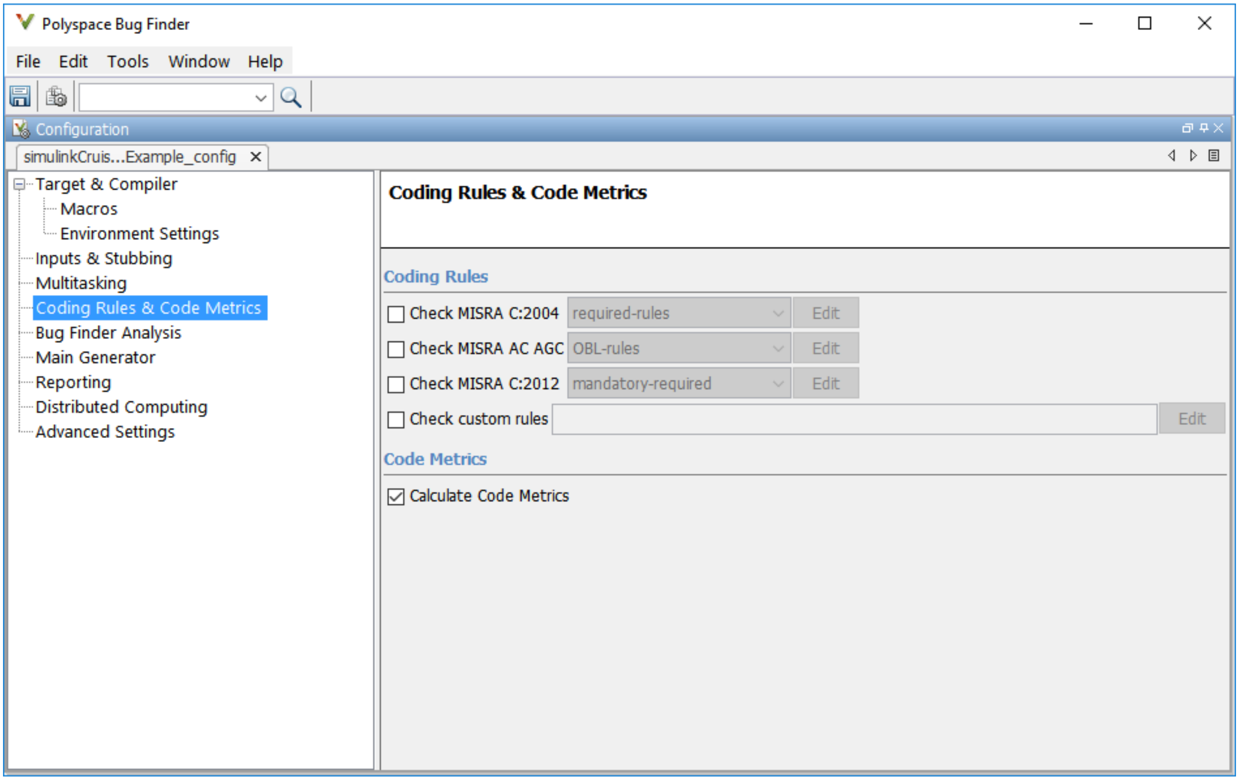
Results review

☑ Open results automatically after verification

Configuration checking

Check configuration before verification: On (proceed with warnings)

4 Click the **Configure** (Polyspace Bug Finder) button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.



Coding Rules & Code Metrics

Coding Rules

□ Check MISRA C:2004 required-rules

□ Check MISRA AC AGC OBL-rules

□ Check MISRA C:2012 mandatory-required

□ Check custom rules

Code Metrics

☑ Calculate Code Metrics

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**.

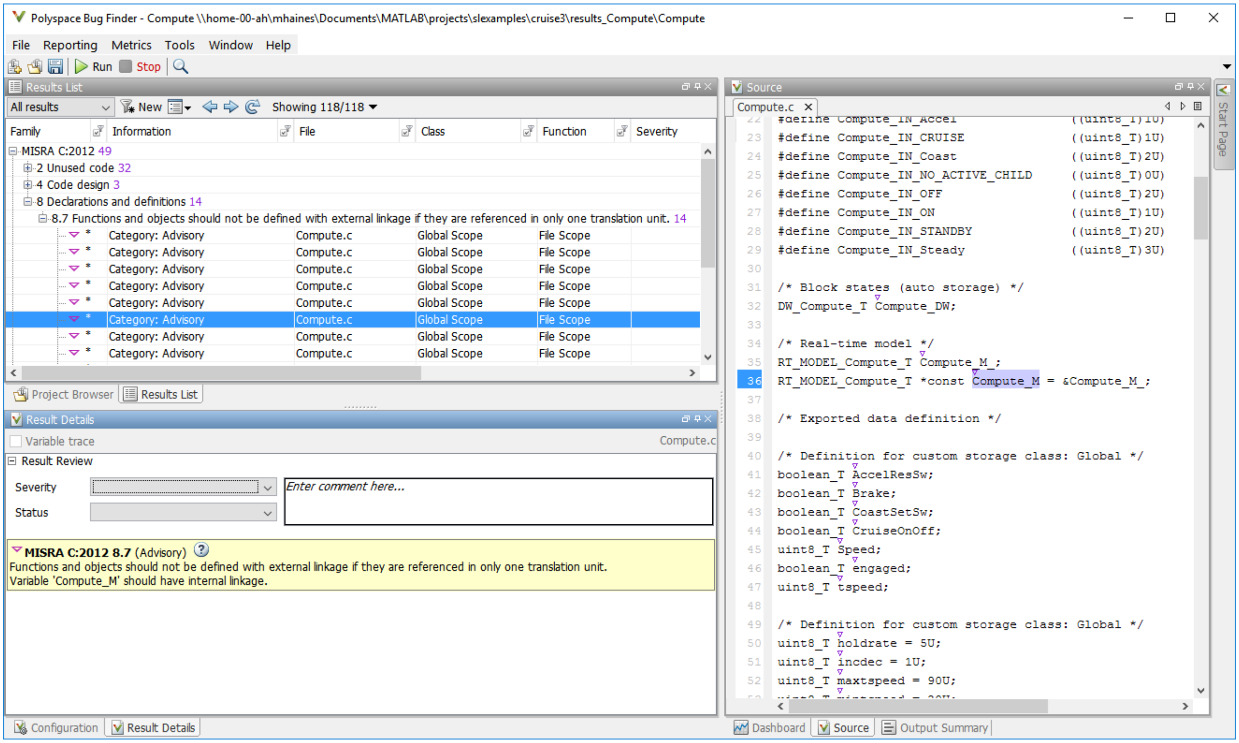
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

Review Results

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

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

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



Family	Information	File	Class	Function	Severity
MISRA C:2012 49					
2 Unused code 32					
4 Code design 3					
8 Declarations and definitions 14					
8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit. 14					
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	

Compute.c

Result Review

Severity

Status

Enter comment here...

MISRA C:2012 8.7 (Advisory)
Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
Variable 'Compute_M' should have internal linkage.

```
#define Compute_IN_Accel                ((uint8_T)1U)
#define Compute_IN_CRUISE               ((uint8_T)1U)
#define Compute_IN_Coast                ((uint8_T)2U)
#define Compute_IN_NO_ACTIVE_CHILD      ((uint8_T)0U)
#define Compute_IN_OFF                  ((uint8_T)2U)
#define Compute_IN_ON                   ((uint8_T)1U)
#define Compute_IN_STANDBY              ((uint8_T)2U)
#define Compute_IN_Steady               ((uint8_T)3U)

/* Block states (auto storage) */
DW_Compute_T Compute_DW;

/* Real-time model */
RT_MODEL_Compute_T Compute_M_;
RT_MODEL_Compute_T *const Compute_M = &Compute_M_;

/* Exported data definition */

/* Definition for custom storage class: Global */
boolean_T AccelResSw;
boolean_T Brake;
boolean_T CoastSetSw;
boolean_T CruiseOnOff;
uint8_T Speed;
boolean_T engaged;
uint8_T tspeed;

/* Definition for custom storage class: Global */
uint8_T holdrate = 5U;
uint8_T incdec = 1U;
uint8_T maxtspeed = 90U;
```

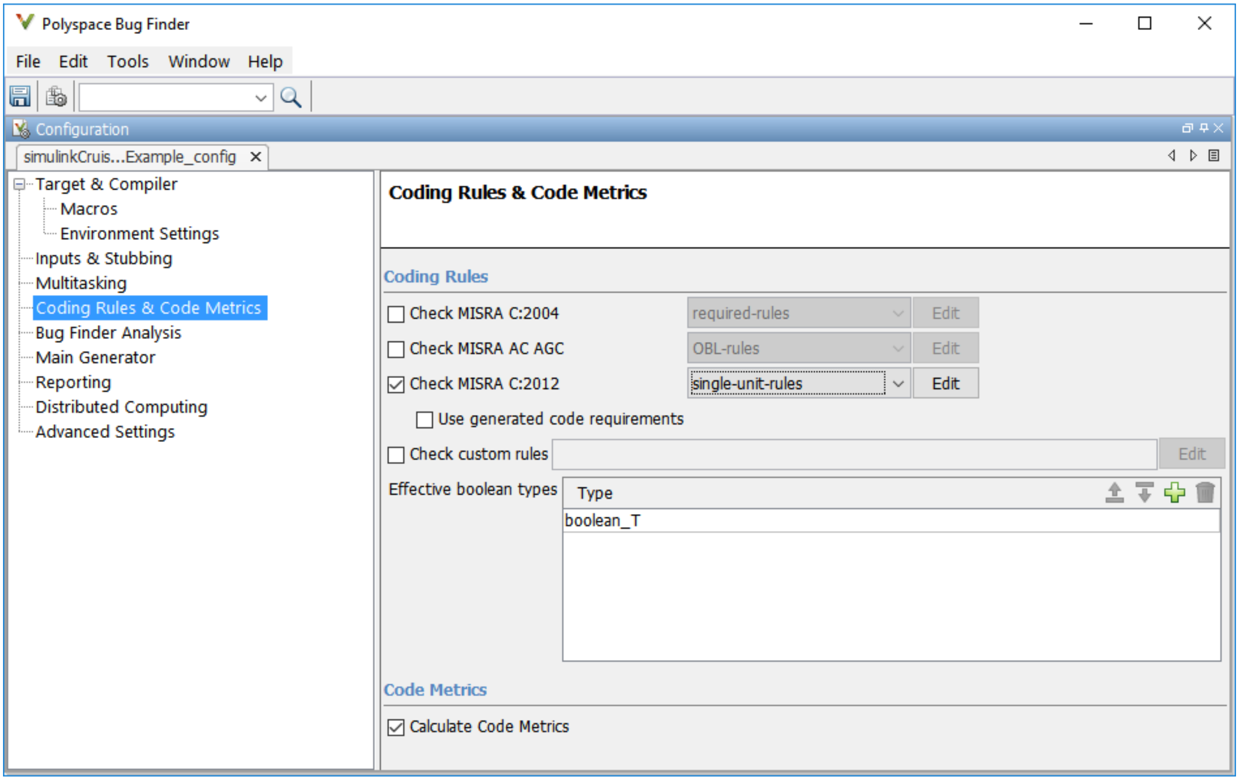
2 In your model, right-click Compute target speed and select **Polyspace > Options**.

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 Rules & Code Metrics** pane, select the check box **Check MISRA C:2012** and from the drop-down list, select

`single-unit-rules`. Now, Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.



Coding Rules & Code Metrics

Coding Rules

☐ Check MISRA C:2004 required-rules Edit

☐ Check MISRA AC AGC OBL-rules Edit

☑ Check MISRA C:2012 single-unit-rules Edit

☐ Use generated code requirements

☐ Check custom rules Edit

Effective boolean types

Type
boolean_T

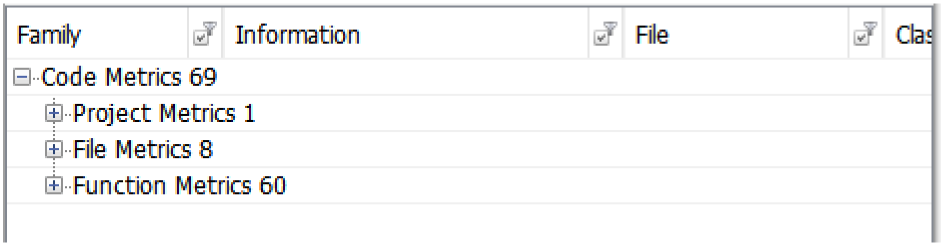
Code Metrics

☑ Calculate Code Metrics

6 Save and close the Polyspace configuration window.

7 Rerun the analysis with the new configuration.

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



Family	Information	File	Clas
⊟ Code Metrics 69			
⊞ Project Metrics 1			
⊞ File Metrics 8			
⊞ Function Metrics 60			

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 Reports" (Simulink Test)

3

Checking Systems Interactively

Check Model Compliance by Using the Model Advisor

You can use the Model Advisor to check a model or subsystem for adherence to modeling guidelines or standards. The Model Advisor includes checks that help you define and implement consistent design guidelines. Using model checks, you can apply guidelines across projects and development teams.

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

- After you complete your model design, run Model Advisor checks interactively.
- Configure the Model Advisor to check for violations while you edit.

When you use the Model Advisor to check systems, these limitations apply:

- 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 only the active subsystem.
- 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 in the documentation for a check, the Model Advisor, by default, does not analyze the contents of a Model Reference block. To run checks on referenced models, use instances of the `Advisor.Application` class.

Check Your Model Interactively by Using Model Advisor Checks

You can use the Model Advisor to check your model interactively against modeling standards and guidelines. In the model window, select **Analysis > Model Advisor > Model Advisor**. Select the model or system that you want to review. Select the checks that you want to run on your model from the **By Product** or **By Task** folders. Then run your selected checks. The Model Advisor reviews your model and, if selected, displays an HTML report of your results.

Depending on which products you have installed, the Model Advisor includes different checks.

For More Information	See
Checking model compliance with the DO-178C safety standard	"Model Checks for DO-178C/DO-331 Standard Compliance" on page 3-46
Checking model compliance with the IEC 61508, IEC 62304, ISO 26262, or EN 50182 safety standards	"Model Checks for IEC 61508, IEC 62304, ISO 26262, and EN 50128 Standard Compliance" on page 3-60
Checking model compliance with MathWorks Automotive Advisory Board (MAAB) guidelines	"Model Checks for MathWorks Automotive Advisory Board (MAAB) Guideline Compliance" on page 3-68
Checking model compliance with the MISRA C:2012 standard	"Model Checks for MISRA C:2012 Compliance" on page 3-89
Checking model compliance with CERT C, CWE, and ISO/IEC TS 17961 secure coding standards	"Model Checks for Secure Coding (CERT C, CWE, and ISO/IEC TS 17961 Standards)" on page 3-90
Checking requirements links	"Model Checks for Requirements Links" on page 3-91
Checking model metrics	"Collect Model Metrics Using the Model Advisor" on page 5-10

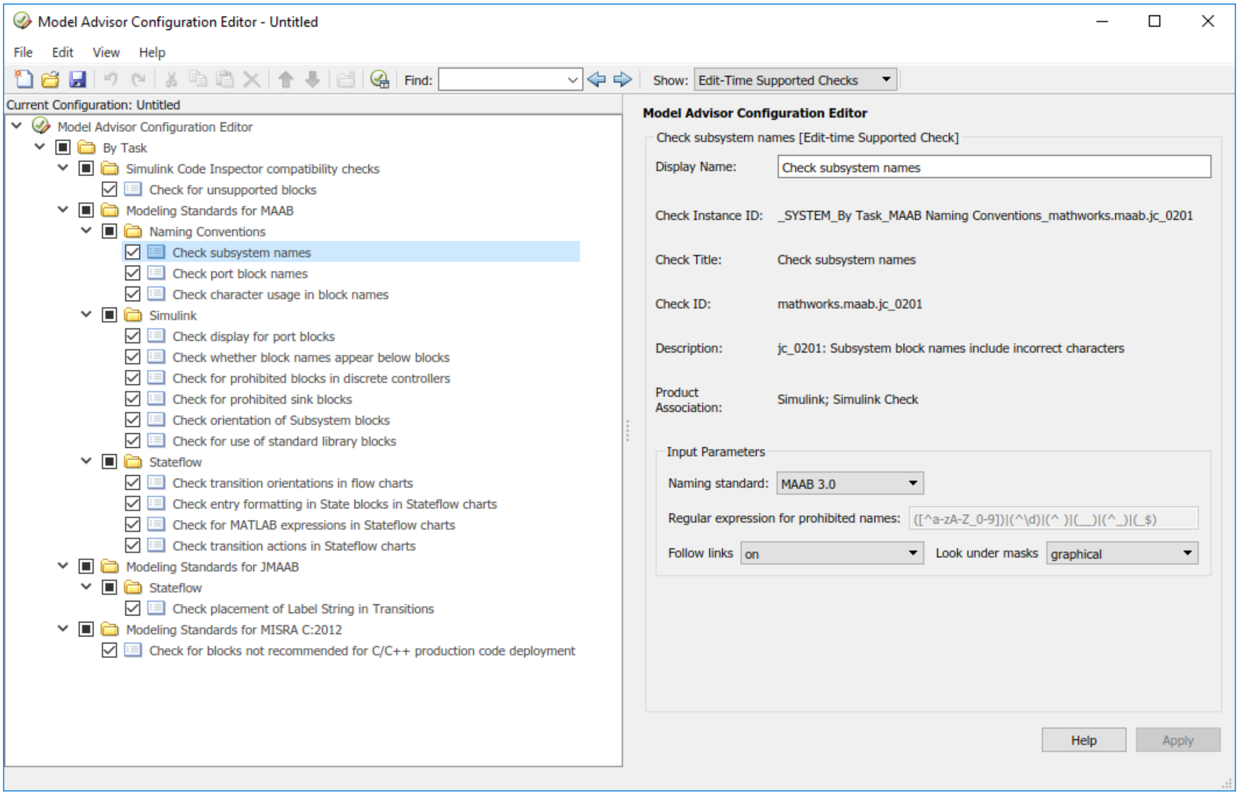
Check Your Model as You Edit by Using Edit-Time Checking

You can identify modeling guideline compliance issues earlier in the model design process by using edit-time checking. When using edit-time checking, 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.

View and Configure Edit-Time Checks

When you select edit-time checking, the Model Advisor evaluates the model against a subset of Model Advisor checks. To view and configure the Model Advisor checks that edit-time checking flags:

1. In the model window, select **Analysis > Model Advisor > Configure Advisor Edit-Time Checks**.
2. In the Model Advisor Configuration Editor, verify that the `Edit-time Supported Checks` item is selected from the **Show** drop-down list. The filtered list identifies the model advisor checks that are flagged.



Show: Edit-Time Supported Checks

Current Configuration: Untitled

- Model Advisor Configuration Editor
  - ■ By Task
    - ■ Simulink Code Inspector compatibility checks
      - ☑ Check for unsupported blocks
    - ■ Modeling Standards for MAAB
      - ■ Naming Conventions
        - ☑ Check subsystem names
        - ☑ Check port block names
        - ☑ Check character usage in block names
      - ■ Simulink
        - ☑ Check display for port blocks
        - ☑ Check whether block names appear below blocks
        - ☑ Check for prohibited blocks in discrete controllers
        - ☑ Check for prohibited sink blocks
        - ☑ Check orientation of Subsystem blocks
        - ☑ Check for use of standard library blocks
      - ■ Stateflow
        - ☑ Check transition orientations in flow charts
        - ☑ Check entry formatting in State blocks in Stateflow charts
        - ☑ Check for MATLAB expressions in Stateflow charts
        - ☑ Check transition actions in Stateflow charts
    - ■ Modeling Standards for JMAAB
      - ■ Stateflow
        - ☑ Check placement of Label String in Transitions
    - ■ Modeling Standards for MISRA C:2012
      - ☑ Check for blocks not recommended for C/C++ production code deployment

Model Advisor Configuration Editor

Check subsystem names [Edit-time Supported Check]

Display Name: Check subsystem names

Check Instance ID: _SYSTEM_By Task_MAAB Naming Conventions_mathworks.maab.jc_0201

Check Title: Check subsystem names

Check ID: mathworks.maab.jc_0201

Description: jc_0201: Subsystem block names include incorrect characters

Product Association: Simulink; Simulink Check

Input Parameters

Naming standard: MAAB 3.0

Regular expression for prohibited names: ([^a-zA-Z_0-9])|(^\d)|(^)|(__)|(^_)|(_$)

Follow links: on

Look under masks: graphical

3 Select or clear checks as needed. Selected checks are included in the edit-time check analysis. You can use the **Input Parameters** options to customize each check.

4 If you have made updates to check selection or behavior, save the current configuration. Then select **File > Set Current Configuration as Default**.

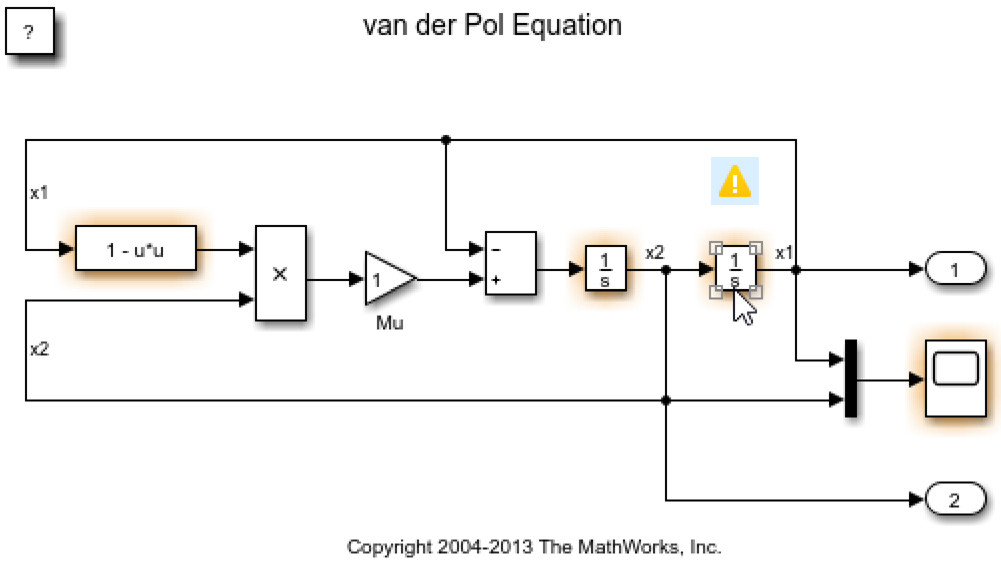
Note Only the default configuration can change the behavior of edit-time checks.

To customize the behavior of edit-time checks, configure updates in the filtered view of edit-time checks in the Model Advisor Configuration Editor. If a check appears in multiple folders of your Model Advisor tree, for edit-time checking, Model Advisor assigns priority to the check in your custom folder. If the check is not in your custom folder, priority goes to the check in the **By Task** folder, and finally to the check in your **By Product** folder.

Use Edit-Time Checking to Highlight Compliance Issues in the Model Editor

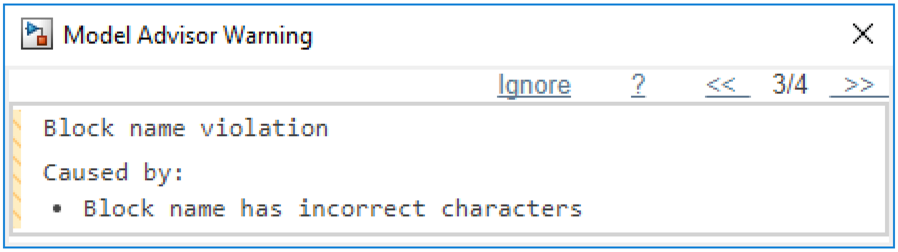
When using edit-time checking, the Model Advisor highlights blocks that violate the Model Advisor checks. To enable edit-time checking of your model, in the model window, select **Analysis** > **Model Advisor** > **Display Advisor Checks in Editor**.

Place your cursor over a highlighted block and click the error or warning icon.



The Model Advisor identifies compliance issues in the block that violate edit-time checks. When a block has multiple check violations, you can move between the edit-time violations by using the **<<** and **>>** buttons. For each issue, you can:

- Review the cause .
- Click the question mark to access detailed documentation about the flagged Model Advisor check.
- Ignore a warning for a block and add the block to the exclusion list for that check by clicking the **Ignore** button.



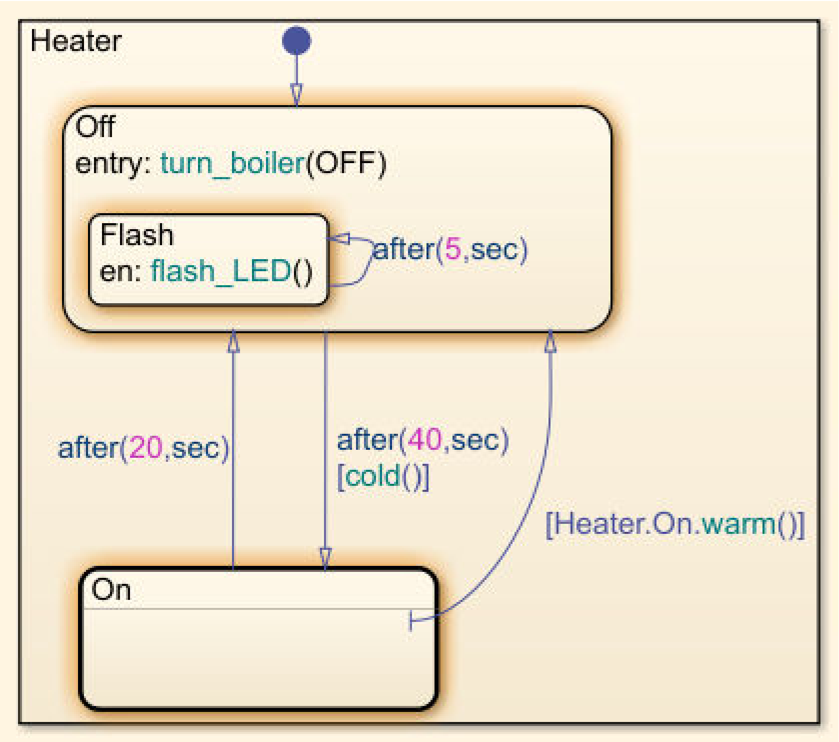
```
Block name violation
Caused by:
  • Block name has incorrect characters
```

Software is inherently complex and may not be completely free of errors. Model Advisor checks might contain bugs. MathWorks reports known bugs brought to its attention on its Bug Report system at https://www.mathworks.com/support/bugreports/. The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

While applying Model Advisor checks to your model will increase the likelihood that your model does not violate certain modeling standards or guidelines, it is ultimately your responsibility to verify, using multiple methods, that the system being developed provides its intended functionality and does not include any unintended functionality.

In this example, you use edit-time checking to verify compliance of Stateflow charts with the MAAB guidelines while you edit:

1. Open your model that contains Stateflow charts. For example, at the command prompt, open model `sf_boiler`.
2. To enable the edit-time checking, go to **Analysis** > **Model Advisor** > **Display Advisor Checks in Editor**.
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 **en:** to comply with the MAAB guidelines. In your model, place your cursor after **en:** and press **Enter**. A new line is added and the warning is cleared.

See Also

Related Examples

- "Select and Run Model Advisor Checks" (Simulink)

More About

- "Detect Modeling Errors During Edit Time" (Stateflow)
- "Select and Run Model Advisor Checks" (Simulink)

- “Model Advisor Limitations” (Simulink)

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:

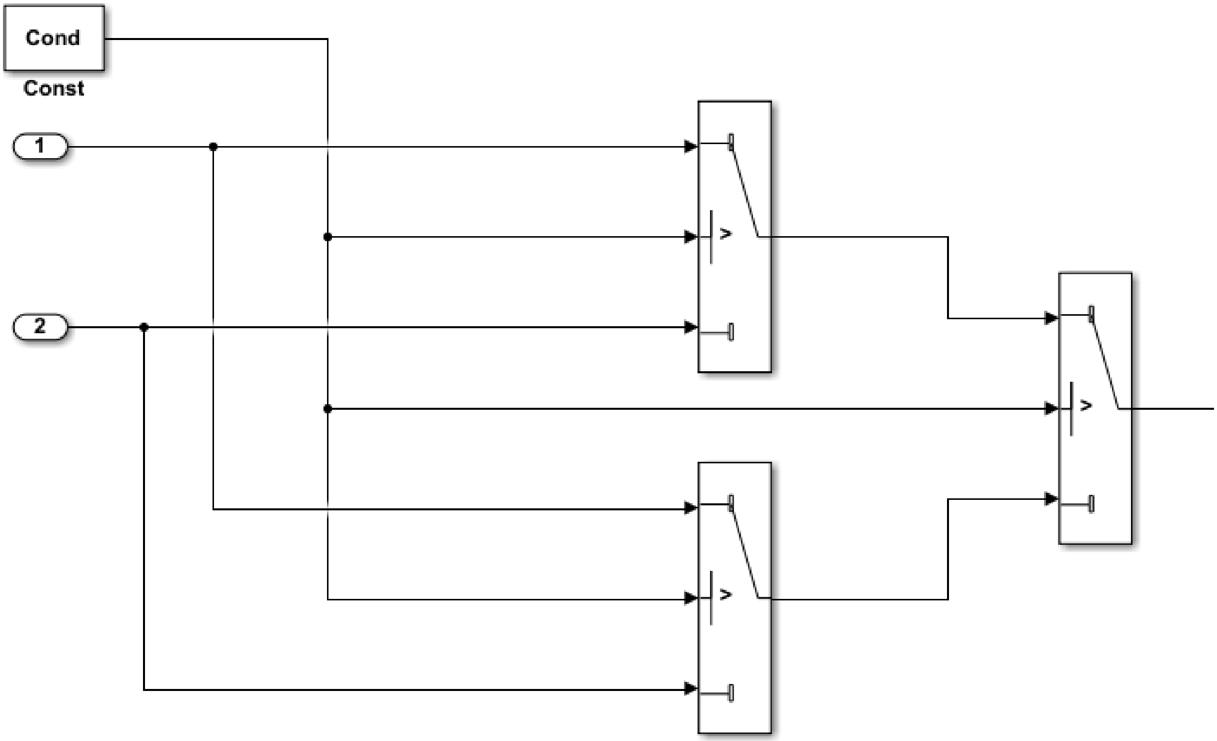
- If an If block connects to one or more If Action Subsystems and each one has one outport, replace this modeling pattern with a subsystem and a Variant Source block.
- If an If block connects to an If Action Subsystem that does not have an outport or has two or more outports, replace this modeling pattern 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`.

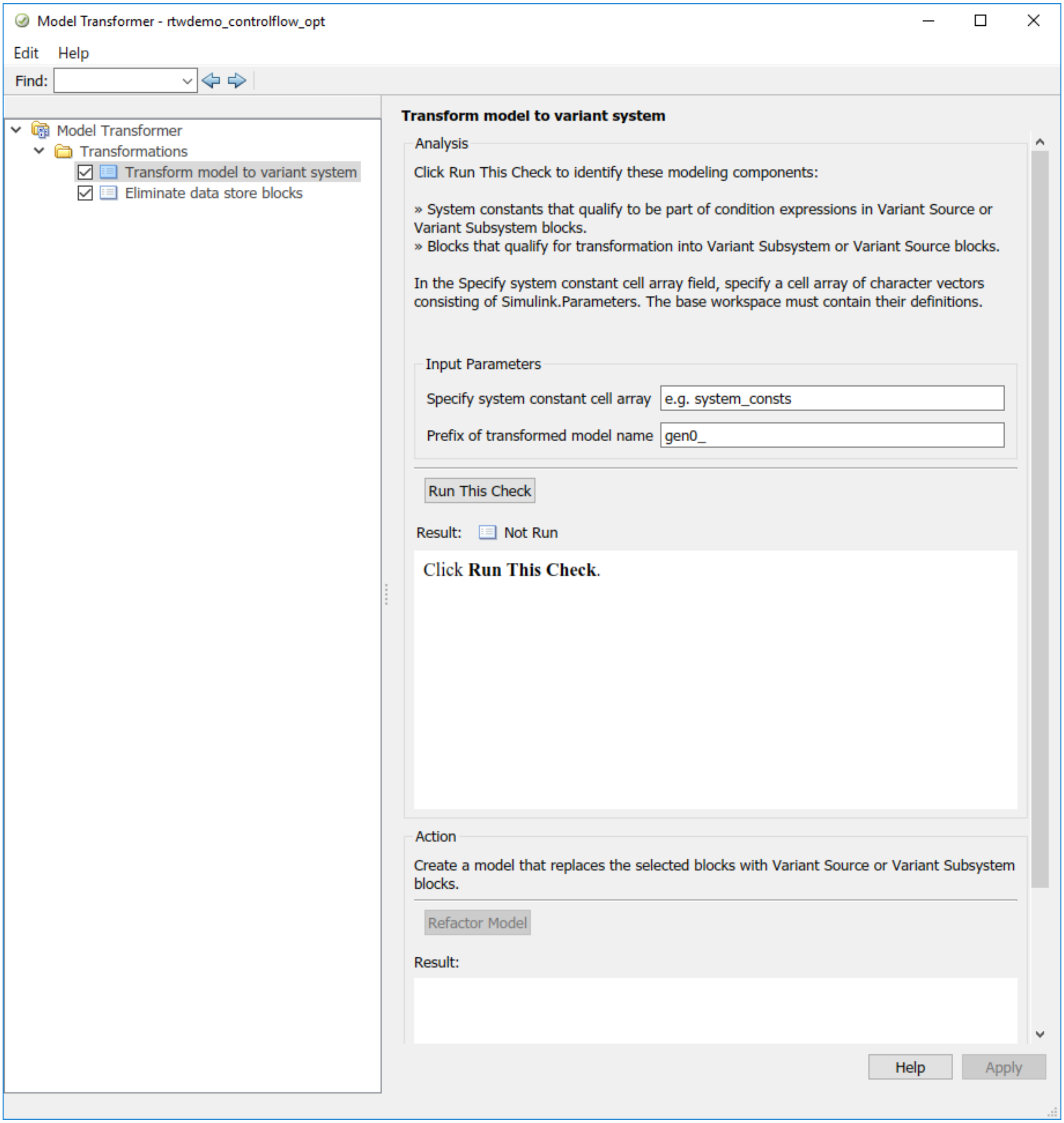
2 Open the `Switch1` Block Parameters dialog box. Change the **Threshold** parameter to `0`. The **Threshold** parameter must be an integer because after the variant transformation it is part of the condition expression in the Variant Source block.

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. From the Model Editor, open the Model Transformer by selecting **Analysis** > **Refactor Model** > **Model Transformer**. Or, in the Command Window, type: `mdltransformer('rtwdemo_controlflow_opt')`
2. Select the check “Transform the model to variant system”.



Model Transformer - rtwdemo_controlflow_opt
Edit
Help
Find:
Model Transformer
Transformations
☑ Transform model to variant system
☑ Eliminate data store blocks

Transform model to variant system

Analysis

Click Run This Check to identify these modeling components:

» System constants that qualify to be part of condition expressions in Variant Source or Variant Subsystem blocks.
» Blocks that qualify for transformation into Variant Subsystem or Variant Source blocks.

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

Input Parameters

Specify system constant cell array: e.g. system_consts

Prefix of transformed model name: gen0_

Result: Not Run

Click **Run This Check**.

Action

Create a model that replaces the selected blocks with Variant Source or Variant Subsystem blocks.

Result:

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 Identify Modeling Clones tool identifies clones across referenced model boundaries. You can use the Identify Modeling Clones tool to enable component reuse 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.

To open the tool, in the Simulink Editor, select **Analysis** > **Refactor Model** > **Identify Modeling Clones**.

Exact Clones Versus 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 different parameters** to `0` (default value). Increasing this parameter value increases the number of similar clones that the tool can potentially detect.

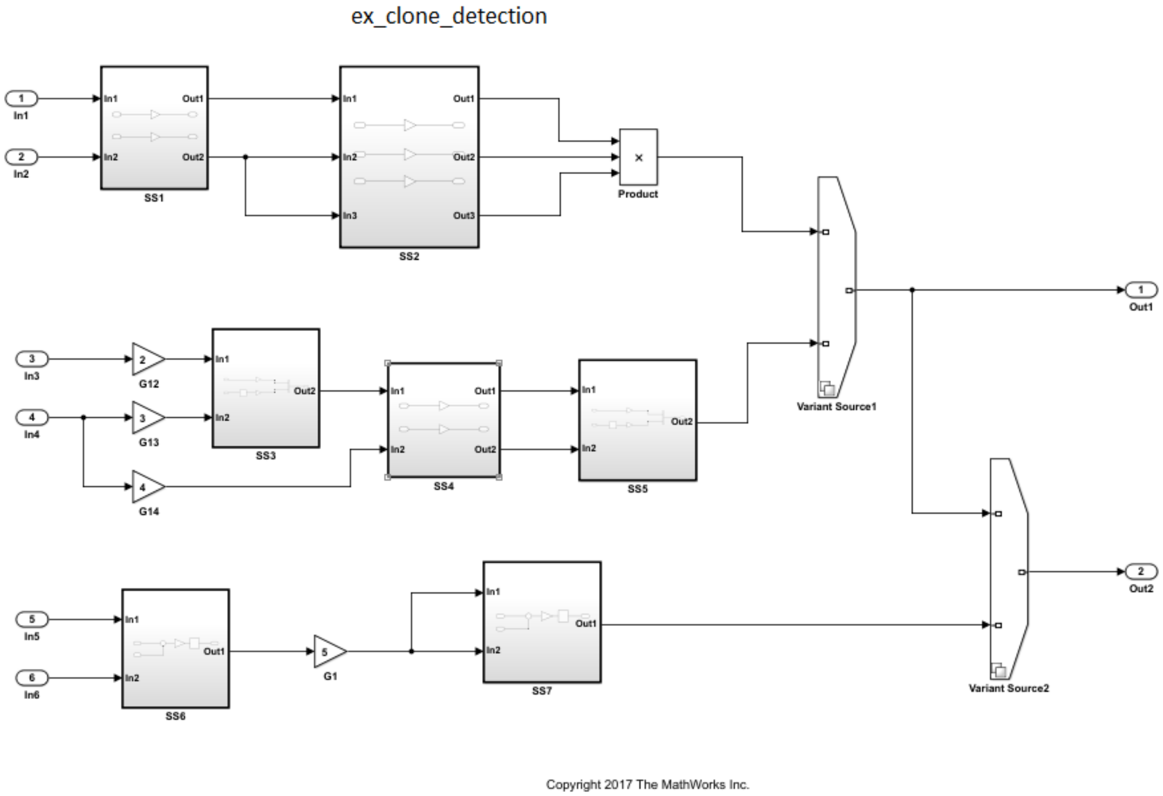
After you identify clones, you can replace them with links to library blocks. Similar clones link to masked library subsystems.

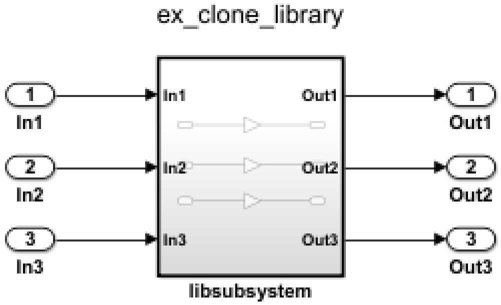
Identify Exact and Similar Clones

This example shows how to use the Identify Modeling Clones tool to identify exact clones and similar clones, and then replace them with links to library blocks.

1 Open the model `ex_clone_detection` and the corresponding library `ex_clone_library`. At the MATLAB® command line, enter:

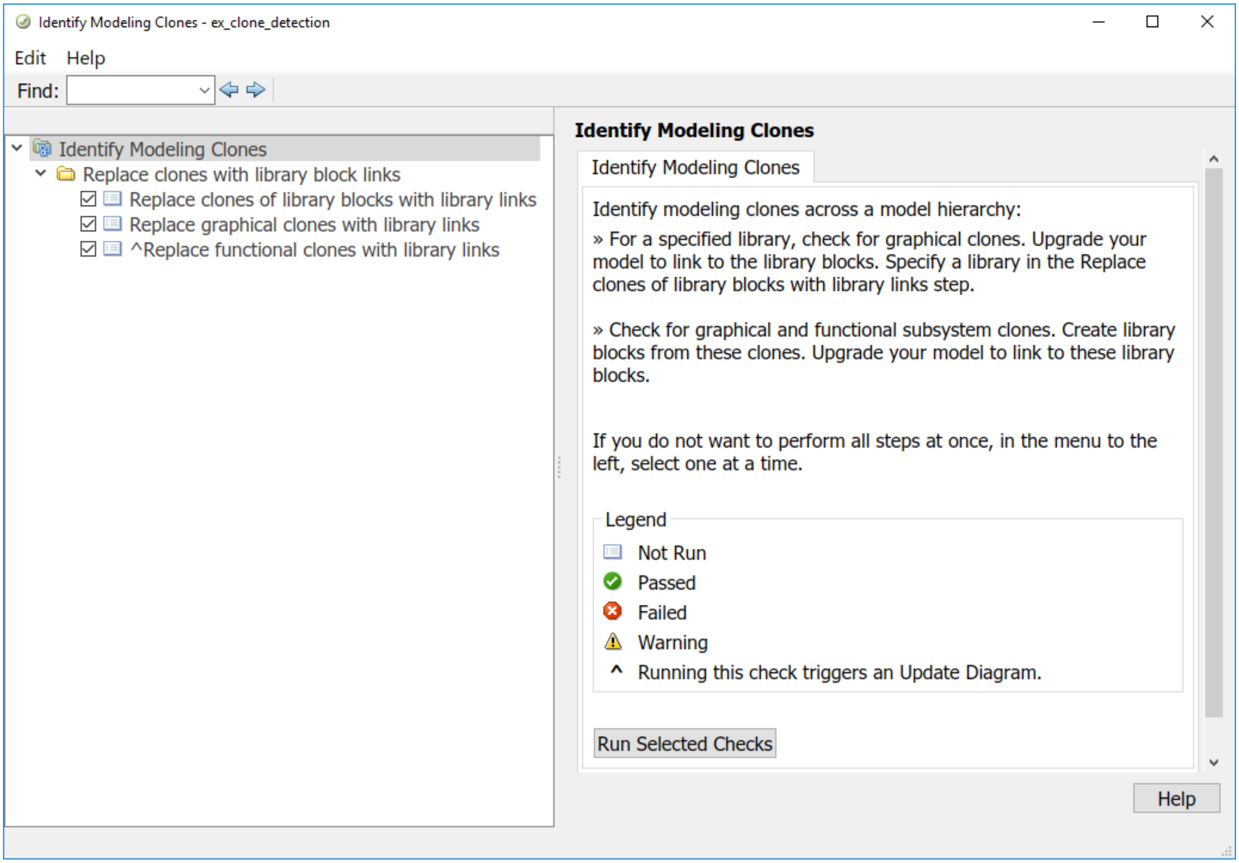
```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
ex_clone_detection
ex_clone_library
```





2 Save the model to your working folder.

3 In the Simulink Editor, from the **Analysis** menu, select **Refactor Model** > **Identify Modeling Clones**. To open the Identify Modeling Clones tool programmatically, at the MATLAB command prompt, type: `clonedetection('ex_clone_detection')`.



- Identify Modeling Clones
  - Replace clones with library block links
    - ☑ Replace clones of library blocks with library links
    - ☑ Replace graphical clones with library links
    - ☑ ^Replace functional clones with library links

Identify Modeling Clones

Identify Modeling Clones

Identify modeling clones across a model hierarchy:

» For a specified library, check for graphical clones. Upgrade your model to link to the library blocks. Specify a library in the Replace clones of library blocks with library links step.

» Check for graphical and functional subsystem clones. Create library blocks from these clones. Upgrade your model to link to these library blocks.

If you do not want to perform all steps at once, in the menu to the left, select one at a time.

Legend

▤	Not Run
✓	Passed
✕	Failed
⚠	Warning
^	Running this check triggers an Update Diagram.

4 Select the folder **Replace clones with library block links**. If you want to perform all or some of the checks in the Identify Modeling Clones tool, you can click **Run Selected Checks**. Selecting this option does not refactor the model. It only identifies the clones. This example takes you through each check, one at a time.

Replace Clones of Library Blocks with Library Links

Identify modeling patterns that are graphical clones of a library subsystem. Graphical clones can be in modeling regions that include inactive variants and commented-out regions. If one clone has a link to a library block, the tool reports a missing link for the other subsystem or subsystem clones. The tool also reports clones that do not have links to library blocks. You choose whether to create a library block and replace a clone with a link to that library block.

1. Select **Replace clones of library blocks with library links**.
2. In the **Library file name** field, specify the library `ex_clone_library`.
3. Leave the **Maximum number of different parameters** value as `0`.
4. Click **Run This Check**. In the top **Result** table, the left column contains hyperlinks to modeling clones. The right column contains hyperlinks to the corresponding library subsystems. The Gain blocks `G12`, `G13`, and `G14` and `SS2` are clones of `libsubsystem`.
5. Click **Refactor Model**.
6. In the bottom **Result** table, there is a message informing you that the model was successfully refactored. The **Refactor Model** button is now unavailable, and the **Undo** button is enabled.
7. The model now contains links to `libsubsystem`. To remove the linked library blocks, click the **Undo** button. After you refactor, you can remove the latest changes to the model by clicking the **Undo** button. Each time you refactor a model, the tool creates a back-up model in the folder that has the prefix `m2m_`plus the model name.

Note The Identify Modeling Clones tool identifies clones that are similar to library blocks. It does not refactor a model to replace similar clones with links to library blocks.

Replace Graphical Clones with Library Links

Now identify graphical subsystem clones and replace them with links to library blocks. If you do not want to refactor a model to replace clones in inactive variants or commented-out regions, you can skip this check and instead run the **Replace functional clones with library links** check.

1. Select **Replace graphical clones with library links**.
2. In the **New library file name** field, specify a library name or use the default name.
3. Change the **Maximum number of different parameters** value to `2`.
4. Click **Run This Check**. The top **Result** table contains separate groupings for exact and similar clones. `Exact Clone Group 1` contains hyperlinks to the subsystem clones. `SS1` and `SS4`. `Similar Clone Group 1` contains hyperlinks to `SS3` and `SS5`. `Similar Clone Group 2` contains hyperlinks to `SS6` and `SS7`.
5. Click the + symbol to reveal the contents in the second row and second column of the Result table. `SS5` has one block and one parameter that is different from `SS3`. SS3 is the baseline clone for comparison.

6. Click **Refactor Model**.
7. In the bottom **Result** table, there is a message informing you that the model was refactored. The **Refactor Model** button is now unavailable, and the **Undo** button is enabled.
8. For each clone group, the refactored model contains links to library subsystems. `Similar Clone Group 1` and `Similar Clone Group 2` link to masked library subsystems.

Replace Functional Clones with Library Links

Identify functional subsystem clones and replace them with links to library blocks. If you want to refactor a model to replace clones in active modeling regions and inactive variants and commented-out regions, you can skip this check and instead run the **Replace graphical clones with library links** check.

1. Select **Replace functional clones with library links**.
2. In the **New library file name** field, specify a name for the library or use the default name.
3. Click **Run This Check**. The top **Result** table does not list new clones because the **Replace Graphical Clones with Library Links** step identifies functional clones.

Save and View Clone Detection Reports

When the Identify Modeling Clones tool runs checks, it generates an HTML report of check results. By default, the HTML report is in the `slprj/modeladvisor/` folder. The Identify Modeling Clones tool uses the `slprj` folder in the code generation folder to store reports and other information. If the `slprj` folder does not exist in the code generation folder, the Identify Modeling Clones tool creates it.

View the report in the Identify Modeling clones tool by clicking the link on the **Replace clones with library block links** folder. Save the report to a new location by clicking the **Save As** button and specifying a location.

Additional Information

- You can run the Identify Modeling Clones tool on a library.
- You can exclude Subsystem and Model Reference blocks from clone detection by right-clicking the subsystem or Model Reference block and selecting **Identify Modeling**

Clones > Subsystem and its contents > Add to exclusions. For more information, see “Exclude subsystems and referenced models from clone detection”.

- For additional practice using the Identify Modeling Clones tool, try the model `aero_dap3dof` and the corresponding libraries `aero_librcs` and `aero_libdap`.

See Also

Related Examples

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

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.

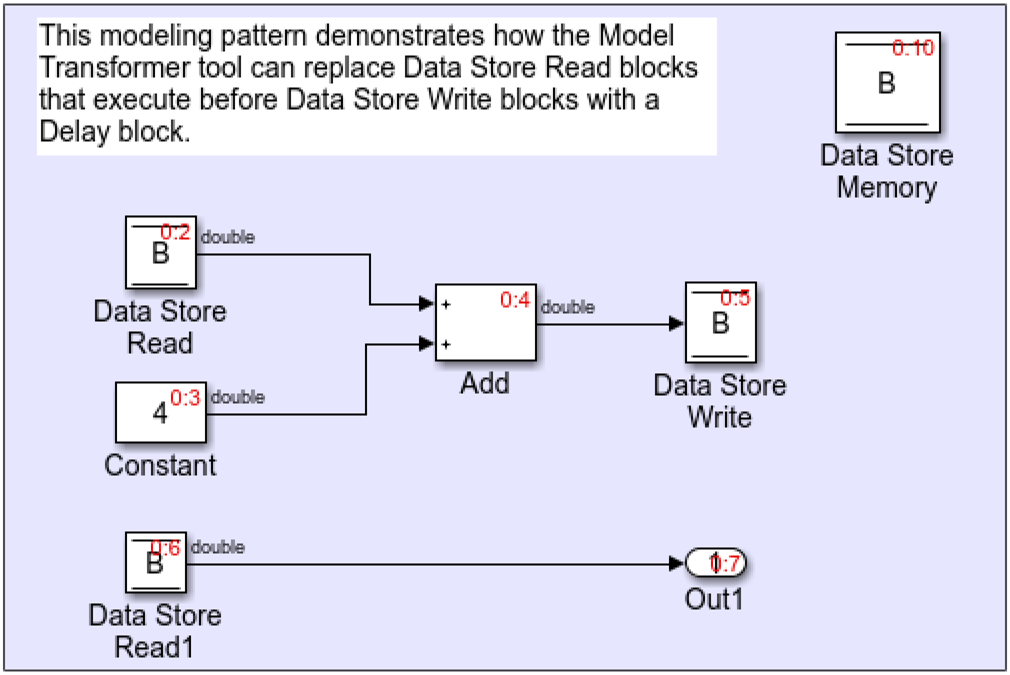
The Model Transformer can replace these data stores:

- 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.

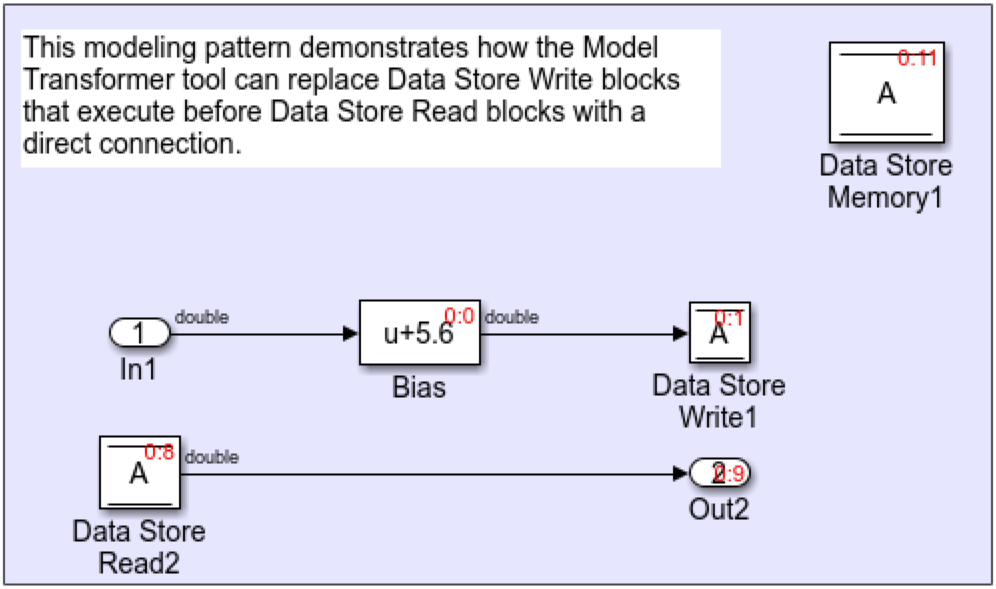
The Model Transformer tool eliminates only local data stores that Data Store Memory blocks define. The tool does not eliminate global data stores. For the Data Store Memory block, on the **Signal Attributes** tab in the block parameters dialog box, you must clear the **Data store name must resolve to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



This modeling pattern demonstrates how the Model Transformer tool can replace Data Store Read blocks that execute before Data Store Write blocks with a Delay block.



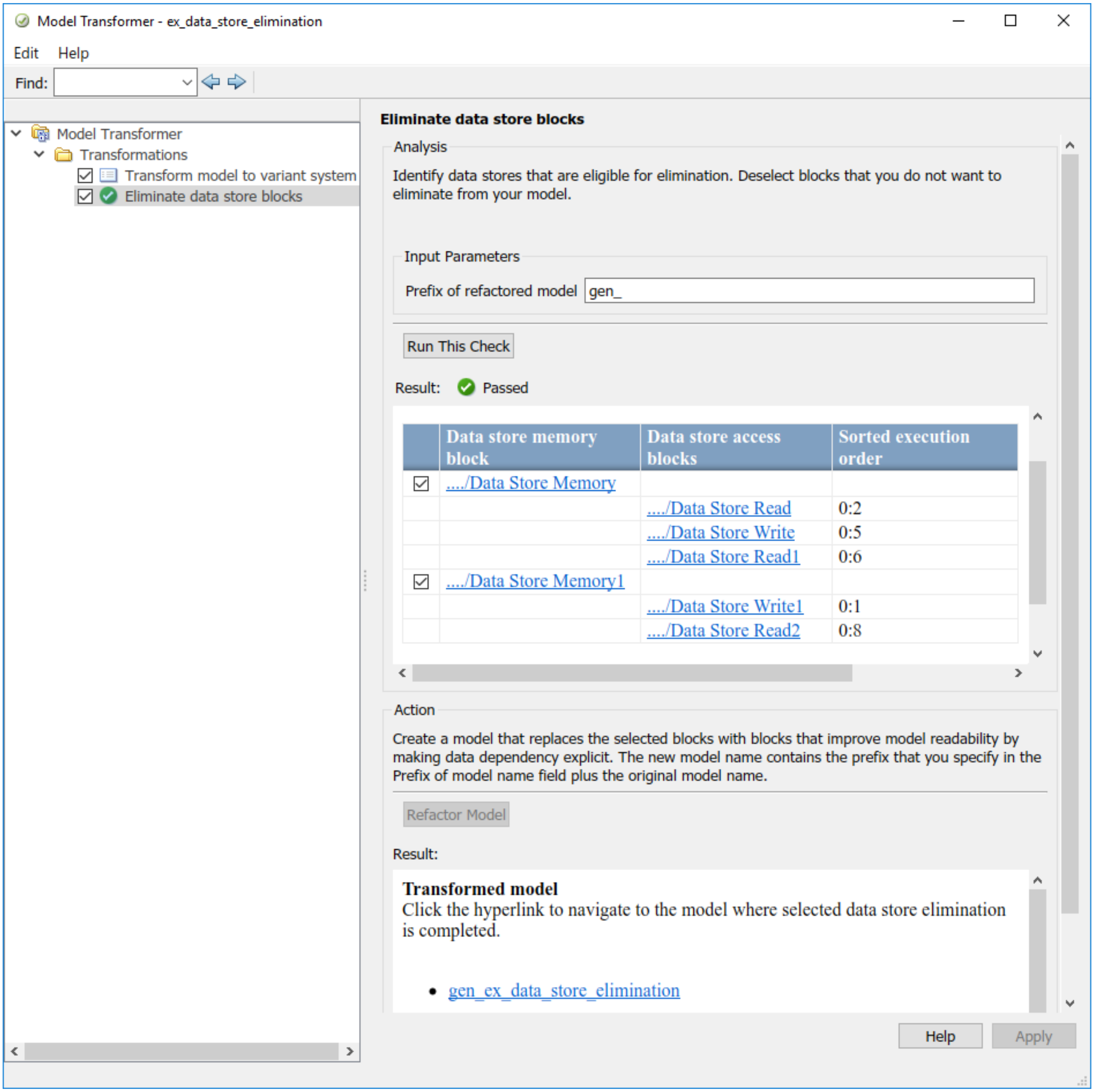
This modeling pattern demonstrates how the Model Transformer tool can replace Data Store Write blocks that execute before Data Store Read blocks with a direct connection.

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.



Model Transformer - ex_data_store_elimination
Edit
Help
Find:
Model Transformer
Transformations
☑ Transform model to variant system
☑ Eliminate data store blocks

Eliminate data store blocks

Analysis

Identify data stores that are eligible for elimination. Deselect blocks that you do not want to eliminate from your model.

Input Parameters

Prefix of refactored model: gen_

Result: ✓ Passed

	Data store memory block	Data store access blocks	Sorted execution order
☑	..../Data Store Memory		
		..../Data Store Read	0:2
		..../Data Store Write	0:5
		..../Data Store Read1	0:6
☑	..../Data Store Memory1		
		..../Data Store Write1	0:1
		..../Data Store Read2	0:8

Action

Create a model that replaces the selected blocks with blocks that improve model readability by making data dependency explicit. The new model name contains the prefix that you specify in the Prefix of model name field plus the original model name.

Result:

Transformed model
Click the hyperlink to navigate to the model where selected data store elimination is completed.

- gen_ex_data_store_elimination

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)

Limit Model Checks

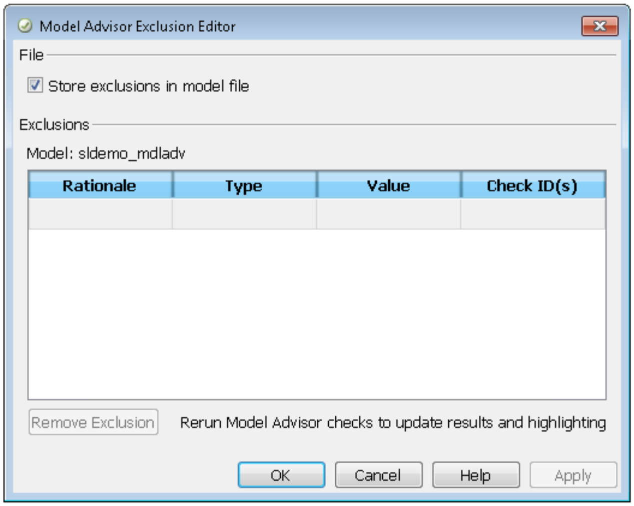
What Is a Model Advisor Exclusion?

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

- Simulink blocks
- Stateflow® charts

After you specify the blocks to exclude, Model Advisor uses the exclusion information to exclude blocks from specified checks during analysis. By default, Model Advisor exclusion information is stored in the model SLX file. Alternately, you can store the information in an exclusion file.

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.



Model Advisor Exclusion Editor

File

☑ Store exclusions in model file

Exclusions

Model: sldemo_mdladv

Rationale	Type	Value	Check ID(s)

Rerun Model Advisor checks to update results and highlighting

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**
Exclude the block from a failed check. After a Model Advisor analysis, this option is available.	**Exclude block only** > **<name of failed check>**
Exclude all blocks of this type from a failed check. After a Model Advisor analysis, this option is available.	**Exclude all blocks with type <block_type>** > **<name of failed check>**

2 In the Model Advisor Exclusion Editor dialog box, to:

- Store exclusions in model file, select **Store exclusions in model file**. Click **OK** or **Apply** to create the exclusion.
- Save the information to an exclusion file, clear **Store exclusions in model file**. Click **OK** or **Apply**. If this exclusion is the first one, a Save Exclusion File as dialog box opens. In this dialog box, click **Save** to create a exclusion file with the default name *<model_name>*`_exclusions.xml` in the current folder. Optionally, you can select a different file name or location.

3 Optionally, if you want to change the exclusion file name or location:

   a In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**.

   b In the Model Advisor Exclusion Editor dialog box, select **Change**.

   c In the Change Exclusion File dialog box, select **Save as**.

   d In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.

   e In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information to an exclusion file.

You can create as many Model Advisor exclusions as you want by right-clicking model blocks and selecting **Model Advisor**. Each time that you create an exclusion, the Model Advisor Exclusion Editor dialog box opens. In the **Rationale** field, you can specify a reason for excluding blocks or checks from the Model Advisor analysis. The rationale is useful to others who review your model.

If you create an exclusion file and save your model, you attach the exclusion file to your model. Each time that you open the model, the blocks and checks specified in the exclusion file are excluded from the analysis.

Review Model Advisor Exclusions

You can review the exclusions associated with your model. Before or after a Model Advisor analysis, to view exclusions information:

- Right-click in the model window or right-click a block and select **Model Advisor > Open Model Advisor Exclusion Editor**. The Model Advisor Exclusion Editor dialog box lists the exclusions for your model.
- On the Model Advisor toolbar, select **Settings > Preferences**. In the Model Advisor Preferences dialog box, select **Show Exclusion tab**. In the right pane of the Model

Advisor window, select the **Exclusions** tab to display checks that are excluded from the Model Advisor analysis.

- In the model window, select **Analysis** > **Model Advisor** > **Model Advisor** to open the Model Advisor.

  1. On the Model Advisor window toolbar, select **Highlighting** > **Highlight Exclusions**. By default, this menu option is selected.
  2. In the Model Advisor window, click **Enable highlighting** ().
  3. In the left pane of the Model Advisor window, select a check. The blocks excluded from the check appear in the model window, highlighted in gray with a black border.

After the Model Advisor analysis, you can view exclusion information for individual checks in the:

- HTML report. Before the analysis, in the Model Advisor window, make sure that you select the **Show report after run** check box.

- Model Advisor window. In the left pane of the Model Advisor window, select **By Product** > **Simulink** > **< name of check >**. If the **By Product** folder is not displayed, select **Show By Product Folder** from the **Settings** > **Preferences** dialog box.

If the check	The HTML report and Model Advisor window
Has no exclusions rules applied.	Show that no exclusions were applied to this check.
Does not support exclusions.	Shows that the check does not support exclusions.
Is excluded from a block.	Lists the check exclusion rules.

Manage Exclusions

Save Exclusions in a File

1. In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file** and click **OK** or **Apply**. If this exclusion is the first one, a Save Exclusion File as dialog box opens. In this dialog box, click **Save** to create an exclusion file with the

default name `<model_name>_exclusions.xml` in the current folder. Optionally, you can select a different file name or location.

2 If you want to change the exclusion file name or location:

   a In the Model Advisor Exclusion Editor dialog box, select **Change**.

   b In the Change Exclusion File dialog box, select **Save as**.

   c In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.

   d In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information in an exclusion file.

Load an Exclusion File

To load an existing exclusion file for use with your model:

1. In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**. Click **Change**.
2. In the Change Exclusion File dialog box, click **Load**.
3. Navigate to the exclusion file that you want to use with your model. Select **Open**.
4. In the Model Advisor Exclusion Editor dialog box, click **OK** to associate the exclusion file with your model.

Detach an Exclusion File

To detach an exclusion file associated with your model:

1. In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**. Click **Change**.
2. In the Change Exclusion File dialog box, click **Detach**.
3. In the Model Advisor Exclusion Editor dialog box, click **OK**.

Remove an Exclusion

1. In the Model Advisor Exclusion Editor dialog box, select the exclusions that you want to remove.
2. Click **Remove Exclusion**.

Add a Rationale to an Exclusion

You can add text that describes why you excluded a particular block or blocks from selected checks during Model Advisor analysis. A description is useful to others who review your model.

1. In the Model Advisor Exclusion Editor dialog box, double-click the **Rationale** field for the exclusion.
2. Delete the existing text.
3. Add the rationale for excluding this object.

Programmatically Specify an Exclusion File

You can use the `MAModelExclusionFile` method to programmatically specify the name of an exclusion file.

1. Use `get_param` to obtain the model object. For example, for `sldemo_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.

Edit-Time Exclusions

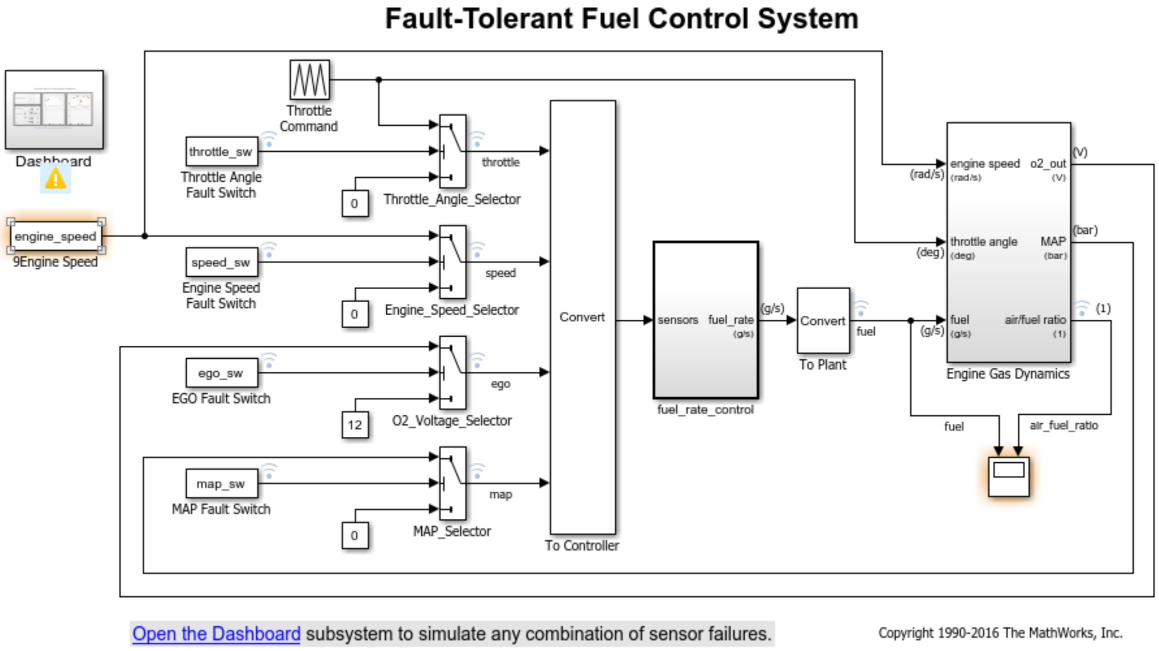
Exclude Checks During Edit-Time

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. This number causes a violation in “Check character usage in block names”.

3 In the menu bar, select **Analysis** > **Model Advisor** > **Display Advisor Checks in Editor**. The Scope block flags the warning `Block name has incorrect characters`.



Open the Dashboard subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

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 and click the **Ignore** button. For this block, clicking **Ignore** adds an exclusion to Model Advisor analysis.

The block is excluded from Model Advisor analysis for that check and no longer displays a highlight. You can repeat this process for further edit-time warnings.

Note The list of edit-time exclusions is shared between the Model Advisor and edit-time checking.

See Also

Related Examples

- “Limit Model Checks by Excluding Gain and Outport Blocks” on page 3-36
- “Exclude Blocks From Custom Checks” on page 7-83

More About

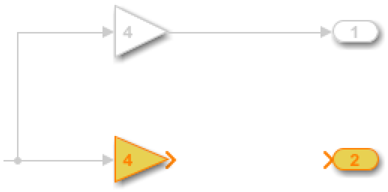
- “Select and Run Model Advisor Checks” (Simulink)

Limit Model Checks by Excluding Gain and Outport Blocks

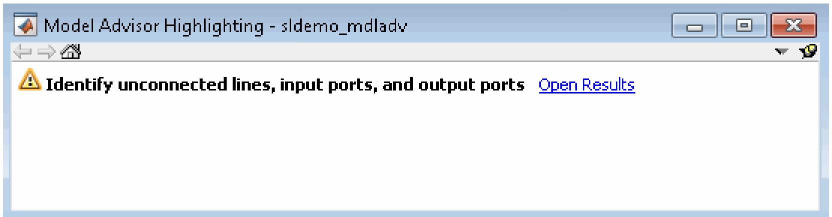
This example shows how to exclude a Gain block and all Outport blocks from a Model Advisor check during a Model Advisor analysis. By excluding individual blocks from checks, you limit the scope of the analysis and might save time during model development and verification.

1. At the MATLAB command line, type `sldemo_mdladv`.
2. From the model window, select **Analysis** > **Model Advisor** > **Model Advisor** to open the Model Advisor.
3. A System Selector — Model Advisor dialog box opens. Click **OK**.
4. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings** > **Preferences** dialog box.
5. In the left pane of the Model Advisor window, expand **By Product > Simulink**. Select the **Show report after run** check box to see an HTML report of check results after you run the checks.
6. Run the selected checks by clicking the **Run selected checks** button. After the Model Advisor runs the checks, an HTML report displays the check results in a browser window. The check **Identify unconnected lines, input ports, and output ports** triggers a warning.
7. In the left pane of the Model Advisor window, select the check **By Product > Simulink > Identify unconnected lines, input ports, and output ports**.
8. In the Model Advisor window, click the **Enable highlighting** button ().

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



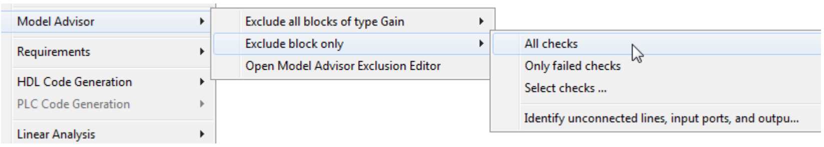
- The Model Advisor Highlighting window opens with a link to the Model Advisor window. In the Model Advisor window, you can find more information about the check results and how to fix the warning condition.



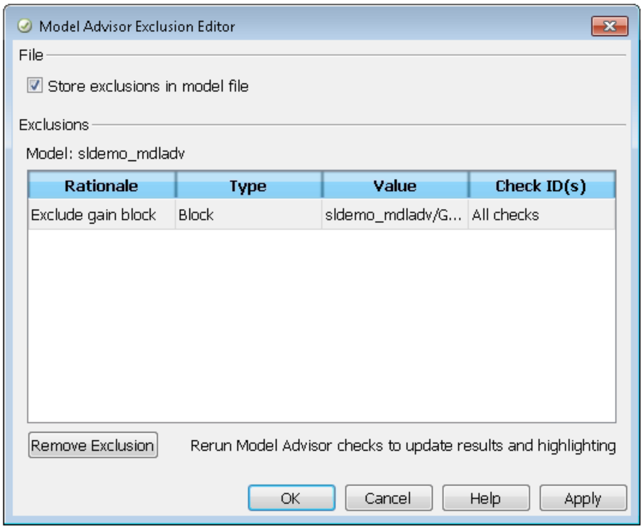
Identify unconnected lines, input ports, and output ports Open Results

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

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



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



Model Advisor Exclusion Editor

File

☑ Store exclusions in model file

Exclusions

Model: sldemo_mdladv

Rationale	Type	Value	Check ID(s)
Exclude gain block	Block	sldemo_mdladv/G...	All checks

Remove Exclusion
Rerun Model Advisor checks to update results and highlighting
OK
Cancel
Help
Apply

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

10 After reviewing the check results, exclude all Outport blocks from the Identify unconnected lines, input ports, and output ports check:

a Right-click the Out4 block and select **Model Advisor** > **Exclude all blocks of type Outport** > **Identify unconnected lines, input ports, and output ports**.

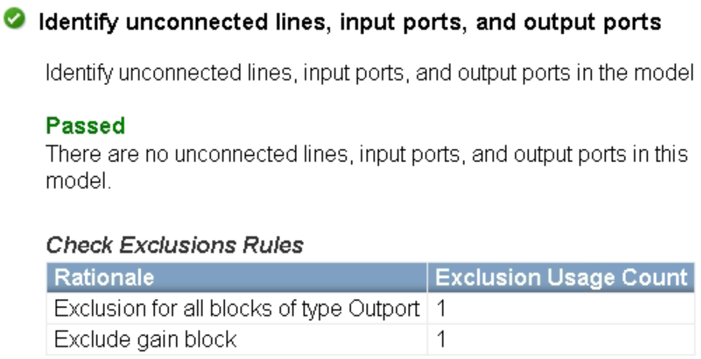
b In the Model Advisor Exclusion Editor dialog box, click **OK** to store the exclusion in the model file.

11 In the left pane of the Model Advisor window, select **By Product** > **Simulink** and then:

- Select the **Show report after run** check box.
- Click **Run Selected Checks** to run a Model Advisor analysis.

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

- HTML report:



✓ **Identify unconnected lines, input ports, and output ports**

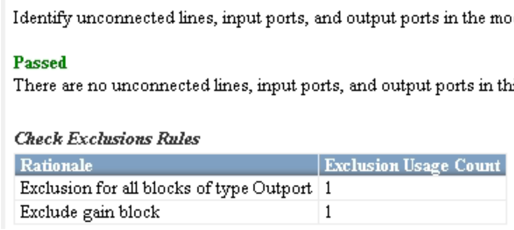
Identify unconnected lines, input ports, and output ports in the model

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

Check Exclusions Rules

Rationale	Exclusion Usage Count
Exclusion for all blocks of type 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

13 Close `sldemo_mdladv`.

See Also

Related Examples

- “Exclude Blocks From Custom Checks” on page 7-83
- “Select and Run Model Advisor Checks” (Simulink)

More About

- “Limit Model Checks” on page 3-27
- “Select and Run Model Advisor Checks” (Simulink)

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 performance of simulation for linear interpolations. The Model Transformer creates a model with these replacements blocks. This new model has the same functionality as the original model.

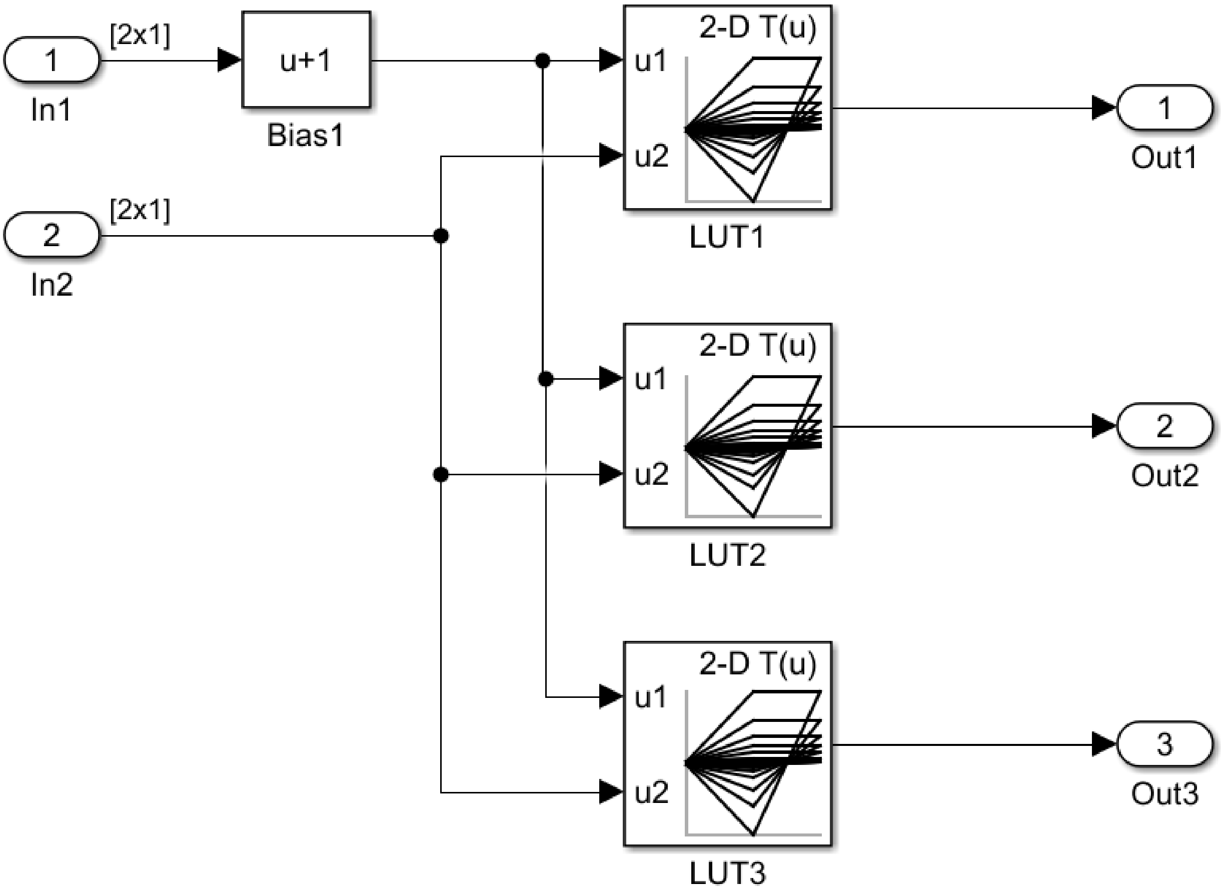
The Model Transformer can replace Lookup Table blocks if:

- 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.
- The data type of the Lookup Table block fractions and breakpoint are double, single, int8, uint8, int16, uint16, int32, or uint32.

The Model Transformer tool works only if all the Lookup Table blocks share all the preceding conditions.

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.

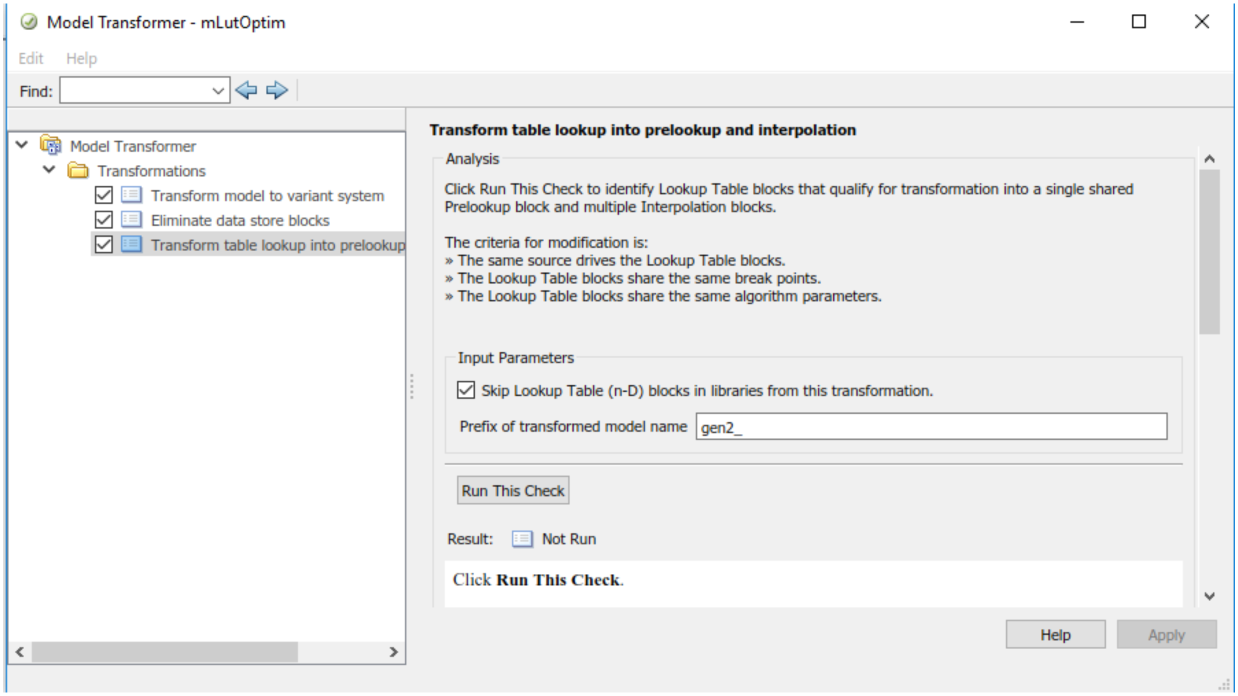
5 Select the **Skip Lookup Table (n-D) blocks in the libraries from this transformation** option to avoid replacing Lookup Table blocks that are linked to a library.

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.



Model Transformer - mLutOptim
Edit
Help
Find:
Model Transformer
Transformations
☑ Transform model to variant system
☑ Eliminate data store blocks
☑ Transform table lookup into prelookup

Transform table lookup into prelookup and interpolation

Analysis

Click Run This Check to identify Lookup Table blocks that qualify for transformation into a single shared Prelookup block and multiple Interpolation blocks.

The criteria for modification is:
» The same source drives the Lookup Table blocks.
» The Lookup Table blocks share the same break points.
» The Lookup Table blocks share the same algorithm parameters.

Input Parameters

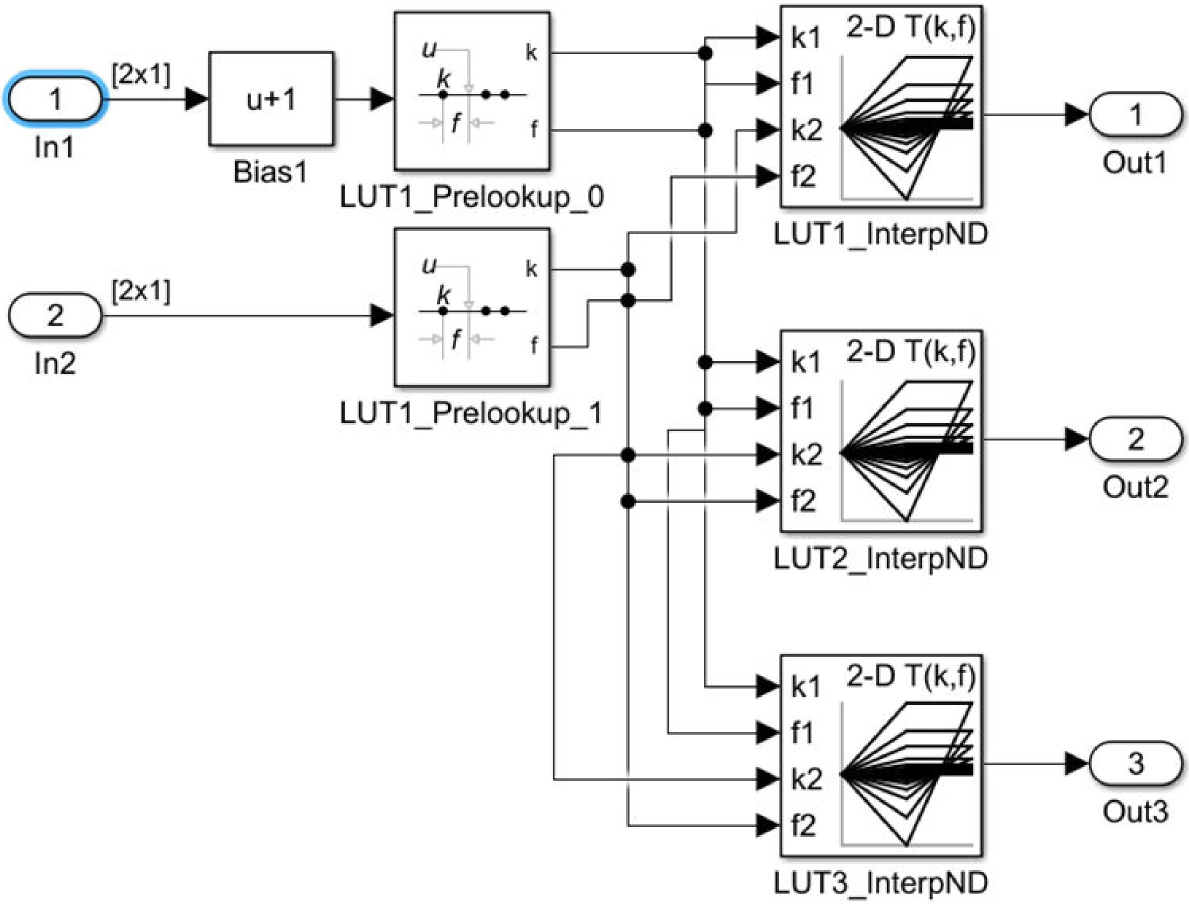
☑ Skip Lookup Table (n-D) blocks in libraries from this transformation.

Prefix of transformed model name: gen2_

Result: Not Run

Click **Run This Check**.

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 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
Check root model Inport block specifications
Identify unconnected lines, input ports, and output ports
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
Identify questionable subsystem settings
Check bus signals treated as vectors
Check for potentially delayed function-call subsystem return values
Check usage of Merge blocks
Check Stateflow data objects with local scope
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

DO-178C/DO-331 Check
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
Display bug reports for Polyspace Code Prover
Display bug reports for Polyspace Bug Finder
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 Test
Display bug reports for Simulink Design Verifier

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 Checks

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, open the Model Advisor and run the checks in **By Task > Modeling Standards for DO-178C/DO-331**.

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
Check for inconsistent vector indexing methods	hisl_0021: Consistent vector indexing method
Check for variant blocks with 'Generate preprocessor conditionals' active	hisl_0023: Verification of model and subsystem variants
Check for root Inports with missing properties	hisl_0024: Inport interface definition
Check for Relational Operator blocks that equate floating-point types	hisl_0017: Usage of blocks that compute relational operators (2)
Check usage of Relational Operator blocks	hisl_0016: Usage of blocks that compute relational operators
Check usage of Logical Operator blocks	hisl_0018: Usage of Logical Operator block
Check usage of While Iterator blocks	hisl_0006: Usage of While Iterator blocks
Check usage of For and While Iterator subsystems	hisl_0007: Usage of For Iterator or While Iterator subsystems
Check usage of For Iterator blocks	hisl_0008: Usage of For Iterator Blocks
Check usage of If blocks and If Action Subsystem blocks	hisl_0010: Usage of If blocks and If Action Subsystem blocks
Check usage Switch Case blocks and Switch Case Action Subsystem blocks	hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks
Check safety-related optimization settings for logic signals	hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)
Check safety-related block reduction optimization settings	hisl_0046: Configuration Parameters > Simulation Target > Block reduction

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

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

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related solver settings for simulation time	hisl_0040: Configuration Parameters > Solver > Simulation time
Check safety-related solver settings for solver options	hisl_0041: Configuration Parameters > Solver > Solver options
Check safety-related solver settings for tasking and sample-time	hisl_0042: Configuration Parameters > Solver > Tasking and sample time options
Check safety-related diagnostic settings for sample time	hisl_0044: Configuration Parameters > Diagnostics > Sample Time
Check safety-related diagnostic settings for parameters	hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters
Check safety-related diagnostic settings for data used for debugging	hisl_0305: Configuration Parameters > Diagnostics > Debugging
Check safety-related diagnostic settings for data store memory	hisl_0013: Usage of data store blocks
Check safety-related diagnostic settings for type conversions	hisl_0309: Configuration Parameters > Diagnostics > Type Conversion
Check safety-related diagnostic settings for signal connectivity	hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals
Check safety-related diagnostic settings for bus connectivity	hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses
Check safety-related diagnostic settings that apply to function-call connectivity	hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls
Check safety-related diagnostic settings for compatibility	hisl_0301: Configuration Parameters > Diagnostics > Compatibility
Check safety-related diagnostic settings for model initialization	hisl_0304: Configuration Parameters > Diagnostics > Model initialization
Check safety-related diagnostic settings for model referencing	hisl_0310: Configuration Parameters > Diagnostics > Model Referencing
Check safety-related diagnostic settings for saving	hisl_0036: Configuration Parameters > Diagnostics > Saving

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

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage of equality operators in MATLAB Function blocks	himl_0009: MATLAB code with equal / not equal relational operators
Check usage of logical operators and functions in MATLAB Function blocks	himl_0010: MATLAB code with logical operators and functions
Check naming of ports in Stateflow charts	hisf_0016: Stateflow port names
Check scoping of Stateflow data objects	hisf_0017: Stateflow data object scoping
Check usage of Gain blocks	hisl_0066: Usage of Gain blocks
Check usage of bitwise operations in Stateflow charts	hisf_0003: Usage of bitwise operations
Check data type of loop control variables	hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance
Check configuration parameters for MISRA C:2012	hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
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

Model Checks for High Integrity Systems Modeling Checks

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, open the Model Advisor and run the checks in **By Task > Modeling Standards for DO-178C/DO-331**.

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
Check for inconsistent vector indexing methods	hisl_0021: Consistent vector indexing method
Check for variant blocks with 'Generate preprocessor conditionals' active	hisl_0023: Verification of model and subsystem variants
Check for root Inports with missing properties	hisl_0024: Inport interface definition
Check for Relational Operator blocks that equate floating-point types	hisl_0017: Usage of blocks that compute relational operators (2)
Check usage of Relational Operator blocks	hisl_0016: Usage of blocks that compute relational operators
Check usage of Logical Operator blocks	hisl_0018: Usage of Logical Operator block
Check usage of While Iterator blocks	hisl_0006: Usage of While Iterator blocks

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage of For and While Iterator subsystems	hisl_0007: Usage of For Iterator or While Iterator subsystems
Check usage of For Iterator blocks	hisl_0008: Usage of For Iterator Blocks
Check usage of If blocks and If Action Subsystem blocks	hisl_0010: Usage of If blocks and If Action Subsystem blocks
Check usage Switch Case blocks and Switch Case Action Subsystem blocks	hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks
Check safety-related optimization settings for logic signals	hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)
Check safety-related block reduction optimization settings	hisl_0046: Configuration Parameters > Simulation Target > Block reduction
Check safety-related optimization settings for application lifespan	hisl_0048: Configuration Parameters > Math and Data Types > Application lifespan (days)
Check safety-related optimization settings for data initialization	hisl_0052: Configuration Parameters > Optimization > Data initialization
Check safety-related optimization settings for data type conversions	hisl_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values
Check safety-related optimization settings for division arithmetic exceptions	hisl_0054: Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions
Check safety-related code generation settings for comments	hisl_0038: Configuration Parameters > Code Generation > Comments
Check safety-related code generation interface settings	hisl_0039: Configuration Parameters > Code Generation > Interface
Check safety-related code generation settings for code style	hisl_0047: Configuration Parameters > Code Generation > Code Style
Check safety-related code generation symbols settings	hisl_0049: Configuration Parameters > Code Generation > Symbols

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

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
Check MATLAB Function metrics	himl_0003: Limitation of MATLAB function complexity
Check MATLAB Code Analyzer messages	himl_0004: MATLAB Code Analyzer recommendations for code generation
Check safety-related model referencing settings	hisl_0037: Configuration Parameters > Model Referencing
Check safety-related diagnostic settings for solvers	hisl_0043: Configuration Parameters > Diagnostics > Solver
Check safety-related solver settings for simulation time	hisl_0040: Configuration Parameters > Solver > Simulation time
Check safety-related solver settings for solver options	hisl_0041: Configuration Parameters > Solver > Solver options
Check safety-related solver settings for tasking and sample-time	hisl_0042: Configuration Parameters > Solver > Tasking and sample time options
Check safety-related diagnostic settings for sample time	hisl_0044: Configuration Parameters > Diagnostics > Sample Time
Check safety-related diagnostic settings for parameters	hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters
Check safety-related diagnostic settings for data used for debugging	hisl_0305: Configuration Parameters > Diagnostics > Debugging
Check safety-related diagnostic settings for data store memory	hisl_0013: Usage of data store blocks
Check safety-related diagnostic settings for type conversions	hisl_0309: Configuration Parameters > Diagnostics > Type Conversion
Check safety-related diagnostic settings for signal connectivity	hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals
Check safety-related diagnostic settings for bus connectivity	hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings that apply to function-call connectivity	hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls
Check safety-related diagnostic settings for compatibility	hisl_0301: Configuration Parameters > Diagnostics > Compatibility
Check safety-related diagnostic settings for model initialization	hisl_0304: Configuration Parameters > Diagnostics > Model initialization
Check safety-related diagnostic settings for model referencing	hisl_0310: Configuration Parameters > Diagnostics > Model Referencing
Check safety-related diagnostic settings for saving	hisl_0036: Configuration Parameters > Diagnostics > Saving
Check safety-related diagnostic settings for Merge blocks	hisl_0303: Configuration Parameters > Diagnostics > Merge block
Check safety-related diagnostic settings for Stateflow	hisl_0311: Configuration Parameters > Diagnostics > Stateflow
Check safety-related optimization settings for Loop unrolling threshold	hisl_0051: Configuration Parameters > Optimization > Loop unrolling threshold
Check model object names	hisl_0032: Model object names
Check for model elements that do not link to requirements	hisl_0070: Placement of requirement links in a model
Check for inappropriate use of transition paths	hisf_0014: Usage of transition paths (passing through states)
Check usage of Bitwise Operator block	hisl_0019: Usage of bitwise operations
Check data types for blocks with index signals	hisl_0022: Data type selection for index signals
Check model file name	hisl_0031: Model file names
Check if/elseif/else patterns in MATLAB Function blocks	himl_0006: MATLAB code if / elseif / else patterns
Check switch statements in MATLAB Function blocks	himl_0007: MATLAB code switch / case / otherwise patterns
Check global variables in graphical functions	hisl_0062: Global variables in graphical functions

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check for length of user-defined object names	hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance
Check usage of Merge blocks	hisl_0015: Usage of Merge blocks
Check usage of conditionally executed subsystems	hisl_0012: Usage of conditionally executed subsystems
Check usage of standardized MATLAB function headers	himl_0001: Usage of standardized MATLAB function headers
Check usage of relational operators in MATLAB Function blocks	himl_0008: MATLAB code relational operator data types
Check usage of equality operators in MATLAB Function blocks	himl_0009: MATLAB code with equal / not equal relational operators
Check usage of logical operators and functions in MATLAB Function blocks	himl_0010: MATLAB code with logical operators and functions
Check naming of ports in Stateflow charts	hisf_0016: Stateflow port names
Check scoping of Stateflow data objects	hisf_0017: Stateflow data object scoping
Check usage of Gain blocks	hisl_0066: Usage of Gain blocks
Check usage of bitwise operations in Stateflow charts	hisf_0003: Usage of bitwise operations
Check data type of loop control variables	hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance
Check configuration parameters for MISRA C:2012	hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
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

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 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 Bug Finder
Display bug reports for Simulink Design Verifier
Display bug reports for Simulink Check

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks
Display bug reports for Simulink Coverage
Display bug reports for Simulink Test

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 Checks

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, open the Model Advisor and run the checks in **By Task > Modeling Standards for DO-178C/DO-331**.

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
Check for inconsistent vector indexing methods	hisl_0021: Consistent vector indexing method
Check for variant blocks with 'Generate preprocessor conditionals' active	hisl_0023: Verification of model and subsystem variants
Check for root Inports with missing properties	hisl_0024: Inport interface definition
Check for Relational Operator blocks that equate floating-point types	hisl_0017: Usage of blocks that compute relational operators (2)

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check usage of Relational Operator blocks	hisl_0016: Usage of blocks that compute relational operators
Check usage of Logical Operator blocks	hisl_0018: Usage of Logical Operator block
Check usage of While Iterator blocks	hisl_0006: Usage of While Iterator blocks
Check usage of For and While Iterator subsystems	hisl_0007: Usage of For Iterator or While Iterator subsystems
Check usage of For Iterator blocks	hisl_0008: Usage of For Iterator Blocks
Check usage of If blocks and If Action Subsystem blocks	hisl_0010: Usage of If blocks and If Action Subsystem blocks
Check usage Switch Case blocks and Switch Case Action Subsystem blocks	hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks
Check safety-related optimization settings for logic signals	hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)
Check safety-related block reduction optimization settings	hisl_0046: Configuration Parameters > Simulation Target > Block reduction
Check safety-related optimization settings for application lifespan	hisl_0048: Configuration Parameters > Math and Data Types > Application lifespan (days)
Check safety-related optimization settings for data initialization	hisl_0052: Configuration Parameters > Optimization > Data initialization
Check safety-related optimization settings for data type conversions	hisl_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values
Check safety-related optimization settings for division arithmetic exceptions	hisl_0054: Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions
Check safety-related code generation settings for comments	hisl_0038: Configuration Parameters > Code Generation > Comments
Check safety-related code generation interface settings	hisl_0039: Configuration Parameters > Code Generation > Interface

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

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check Stateflow charts for unary operators	hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance
Check for Strong Data Typing with Simulink I/O	hisf_0009: Strong data typing (Simulink and Stateflow boundary)
Check for MATLAB Function interfaces with inherited properties	himl_0002: Strong data typing at MATLAB function boundaries
Check MATLAB Function metrics	himl_0003: Limitation of MATLAB function complexity
Check MATLAB Code Analyzer messages	himl_0004: MATLAB Code Analyzer recommendations for code generation
Check safety-related model referencing settings	hisl_0037: Configuration Parameters > Model Referencing
Check safety-related diagnostic settings for solvers	hisl_0043: Configuration Parameters > Diagnostics > Solver
Check safety-related solver settings for simulation time	hisl_0040: Configuration Parameters > Solver > Simulation time
Check safety-related solver settings for solver options	hisl_0041: Configuration Parameters > Solver > Solver options
Check safety-related solver settings for tasking and sample-time	hisl_0042: Configuration Parameters > Solver > Tasking and sample time options
Check safety-related diagnostic settings for sample time	hisl_0044: Configuration Parameters > Diagnostics > Sample Time
Check safety-related diagnostic settings for parameters	hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters
Check safety-related diagnostic settings for data used for debugging	hisl_0305: Configuration Parameters > Diagnostics > Debugging
Check safety-related diagnostic settings for data store memory	hisl_0013: Usage of data store blocks
Check safety-related diagnostic settings for type conversions	hisl_0309: Configuration Parameters > Diagnostics > Type Conversion

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for signal connectivity	hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals
Check safety-related diagnostic settings for bus connectivity	hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses
Check safety-related diagnostic settings that apply to function-call connectivity	hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls
Check safety-related diagnostic settings for compatibility	hisl_0301: Configuration Parameters > Diagnostics > Compatibility
Check safety-related diagnostic settings for model initialization	hisl_0304: Configuration Parameters > Diagnostics > Model initialization
Check safety-related diagnostic settings for model referencing	hisl_0310: Configuration Parameters > Diagnostics > Model Referencing
Check safety-related diagnostic settings for saving	hisl_0036: Configuration Parameters > Diagnostics > Saving
Check safety-related diagnostic settings for Merge blocks	hisl_0303: Configuration Parameters > Diagnostics > Merge block
Check safety-related diagnostic settings for Stateflow	hisl_0311: Configuration Parameters > Diagnostics > Stateflow
Check safety-related optimization settings for Loop unrolling threshold	hisl_0051: Configuration Parameters > Optimization > Loop unrolling threshold
Check model object names	hisl_0032: Model object names
Check for model elements that do not link to requirements	hisl_0070: Placement of requirement links in a model
Check for inappropriate use of transition paths	hisf_0014: Usage of transition paths (passing through states)
Check usage of Bitwise Operator block	hisl_0019: Usage of bitwise operations
Check data types for blocks with index signals	hisl_0022: Data type selection for index signals
Check model file name	hisl_0031: Model file names
Check if/elseif/else patterns in MATLAB Function blocks	himl_0006: MATLAB code if / elseif / else patterns

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check switch statements in MATLAB Function blocks	himl_0007: MATLAB code switch / case / otherwise patterns
Check global variables in graphical functions	hisl_0062: Global variables in graphical functions
Check for length of user-defined object names	hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance
Check usage of Merge blocks	hisl_0015: Usage of Merge blocks
Check usage of conditionally executed subsystems	hisl_0012: Usage of conditionally executed subsystems
Check usage of standardized MATLAB function headers	himl_0001: Usage of standardized MATLAB function headers
Check usage of relational operators in MATLAB Function blocks	himl_0008: MATLAB code relational operator data types
Check usage of equality operators in MATLAB Function blocks	himl_0009: MATLAB code with equal / not equal relational operators
Check usage of logical operators and functions in MATLAB Function blocks	himl_0010: MATLAB code with logical operators and functions
Check naming of ports in Stateflow charts	hisf_0016: Stateflow port names
Check scoping of Stateflow data objects	hisf_0017: Stateflow data object scoping
Check usage of Gain blocks	hisl_0066: Usage of Gain blocks
Check usage of bitwise operations in Stateflow charts	hisf_0003: Usage of bitwise operations
Check data type of loop control variables	hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance
Check configuration parameters for MISRA C:2012	hisl_0060: Configuration parameters that improve MISRA C:2012 compliance

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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

Model Checks for MathWorks Automotive Advisory Board (MAAB) Guideline Compliance

You can check that your model or subsystem complies with MathWorks Automotive Advisory Board (MAAB) Guidelines by running the Model Advisor. Navigate to **By Task > Modeling Standards for MAAB** and run the checks.

The MAAB involves major automotive OEMs and suppliers in the process of evolving MathWorks controls, simulation, and code generation products, including Simulink, Stateflow, and Simulink Coder™. An important result of this collaboration has been the MAAB Control Algorithm Modeling Guidelines.

For MAAB checks, you can control whether the Model Advisor looks under masks or follows links. See “Set MAAB and JMAAB Checks to Look Under Masks or Follow Links” on page 3-88.

The table lists the MAAB checks with the applicable MAAB Control Algorithm Modeling guideline. For JMAAB checks, see “Model Checks for Japan MATLAB Automotive Advisory Board (JMAAB) Guideline Compliance” on page 3-74.

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
Naming Conventions	Check file names	ar_0001: Filenames
	Check folder names	ar_0002: Directory names
	Check subsystem names	jc_0201: Usable characters for Subsystem names
	Check port block names	jc_0211: Usable characters for Inport blocks and Outport blocks
	Check character usage in signal labels	jc_0221: Usable characters for signal line names
	Check character usage in block names	jc_0231: Usable characters for block names

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check Simulink bus signal names	na_0030: Usable characters for Simulink Bus names
Model Architecture	Check for mixing basic blocks and subsystems	db_0143: Similar block types on the model levels
	Check unused ports in Variant Subsystems	na_0020: Number of inputs to variant subsystems
	Check use of default variants	na_0036: Default variant
	Check use of single variable variant conditionals	na_0037: Use of single variable variant conditionals
Model Configuration Options	Check Implement logic signals as Boolean data (vs. double)	jc_0011: Optimization parameters for Boolean data types
	Check model diagnostic parameters	jc_0021: Model diagnostic settings
Simulink	Check for Simulink diagrams using nonstandard display attributes	na_0004: Simulink model appearance
	Check font formatting	db_0043: Simulink font and font size
	Check positioning and configuration of ports	db_0042: Port block in Simulink models
	Check visibility of block port names	na_0005: Port block name visibility in Simulink models
	Check display for port blocks	jc_0081: Icon display for Port block
	Check whether block names appear below blocks	db_0142: Position of block names
	Check the display attributes of block names	jc_0061: Display of block names
	Check position of Trigger and Enable blocks	db_0146: Triggered, enabled, conditional Subsystems

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check for nondefault block attributes	db_0140: Display of basic block parameters
	Check for matching port and signal names	jm_0010: Port block names in Simulink models
	Check signal line labels	na_0008: Display of labels on signals
	Check for propagated signal labels	na_0009: Entry versus propagation of signal labels
	Check for unconnected ports and signal lines	db_0081: Unconnected signals, block inputs and block outputs
	Check for prohibited blocks in discrete controllers	jm_0001: Prohibited Simulink standard blocks inside controllers
	Check for prohibited sink blocks	hd_0001: Prohibited Simulink sinks
	Check scope of From and Goto blocks	na_0011: Scope of Goto and From blocks
	Check usage of Switch blocks	jc_0141: Use of the Switch block
	Check usage of Relational Operator blocks	jc_0131: Use of Relational Operator block
	Check for indexing in blocks	db_0112: Indexing
	Check usage of buses and Mux blocks	na_0010: Grouping data flows into signals
	Check usage of tunable parameters in blocks	db_0110: Tunable parameters in basic blocks
	Check orientation of Subsystem blocks	jc_0111: Direction of Subsystem
	Check fundamental logical and numerical operations	na_0002: Appropriate implementation of fundamental logical and numerical operations

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check usage of merge blocks	na_0032: Use of merge blocks
	Check logical expressions in 'If' blocks	na_0003: Simple logical expressions in If Condition block
	Check usage of Goto and From blocks between Subsystems	jc_0171: Maintaining signal flow when using Goto and From blocks
	Check usage of enumerated values	na_0031: Definition of default enumerated value
	Check Simulink signal appearance	db_0032: Simulink signal appearance
	Check usage of non-compliant blocks	na_0027: Use of only standard library blocks
Stateflow	Check usage of exclusive and default states in state machines	db_0137: States in state machines
	Check transition orientations in flow charts	db_0132: Transitions in flow charts
	Check entry formatting in State blocks in Stateflow charts	jc_0501: Format of entries in a State block
	Check return value assignments in Stateflow graphical functions	jc_0511: Setting the return value from a graphical function
	Check default transition placement in Stateflow charts	jc_0531: Placement of the default transition
	Check for Strong Data Typing with Simulink I/O	db_0122: Stateflow and Simulink interface signals and parameters
	Check Stateflow data objects with local scope	db_0125: Scope of internal signals and local auxiliary variables
	Check usage of return values from Stateflow graphical functions	jc_0521: Use of the return value from graphical functions
	Check for MATLAB expressions in Stateflow charts	db_0127: MATLAB commands in Stateflow

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check for pointers in Stateflow charts	jm_0011: Pointers in Stateflow
	Check for event broadcasts in Stateflow charts	jm_0012: Event broadcasts
	Check transition actions in Stateflow charts	db_0151: State machine patterns for transition actions
	Check for bitwise operations in Stateflow charts	na_0001: Bitwise Stateflow operators
	Check usage of unary minus operations in Stateflow charts	jc_0451: Use of unary minus on unsigned integers in Stateflow
	Check for comparison operations in Stateflow charts	na_0013: Comparison operation in Stateflow
	Check usage of floating-point expressions in Stateflow charts	jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow
	Check for names of Stateflow ports and associated signals	db_0123: Stateflow port names
	Check nested states in Stateflow charts	na_0038: Levels in Stateflow charts
	Check use of Simulink in Stateflow charts	na_0039: Use of Simulink in Stateflow charts
	Check number of Stateflow states per container	na_0040: Number of states per container
	Check reuse of Variables within a Stateflow scope	jc_0491: Reuse of variables within a single Stateflow scope
	Check for Stateflow transition appearance	db_0129: Stateflow transition appearance

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
MATLAB Functions and Code	Check input and output settings of MATLAB Functions	na_0034: MATLAB Function block input/output settings
	Check MATLAB Function metrics	• na_0016: Source lines of MATLAB Functions • na_0018: Number of nested if/else and case statement
	Check MATLAB code for global variables	na_0024: Global Variables
	Check the number of function calls in MATLAB Function blocks	na_0017: Number of called function levels
	Check usage of restricted variable names	na_0019: Restricted Variable Names
	Check usage of character vector inside MATLAB Function block	na_0021: Strings
	Check usage of recommended patterns for Switch/Case statements	na_0022: Recommended patterns for Switch/Case statements

See Also

Related Examples

- Select and Run Model Advisor Checks (Simulink)

Model Checks for Japan MATLAB Automotive Advisory Board (JMAAB) Guideline Compliance

You can check that your model or subsystem complies with Japan MATLAB Automotive Advisory Board (JMAAB) guidelines by running the Model Advisor. Navigate to **By Task > Modeling Standards for JMAAB** and run the checks.

The JMAAB involves major automotive OEMs and suppliers in the process of evolving MathWorks controls, simulation, and code generation products, including Simulink, Stateflow, and Simulink Coder. An important result of this collaboration has been the Control Algorithm Modeling Guidelines (JMAAB), Version 5.1.

For JMAAB checks, you can control whether the Model Advisor looks under masks or follows links. See “Set MAAB and JMAAB Checks to Look Under Masks or Follow Links” on page 3-88.

The table lists the JMAAB checks with the applicable JMAAB Control Algorithm Modeling guideline.

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
Naming Conventions	Check file names	ar_0001: Usable characters for file names
	Check folder names	ar_0002: Usable characters for folder names
	Check subsystem names	jc_0201: Usable characters for Subsystem names

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check port block names	jc_0211: Usable characters for Inport block and Outport block
	Check character usage in block names	jc_0231: Usable characters for block names
	Check usable characters for signal names and bus names	jc_0222:
	Check usable characters for parameter names	jc_0232: Usable characters for parameter names
	Check length of model file name	jc_0241: Length restrictions for model file names
	Check length of folder name at every level of model path	jc_0242: Length restrictions for folder names
	Check length of subsystem names	jc_0243: Length restrictions for subsystem names
	Check length of Inport and Outport names	jc_0244: Length restrictions for Inport and Outport names

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check length of signal and bus names	jc_0245: Length restrictions for signal and bus names
	Check length of parameter names	jc_0246: Length restrictions for parameter names
	Check length of block names	jc_0247: Length restrictions for block names
Model Architecture	Check for mixing basic blocks and subsystems	db_0143: Usage block types in model hierarchy
Model Configuration Options	Check Implement logic signals as Boolean data (vs. double)	jc_0011: Optimization parameters for Boolean data types
Simulink	Check for Simulink diagrams using nonstandard display attributes	na_0004: Simulink model appearance settings
	Check font formatting	db_0043: Model font and font size

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check positioning and configuration of ports	db_0042: Usage of Inport and Outport blocks
	Check whether block names appear below blocks	db_0142: Position of block names
	Check the display attributes of block names	jc_0061: Display of block names
	Check position of Trigger and Enable blocks	db_0146: Block layout in conditional subsystems
	Check for nondefault block attributes	db_0140: Display of block parameters
	Check for unconnected ports and signal lines	db_0081: Unconnected signals / block
	Check usage of Switch blocks	jc_0141: Use of the Switch block
	Check usage of Relational Operator blocks	jc_0131: Usage of Relational Operators
	Check for indexing in blocks	db_0112: Usage of index

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check usage of tunable parameters in blocks	db_0110: Guidelines for block parameters
	Check signal line labels (JMAAB)	jc_0008: Definition of Signal labels
	Check for propagated signal labels	jc_0009: Signal name propagation
	Check usage of Discrete-Time Integrator block	jc_0627: Guideline for using the Discrete-Time Integrator block
	Check settings for data ports in Multiport Switch blocks	jc_0630: Usage of Multiport Switch block
	Check usage of fixed-point data type with non-zero bias	jc_0643: Fixed-point setting
	Check input and output datatype for Switch blocks	jc_0650: Block input/output data type with switching function

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check signs of input signals in product blocks	jc_0611: Input signal for multiplication and division blocks
	Check Signed Integer Division Rounding mode	jc_0642: Integer rounding mode setting
	Check type setting by data objects	jc_0644: Guideline for type setting
	Check usage of the Saturation blocks	jc_0628: Guideline for using the Saturation Block
	Check usage of Merge block	jc_0659: Usage restrictions of signal lines inputted to Merge block
	Check usage of Memory and Unit Delay blocks	jc_0623: Use of continuous-time delay blocks and discrete-time blocks
	Check block orientation	jc_0110: Direction of block

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check if blocks are shaded in the model	jc_0604: Block shading
	Check operator order of Product blocks	jc_0610: Operator order for multiplication and division blocks
	Check icon shape of Logical Operator blocks	jc_0621: Guideline for using the Logical Operator block
	Check if tunable block parameters are defined as named constants	jc_0645: Parameter definition for calibration
	Check default/else case in Switch Case blocks and If blocks	jc_0656: Usage of Conditional Control block
	Check usage of Lookup Tables	jc_0626: Guideline for using the Lookup Table block
	Check for parentheses in Fcn block expressions	jc_0622: Guideline for using the Fcn block

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check undefined initial output for conditional subsystems	jc_0640: Initial value settings for Outport blocks in conditional subsystems
	Check for use of the Sum block	jc_0121: Usage of add/ subtraction blocks
	Check for avoiding algebraic loops between subsystems	jc_0653: Delay block layout in feedback loops
	Comparing floating point types in Simulink	jc_0800: Comparing floating-point types in Simulink
	Check duplication of Simulink data names	jc_0791: Duplicate data names
Stateflow	Check transition orientations in flow charts	db_0132: Transitions in Flow Charts
	Check return value assignments in Stateflow graphical functions	jc_0511: Setting the return value from a graphical function

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check default transition placement in Stateflow charts (JMAAB)	jc_0531: Placement of the default transition.
	Check for Strong Data Typing with Simulink I/O	db_0122: Stateflow and Simulink interface signals and parameters
	Check Stateflow data objects with local scope	db_0125: Stateflow local data
	Check usage of return values from Stateflow graphical functions	Not applicable
	Check for pointers in Stateflow charts	jm_0011: Pointers in Stateflow
	Check for event broadcasts in Stateflow charts	jm_0012: Usage restrictions of events and broadcasting events
	Check for bitwise operations in Stateflow charts	na_0001: Standard usage of Stateflow operators

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check for unary minus operations on unsigned integers in Stateflow charts	jc_0451: Use of unary minus on unsigned integers
	Check comments in state actions	jc_0738: Usage of Stateflow comments
	Check prohibited comparison operation of logical type signals	jc_0655: Prohibition of logical value comparison in Stateflow
	Check usage of internal transitions in Stateflow states	jc_0763: Usage of multiple internal transitions
	Check usage of transition conditions in Stateflow transitions	jc_0772: Execution order and transition conditions of transition lines
	Check uniqueness of Stateflow State and Data names	jc_0732: Distinction between state names, data names, and event names

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check uniqueness of State names	jc_0730: Unique state name in Stateflow blocks
	Check usage of parentheses in Stateflow transitions	jc_0752: Format of condition action in transition label
	Check prohibited combination of state action and flow chart	jc_0762: Prohibited of state action and flow chart combination
	Check condition actions and transition actions in Stateflow	jc_0753: Condition actions and transition actions in Stateflow
	Check first index of arrays in Stateflow	jc_0701: Usable number for first index
	Check usage of State names	jc_0731: State name format
	Check execution timing for default transition path	jc_0712: Execution timing for default transition path

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check repetition of Action types	jc_0734: Number of state action types
	Check for unused data in Stateflow Charts	jc_0700: Unused data in Stateflow block
	Check updates to variables used in state transition conditions	jc_0741: Timing to update data used in state chart transition conditions
	Check starting point of internal transition in Stateflow	jc_0760: Starting point of internal transition
	Check for parallel Stateflow state used for grouping	jc_0721: Usage of parallel states
	Check scope of data in parallel states	jc_0722: Local data definition in parallel states
	Check indentation of Stateflow blocks	jc_0736: Uniform indentations in Stateflow blocks

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
	Check for unexpected backtracking in state transitions	jc_0751 : Backtracking prevention in state transition
	Check for usage of text inside states	jc_0739: Guidelines for describing texts inside states
	Check for unconnected objects in Stateflow Charts	jc_0797: Unconnected transition lines / states / connective junctions
	Check placement of Label String in Transitions	jc_0770: Transition label layout
	Check Stateflow chart action language	jc_0790: Chart action language
	Check usable characters for Stateflow data names	jc_0795: Usable characters for Stateflow data names
	Check length of Stateflow data name	jc_0796: Length restriction for Stateflow data names

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 5.1
MATLAB Functions	Check input and output settings of MATLAB Functions	na_0034: MATLAB Function block input/output settings
	Check MATLAB code for global variables	na_0024: Shared data in MATLAB functions

See Also

Related Examples

- Select and Run Model Advisor Checks (Simulink)

Set MAAB and JMAAB Checks to Look Under Masks or Follow Links

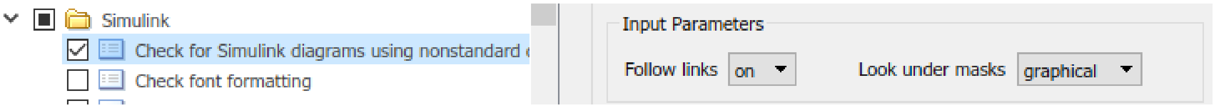
Control Whether Checks Look Under Masks or Follow Links

You can define whether MAAB and JMAAB checks look under masks or follow links. These checks are available in the Model Advisor at:

- **By Task** > **Modeling Standards for MAAB** > **Simulink**
- **By Task** > **Modeling Standards for JMAAB** > **Simulink**

To customize a check to look under masks or follow links for an open model:

1. Navigate to **Model Advisor** > **Settings** > **Open Configuration Editor**.
2. Select **By Task** > **Model Standards for MAAB** > **Simulink** > **Check for Simulink diagrams using nonstandard display attributes**.
3. On the right pane, under **Input Parameters**, you can specify to either **Follow links** or **Look under masks**



Follow links: on

Look under masks: graphical

4. For the **Follow links** or **Look under masks** settings to take effect, you must save the configuration file (**File** > **Save As**), close the Configuration Editor, and load the saved configuration file in the Model Advisor from **Settings** > **Load Configuration**.

For more information on options for each parameter, see `find_system`.

See Also

`find_system`

More About

- “Select and Run Model Advisor Checks” (Simulink)

Model Checks for MISRA C:2012 Compliance

You can check that your model or subsystem has a likelihood of generating MISRA C:2012 compliant code. Navigate to **By Task > Modeling Guidelines for MISRA C:2012** and run the checks:

- Check usage of Assignment blocks
- Check for blocks not recommended for MISRA C:2012
- Check for unsupported block names
- Check configuration parameters for MISRA C:2012
- Check for equality and inequality operations on floating-point values
- Check for bitwise operations on signed integers
- Check for recursive function calls
- Check for switch case expressions without a default case
- Check for blocks not recommended for C/C++ production code deployment
- 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

- “Select and Run Model Advisor Checks” (Simulink)

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

You can check that your code complies with the CERT C, CWE, and ISO/IEC TS 17961 (Embedded Coder) secure coding standards. Navigate to **By Task** > **Modeling Guidelines for secure coding standards (CERT C, CWE, ISO/IEC TS 17961)** and run the checks:

- 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 Violation of Specified Minimum and Maximum Values

See Also

Related Examples

- “Select and Run Model Advisor Checks” (Simulink)

Model Checks for Requirements Links

To check that every requirements link in your model has a valid target in a requirements document, navigate to **Analysis** > **Requirements Traceability** > **Check Consistency** and run the Model Advisor checks:

- Identify requirement links with missing documents
- Identify requirement links that specify invalid locations within documents
- Identify selection-based links having descriptions that do not match their requirements document text
- Identify requirement links with path type inconsistent with preferences

To execute these checks from the Model Advisor, navigate to **By Product** > **Simulink Requirements** > **Requirements Consistency**.

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)
- "Select and Run Model Advisor Checks" (Simulink)
- "High-Integrity System Modeling" (Simulink)

Generate Model Advisor Reports in Adobe PDF and Microsoft Word Formats

By default, when the Model Advisor runs checks, it generates an HTML report of check results in the `slprj/modeladvisor/`*`model_name`* folder. On Windows® platforms, you can generate Model Advisor reports in Adobe® PDF and Microsoft Word .docx formats.

The beginning of the PDF and Microsoft Word versions of the Model Advisor reports contain the:

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

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

1. In the Model Advisor window, navigate to the folder that contains the checks that you ran.
2. Select the folder. The right pane of the Model Advisor window displays information about that folder. The pane includes a **Report** box.
3. In the Report box, click **Generate Report**.
4. In the Generate Model Advisor Report dialog box, enter the path to the folder where you want to generate the report. Provide a file name.
5. In the Generate Model Advisor Report dialog box **File format** field, select `PDF` or `Word`.
6. Click **OK**. The Model Advisor generates the report in PDF or Microsoft Word format to the location that you specified.

Modify Default Template

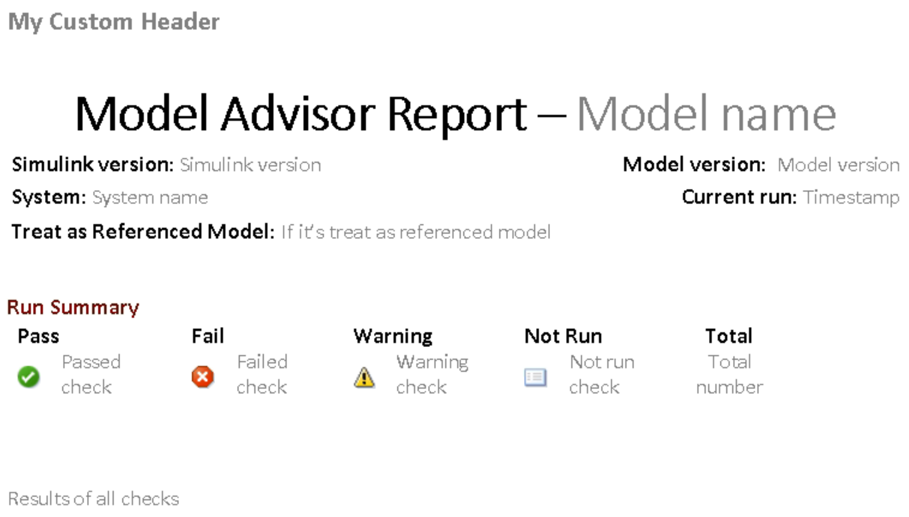
If you have a MATLAB Report Generator license, you can modify the default template that the Model Advisor uses to generate the report in PDF or Microsoft Word.

The default template contains holes that the Model Advisor uses to populate the generated report with information about the analysis. If you want your Model Advisor report to contain the analysis information, do not delete the holes. When the Model Advisor uses the template to generate the report, analysis information overrides the text that you enter in the template hole field.

Template Hole	**In generated report, displays**
`ModelName`	Model name
`SimulinkVersion`	Simulink version
`SystemName`	System name
`TreatAsMdlRef`	Whether or not model is treated as a referenced model
`ModelVersion`	Model version
`CurrentRun`	Model Advisor analysis time stamp
`PassCount`	Number of checks that pass
`FailCount`	Number of checks that fail
`WarningCount`	Number of checks that cause a warning
`NrunCount`	Number of checks that did not run
`TotalCount`	Total number of checks
`CheckResults`	Results for each check

This example shows how to add a header to a PDF version of a Model Advisor report.

1. Using Microsoft Word, open the default template `*matlabroot*/toolbox/simulink/simulink/modeladvisor/resources/templates/default.dotx`.
2. Rename and save the template `default.dotx` to a writable location. For example, save template `default.dotx` to `C:/work/ma_format/mytemplate.dotx`.
3. In the template `C:/work/ma_format/mytemplate.dotx` file, add a header. For example, in the template header, add the text **`My Custom Header`**. Save the template as a Microsoft Word .dotx file.



My Custom Header

Model Advisor Report – Model name

Simulink version: Simulink version **Model version:** Model version

System: System name **Current run:** Timestamp

Treat as Referenced Model: If it's treat as referenced model

Run Summary

Pass	Fail	Warning	Not Run	Total
✓ Passed check	✗ Failed check	⚠ Warning check	Not run check	Total number

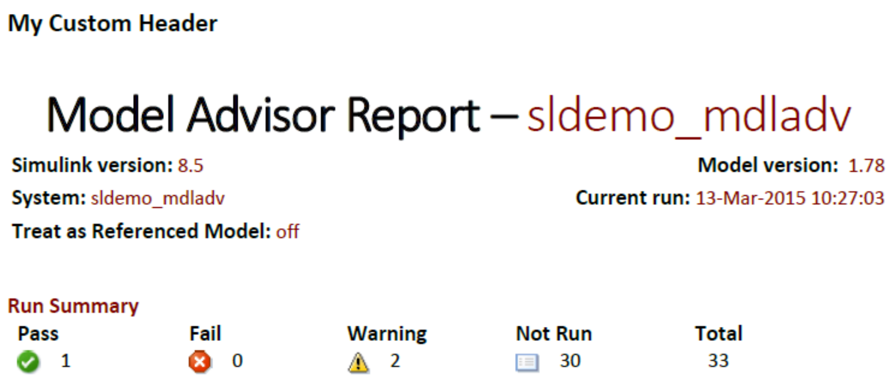
Results of all checks

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

5 In the Generate Model Advisor Report dialog box:

- Enter the path to the folder where you want to generate the report and provide a file name.
- Set **File format** to `PDF`.
- Select **View report after generation**.
- Set **Report template** to `C:\work\ma_format\mytemplate.dotx`.

6 Click **OK**. The Model Advisor generates the report in PDF format with a custom header. Because the template `mytemplate.dotx` contains holes that Model Advisor uses to populate the generated report, the report contains information about the Model Advisor analysis. For example, the report contains the model name, model version, and number of checks that pass.



My Custom Header

Model Advisor Report – sldemo_mdladv

Simulink version: 8.5

Model version: 1.78

System: sldemo_mdladv

Current run: 13-Mar-2015 10:27:03

Treat as Referenced Model: off

Run Summary

Pass	Fail	Warning	Not Run	Total
✓ 1	✗ 0	⚠ 2	▤ 30	33

See Also

Related Examples

- “Save and View Model Advisor Reports” (Simulink)
- “Customize Microsoft Word Component Templates” (MATLAB Report Generator)
- “Select and Run Model Advisor Checks” (Simulink)

4

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 4-10
- “Find Check IDs” on page 4-3

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5

Find Check IDs

An *ID* is a unique identifier for a Model Advisor check. You find check IDs in the Model Advisor, using check context menus.

To Find	Do This
Check Title, ID, or location of the MATLAB source code	**1** On the model window toolbar, select **Settings** > **Preferences**. **2** In the Model Advisor Preferences dialog box, select **Show Source Tab**. **3** In the right pane of the Model Advisor window, click the **Source** tab. The Model Advisor window displays the check Title, TitleId, and location of the MATLAB source code for the check.
Check ID	**1** In the left pane of the Model Advisor, select the check. **2** Right-click the check name and select **Send Check ID to Workspace**. The ID is displayed in the Command Window and sent to the base workspace.
Check IDs for selected checks in a folder	**1** In the left pane of the Model Advisor, select the checks for which you want IDs. Clear the other checks in the folder. **2** Right-click the folder and select **Send Check ID to Workspace**. An array of the selected check IDs are sent to the base workspace.

If you know a check ID from a previous release, you can find the current check ID using the `ModelAdvisor.lookupCheckID` function. For example, the check ID for **By Product** > **Simulink Check** > **Modeling Standards** > **DO-178C/DO-331 Checks** > **Check safety-related optimization settings** prior to Release 2010b was `DO178B:OptionSet`. Using the `ModelAdvisor.lookupCheckID` function returns:

```
>> NewID = ModelAdvisor.lookupCheckID('DO178B:OptionSet')

NewID =

mathworks.do178.OptionSet
```

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

See Also

`ModelAdvisor.lookupCheckID`

Create a Function for Checking Multiple Systems

The following tutorial guides you through creating and testing a function to run multiple checks on any model. The function returns the number of failures and warnings.

1. In the MATLAB window, select **New** > **Function**.
2. Save the function as `run_configuration.m`.
3. In the MATLAB Editor, specify `[output_args]` as `[fail, warn]`.
4. Rename the function `run_configuration`.
5. Specify `input_args` to `SysList`.
6. Inside the function, specify the list of checks to run using the example Model Advisor configuration file:

   ```
   fileName = 'slvnvdemo_mdladv_config.mat';
   ```

7. Call the `ModelAdvisor.run` function:

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

8. Determine the number of checks that return warnings and failures:

   ```
   fail=0;
   warn=0;

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

   The function should now look like this:

   ```
   function [fail, warn] = run_configuration(SysList)
   %RUN_CONFIGURATION Check systems with Model Advisor
   %   Check systems given as input and return number of warnings and
   %   failures.

   fileName = 'slvnvdemo_mdladv_config.mat';
   fail=0;
   warn=0;

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

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

   end
   ```

9 Save the function.

10 Test the function. In the MATLAB Command Window, run `run_configuration.m` on the `sldemo_auto_climatecontrol/Heater Control` subsystem:

```
[failures, warnings] = run_configuration(...
   'sldemo_auto_climatecontrol/Heater Control');
```

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

See Also

`ModelAdvisor.run`

Related Examples

- “Check Multiple Systems in Parallel” on page 4-7
- “Create a Function for Checking Multiple Systems in Parallel” on page 4-8

Check Multiple Systems in Parallel

Checking multiple systems in parallel reduces the processing time required by the Model Advisor to check multiple systems. If you have 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.

To enable parallel processing, use the `ModelAdvisor.run` function with `'ParallelMode'` set to `'On'`. By default, `'ParallelMode'` is set to `'Off'`. When you use `ModelAdvisor.run` with `'ParallelMode'` set to `'On'`, MATLAB creates a parallel pool.

See Also

`ModelAdvisor.run`

Related Examples

- “Create a Function for Checking Multiple Systems in Parallel” on page 4-8

Create a Function for Checking Multiple Systems in Parallel

If you have a Parallel Computing Toolbox license and a multicore host machine, you can create the following function to check multiple systems in parallel:

1. Create the `run_configuration` function.
2. Save the function as `run_fast_configuration.m`.
3. In the Editor, change the name of the function to `run_fast_configuration`.
4. In the `ModelAdvisor.run` function, set `'ParallelMode'` to `'On'` . When you use `ModelAdvisor.run` with `'ParallelMode'` set to `'On'`, MATLAB automatically creates a parallel pool.

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

   The function should now look like this:

   ```
   function [fail, warn] = run_fast_configuration(SysList)
   %RUN_FAST_CONFIGURATION Check systems in parallel with Model Advisor
   %   Return number of warnings and failures.
   fileName = '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
   ```
5. Save the function.
6. Test the function. In the MATLAB Command Window, create a list of systems:

   ```
   SysList={'sldemo_auto_climatecontrol/Heater Control',...
       'sldemo_auto_climatecontrol/AC Control','rtwdemo_iec61508'};
   ```
7. Run `run_fast_configuration` on the list of systems:

   ```
   [failures, warnings] = run_fast_configuration(SysList);
   ```
8. Review the results. Click the Summary Report link to open the Model Advisor Command-Line Summary report.

See Also

`ModelAdvisor.run`

Related Examples

- “Check Multiple Systems in Parallel” on page 4-7

Archive and View Results

Archive Results

After you run the Model Advisor programmatically, you can archive the results. The `ModelAdvisor.run` function returns a cell array of `ModelAdvisor.SystemResult` objects, one for each system run. If you save the objects, you can use them to view the results at a later time without rerunning the Model Advisor.

View Results in Command Window

When you run the Model Advisor programmatically, the system-level results of the run are displayed in the Command Window. For example:

```
Systems passed: 0 of 1
Systems with warnings: 1 of 1
Systems failed: 0 of 1
Summary Report
```

The Summary Report link provides access to the Model Advisor Command-Line Summary report.

You can review additional results in the Command Window by calling the `DisplayResults` parameter when you run the Model Advisor. For example, run the Model Advisor as follows:

```
SysResultObjArray = ModelAdvisor.run('sldemo_auto_climatecontrol/Heater Control',...
    'Configuration','slvnvdemo_mdladv_config.mat','DisplayResults','Details');
```

The results displayed in the Command Window are:

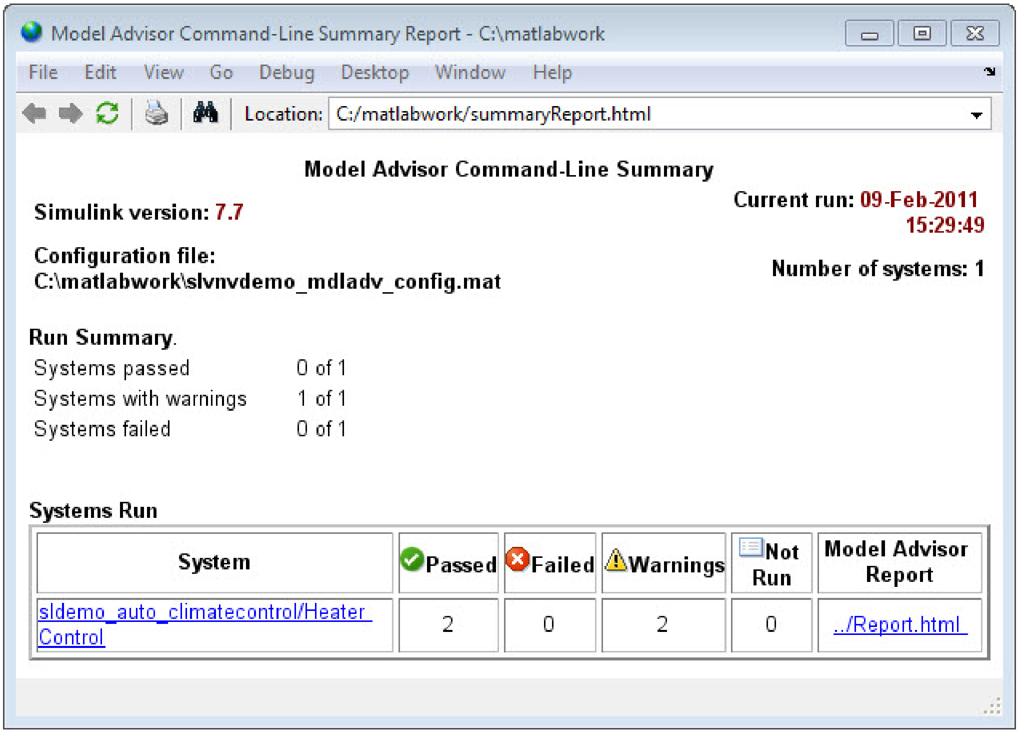
```
   Running Model Advisor
         Running Model Advisor on sldemo_auto_climatecontrol/Heater Control
         ============================================================
         Model Advisor run: 29-Oct-2012 16:30:00
         Configuration: slvnvdemo_mdladv_config.mat
         System: sldemo_auto_climatecontrol/Heater Control
         System version: 8.1
         Created by: The MathWorks Inc.
         ============================================================
         (1) Warning: Check model diagnostic parameters [check ID: mathworks.maab.jc_0021]
         ------------------------------------------------------------
         (2) Warning: Check for fully defined interface [check ID: mathworks.iec61508.RootLevelInports]
         ------------------------------------------------------------
         (3) Pass: Check for unconnected objects [check ID: mathworks.iec61508.UnconnectedObjects]
         ------------------------------------------------------------
         (4) Pass: Check for blocks not recommended for C/C++ production code deployment
[check ID: mathworks.iec61508.PCGSupport]
         ------------------------------------------------------------
```

```
Summary:    Pass    Warning    Fail    Not Run
             2        2          0         0
============================================================


Systems passed: 0 of 1

Systems with warnings: 1 of 1

Systems failed: 0 of 1
Summary Report
```

To display the results in the Command Window after loading an object, use the `viewReport` function.

View Results in Model Advisor Command-Line Summary Report

When you run the Model Advisor programmatically, a Summary Report link is displayed in the Command Window. Clicking this link opens the Model Advisor Command-Line Summary report. The following graphic is the report that the Model Advisor generates for `run_configuration`.


Model Advisor Command-Line Summary Report - C:\matlabwork
File Edit View Go Debug Desktop Window Help

Location: C:/matlabwork/summaryReport.html

Model Advisor Command-Line Summary

Simulink version: 7.7

Current run: 09-Feb-2011 15:29:49

Configuration file: C:\matlabwork\slvnvdemo_mdladv_config.mat

Number of systems: 1

Run Summary

Systems passed	0 of 1
Systems with warnings	1 of 1
Systems failed	0 of 1

Systems Run

System	✓Passed	✗Failed	⚠Warnings	Not Run	Model Advisor Report
sldemo_auto_climatecontrol/Heater Control	2	0	2	0	../Report.html

To view the Model Advisor Command-Line Summary report after loading an object, use the `summaryReport` function.

View Results in Model Advisor GUI

In the Model Advisor window, you can view the results of running the Model Advisor programmatically using the `viewReport` function. In the Model Advisor window, you can review results, run checks, fix warnings and failures, and view and save Model Advisor reports.

Tip To fix warnings and failures, you must rerun the check in the Model Advisor window.

View Model Advisor Report

For a single system or check, you can view the same Model Advisor report that you access from the Model Advisor GUI.

To view the Model Advisor report for a system:

- Open the Model Advisor Command-Line Summary report. In the Systems Run table, click the link for the Model Advisor report.
- Use the `viewReport` function.

To view individual check results:

- In the Command Window, generate a detailed report using the `viewReport` function with the `DisplayResults` parameter set to `Details`, and then click the Pass, Warning, or Fail link for the check. The Model Advisor report for the check opens.
- Use the `view` function.

See Also

`ModelAdvisor.run` | `ModelAdvisor.summaryReport` | `view` | `viewReport`

Related Examples

- “Archive and View Model Advisor Run Results” on page 4-14
- “Check Multiple Systems in Parallel” on page 4-7
- “Create a Function for Checking Multiple Systems in Parallel” on page 4-8

More About

- “Run Model Checks” (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 4-10

5

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:

- From a model editor window, select **Analysis > 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**.

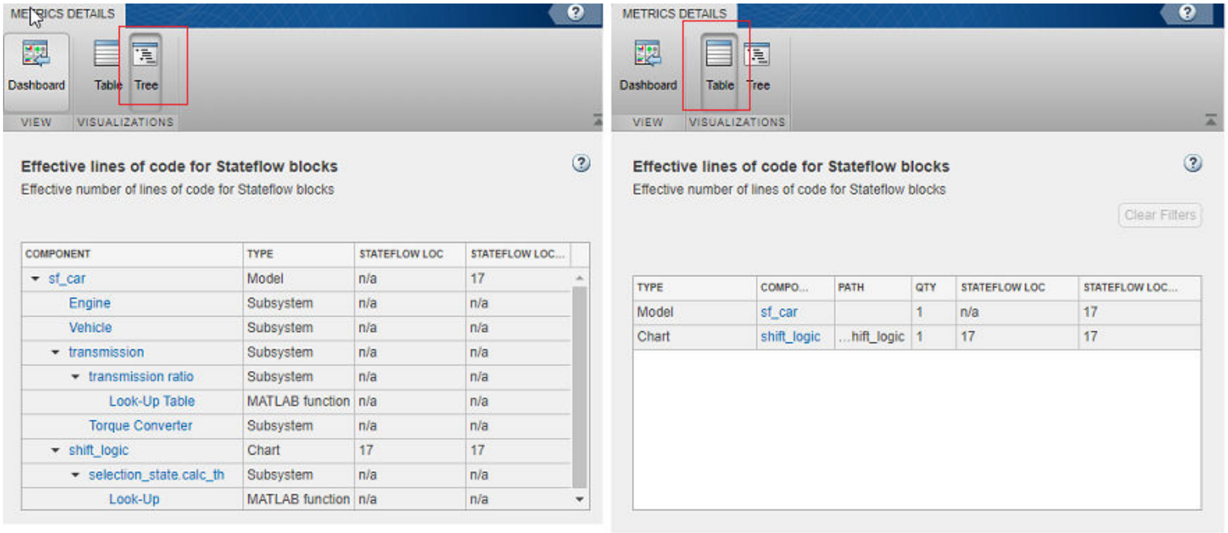
The Metrics Dashboard provides the system name and a data collection timestamp. If there were issues during data collection, click the alert icon to see warnings.

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.

- For all widgets, you can view results in either a **Tree** or **Table** view. For the **High Integrity** and **MAAB** compliance widgets, you can also choose a **Grid** view. To view highlighted results, in the grid view, click the appropriate cell.
- To sort the results by value or aggregated value, click the corresponding value column header.
- For all metrics except the **High Integrity** and **MAAB** 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 no available results 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.



Effective lines of code for Stateflow blocks

Effective number of lines of code for Stateflow blocks

COMPONENT	TYPE	STATEFLOW LOC	STATEFLOW LOC...
sf_car	Model	n/a	17
Engine	Subsystem	n/a	n/a
Vehicle	Subsystem	n/a	n/a
transmission	Subsystem	n/a	n/a
transmission ratio	Subsystem	n/a	n/a
Look-Up Table	MATLAB function	n/a	n/a
Torque Converter	Subsystem	n/a	n/a
shift_logic	Chart	17	17
selection_state.calc_th	Subsystem	n/a	n/a
Look-Up	MATLAB function	n/a	n/a

Effective lines of code for Stateflow blocks

Effective number of lines of code for Stateflow blocks

TYPE	COMPO...	PATH	QTY	STATEFLOW LOC	STATEFLOW LOC...
Model	sf_car		1	n/a	17
Chart	shift_logic	...hift_logic	1	17	17

- 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 5-49.

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.
MAAB Compliance	**Model Advisor standards check compliance - MAAB** (`mathworks.metrics.ModelAdvisorCheckCompliance.maab`)	For each component: • Percentage of checks passed • Status of each check Integration with the Model Advisor for more detailed results.
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.
MAAB Check Issues	**Model Advisor standards issues - MAAB** (`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.

Widget	Metric	Drill-In Data
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 5-49.

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 5-49.

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 support self-modifying masked library blocks. Analysis of these components might be incomplete.
- The Metrics Dashboard does not count MAAB 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 MAAB issues, but still report issues in the MAAB Modeling Guideline Compliance widget. For more information about these issues, click the MAAB Modeling Guideline Compliance widget.

See Also

More About

- “Collect Model Metrics Programmatically” on page 5-20
- “Model Metrics”
- “Collect Compliance Data and Explore Results in the Model Advisor” on page 5-33
- “Collect Metric Data Programmatically and View Data Through the Metrics Dashboard” on page 5-38

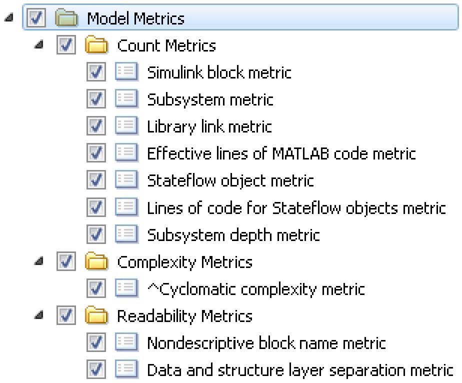
Collect Model Metrics Using the Model Advisor

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 From the Simulink Editor, select **Analysis > Model Advisor > Model Advisor**. A System Selector — Model Advisor dialog box opens. Click **OK**.

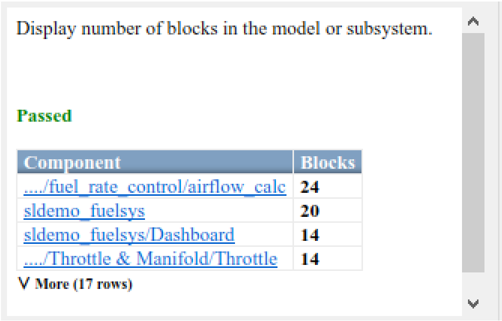
3 In the left pane of the Model Advisor, navigate to **By Task > Model Metrics**. Select the model metrics to run on your model.



- ☑ Model Metrics
  - ☑ Count Metrics
    - ☑ Simulink block metric
    - ☑ Subsystem metric
    - ☑ Library link metric
    - ☑ Effective lines of MATLAB code metric
    - ☑ Stateflow object metric
    - ☑ Lines of code for Stateflow objects metric
    - ☑ Subsystem depth metric
  - ☑ Complexity Metrics
    - ☑ ^Cyclomatic complexity metric
  - ☑ Readability Metrics
    - ☑ Nondescriptive block name metric
    - ☑ Data and structure layer separation metric

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.



Display number of blocks in the model or subsystem.

Passed

Component	Blocks
..../fuel_rate_control/airflow_calc	24
sldemo_fuelsys	20
sldemo_fuelsys/Dashboard	14
..../Throttle & Manifold/Throttle	14

V More (17 rows)

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 5-24
- “Collect Model Metrics Programmatically” on page 5-20
- “Create a Custom Model Metric” on page 5-12

Create a Custom Model Metric

To create your own custom model metric:

1. Use the `slmetric.metric.createNewMetricClass` function to create a new metric class derived from the base class `slmetric.metric.Metric`.
2. Set the following properties of the 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 metric 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.
   - `AggregateComponentDetails`: Returns all detailed results or aggregates detailed results of the component.

   Optionally, set these additional properties:

   - `Description`: Description of the metric.
   - `Version`: Metric version.
3. 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.

Create 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

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);
```

Create Nonvirtual Block Count Metric

To write the metric algorithm, open the `nonvirtualblockcount.m` file and add the metric to the file. For example, to edit the file, use the command `edit(className)`. For this example, you can create the metric algorithm by copying this logic into `nonvirtualblockcount.m file`.

```
classdef nonvirtualblockcount < slmetric.metric.Metric
    %nonvirtualblockcount calculates number of nonvirtual blocks per level.
    % BusCreator, BusSelector and BusAssign are treated as nonvirtual.
    properties
        VirtualBlockTypes = {'Demux','From','Goto','Ground', ...
            'GotoTagVisiblity','Mux','SignalSpecification', ...
            'Terminator','Inport'};
    end

    methods
    function this = nonvirtualblockcount()
        this.ID = 'nonvirtualblockcount';
        this.Name = 'Nonvirtual Block Count';
        this.Version = 1;
        this.CompileContext = 'None';
        this.Description = 'Algorithm that counts nonvirtual blocks per level.';
        this.ComponentScope = [Advisor.component.Types.Model, ...
            Advisor.component.Types.SubSystem];
        this.AggregationMode = slmetric.AggregationMode.Sum;
        this.AggregateComponentDetails = true;
        this.ResultChecksumCoverage = true;
        this.SupportsResultDetails = true;

    end

    function res = algorithm(this, component)
        % create a result object for this component
        res = slmetric.metric.Result();

        % set the component and metric ID
        res.ComponentID = component.ID;
        res.MetricID = this.ID;

        % Practice

        D1=slmetric.metric.ResultDetail('identifier 1','Name 1');
        D1.Value=0;
        D1.setGroup('Group1','Group1Name');
        D2=slmetric.metric.ResultDetail('identifier 2','Name 2');
        D2.Value=1;
        D2.setGroup('Group1','Group1Name');



        % use find_system to get all blocks inside this component
        blocks = find_system(getPath(component), ...
            'SearchDepth', 1, ...
            'Type', 'Block');

        isNonVirtual = true(size(blocks));

        for n=1:length(blocks)
            blockType = get_param(blocks{n}, 'BlockType');

            if any(strcmp(this.VirtualBlockTypes, blockType))
                isNonVirtual(n) = false;
            else
                switch blockType
                    case 'SubSystem'
                        % Virtual unless the block is conditionally executed
                        % or the Treat as atomic unit check box is selected.
                        if strcmp(get_param(blocks{n}, 'IsSubSystemVirtual'), ...
                                'on')
                            isNonVirtual(n) = false;
                        end
                    case 'Outport'
                        % Outport: Virtual when the block resides within
                        % SubSystem block (conditional or not), and
                        % does not reside in the root (top-level) Simulink window.
                        if component.Type ~= Advisor.component.Types.Model
                            isNonVirtual(n) = false;
                        end
                    case 'Selector'
                        % Virtual only when Number of input dimensions
                        % specifies 1 and Index Option specifies Select
```

```
                        % all, Index vector (dialog), or Starting index (dialog).
                        nod = get_param(blocks{n}, 'NumberOfDimensions');
                        ios = get_param(blocks{n}, 'IndexOptionArray');

                        ios_settings = {'Assign all', 'Index vector (dialog)', ...
                            'Starting index (dialog)'};

                        if nod == 1 && any(strcmp(ios_settings, ios))
                            isNonVirtual(n) = false;
                        end
                    case 'Trigger'
                        % Virtual when the output port is not present.
                        if strcmp(get_param(blocks{n}, 'ShowOutputPort'), 'off')
                            isNonVirtual(n) = false;
                        end
                    case 'Enable'
                        % Virtual unless connected directly to an Outport block.
                        isNonVirtual(n) = false;

                        if strcmp(get_param(blocks{n}, 'ShowOutputPort'), 'on')
                            pc = get_param(blocks{n}, 'PortConnectivity');

                            if ~isempty(pc.DstBlock) && ...
                                    strcmp(get_param(pc.DstBlock, 'BlockType'), ...
                                    'Outport')
                                isNonVirtual(n) = true;
                            end
                        end
                end
            end
        end

        blocks = blocks(isNonVirtual);

        res.Value = length(blocks);
    end
    end
end
```

Now that your new model metric is defined in `nonvirtualblockcount.m`, 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.

Load the `sldemo_mdlref_bus` model.

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

Create a metric engine object and set the analysis root.

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

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 instances of `slmetric.metric.Result`. In this example, display the nonvirtual block count metrics for the `sldemo_mdlref_busmodel`. 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
```

Here are the results.

```
MetricID: nonvirtualblockcount
  ComponentPath: sldemo_mdlref_bus
  Value: 15

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.

See Also

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

More About

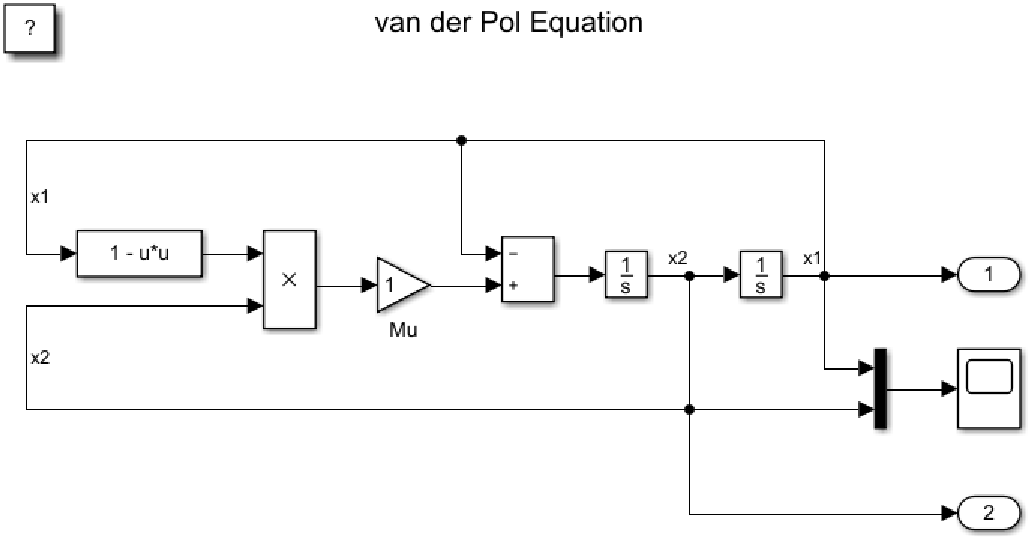
- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-24
- “Collect Model Metrics Programmatically” on page 5-20

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);
shh = get(0,'ShowHiddenHandles');
set(0,'ShowHiddenHandles','On');
hscope = findobj(0,'Type','Figure','Tag','SIMULINK_SIMSCOPE_FIGURE');
close(hscope);
set(0,'ShowHiddenHandles',shh);
```

Collect Metrics

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

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

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'});
```

Display and Store 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.

```
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
```

Here are the results.

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

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

Export Results to a Spreadsheet

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);
```

Export Results to an XML File

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 5-24
- “Collect Model Metrics Using the Model Advisor” on page 5-10
- “Create a Custom Model Metric” on page 5-12

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 these two aggregation properties:

- `AggregationMode`, 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.
- `AggregateComponentDetails` is a Boolean value, which has these options:
  - `true`: For metrics that return fine-granular results (that is, more than one result per component), the software aggregates these results to the component level by taking the sum of the values and measures properties. Returns a result that spans the complete component.
  - `false`: Returns the component results. The software does not aggregate the fine-granular results.

  The MathWorks model metrics that return fine-granular results are:

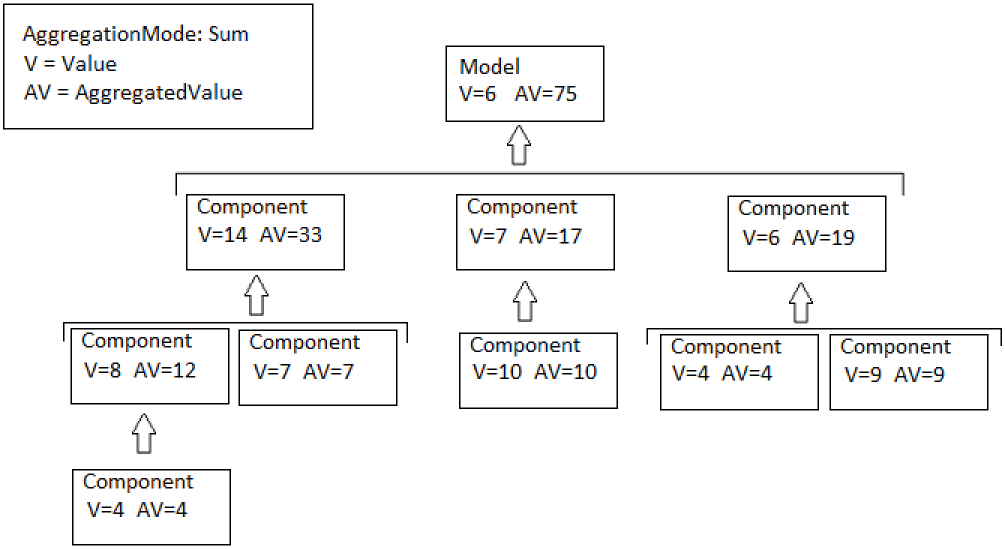
  - “Cyclomatic complexity metric”, which creates a result for each state in a Chart.

- “Effective lines of MATLAB code metric”, which creates a result for each function or subfunction inside a MATLAB function block or a MATLAB function in Stateflow.

You can find descriptions of MathWorks model metrics and their aggregation property settings 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” on page 5-12.

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 5-24
- “Collect Model Metrics Using the Model Advisor” on page 5-10
- “Create a Custom Model Metric” on page 5-12

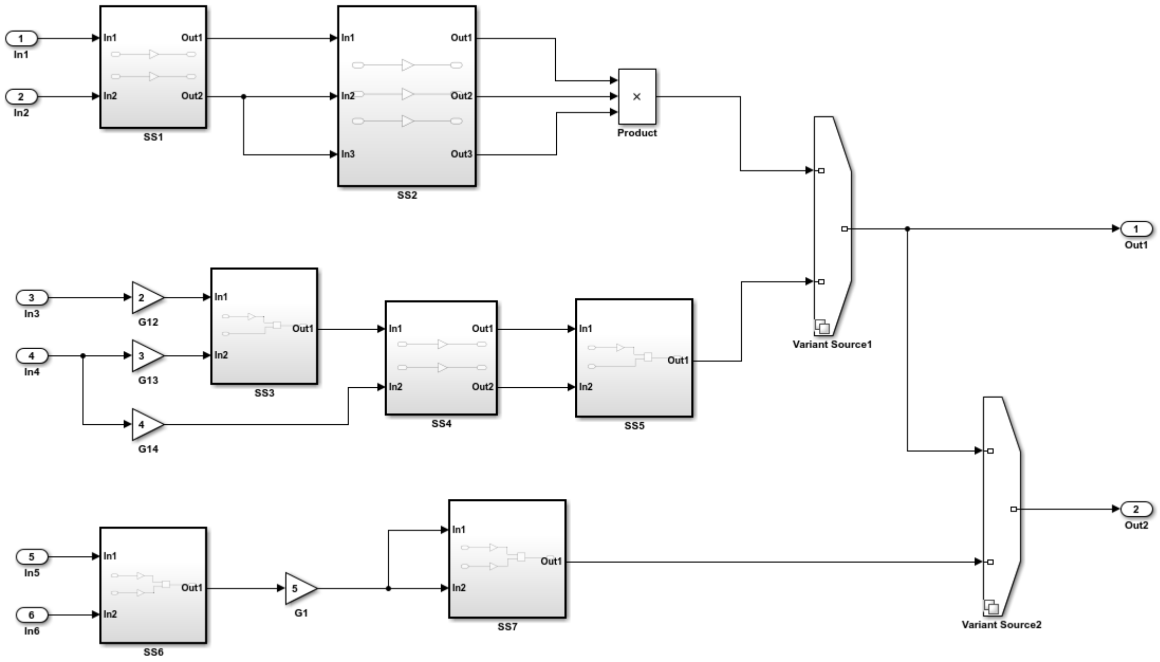
Enable Subsystem Reuse with Clone Detection

You can use the Metrics Dashboard tool to enable subsystem reuse 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. They can have different parameter settings and values. To replace clones with links to library blocks, from the Metrics Dashboard, you can open the Identify Modeling Clones tool.

Identify Exact Graphical Clones

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

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

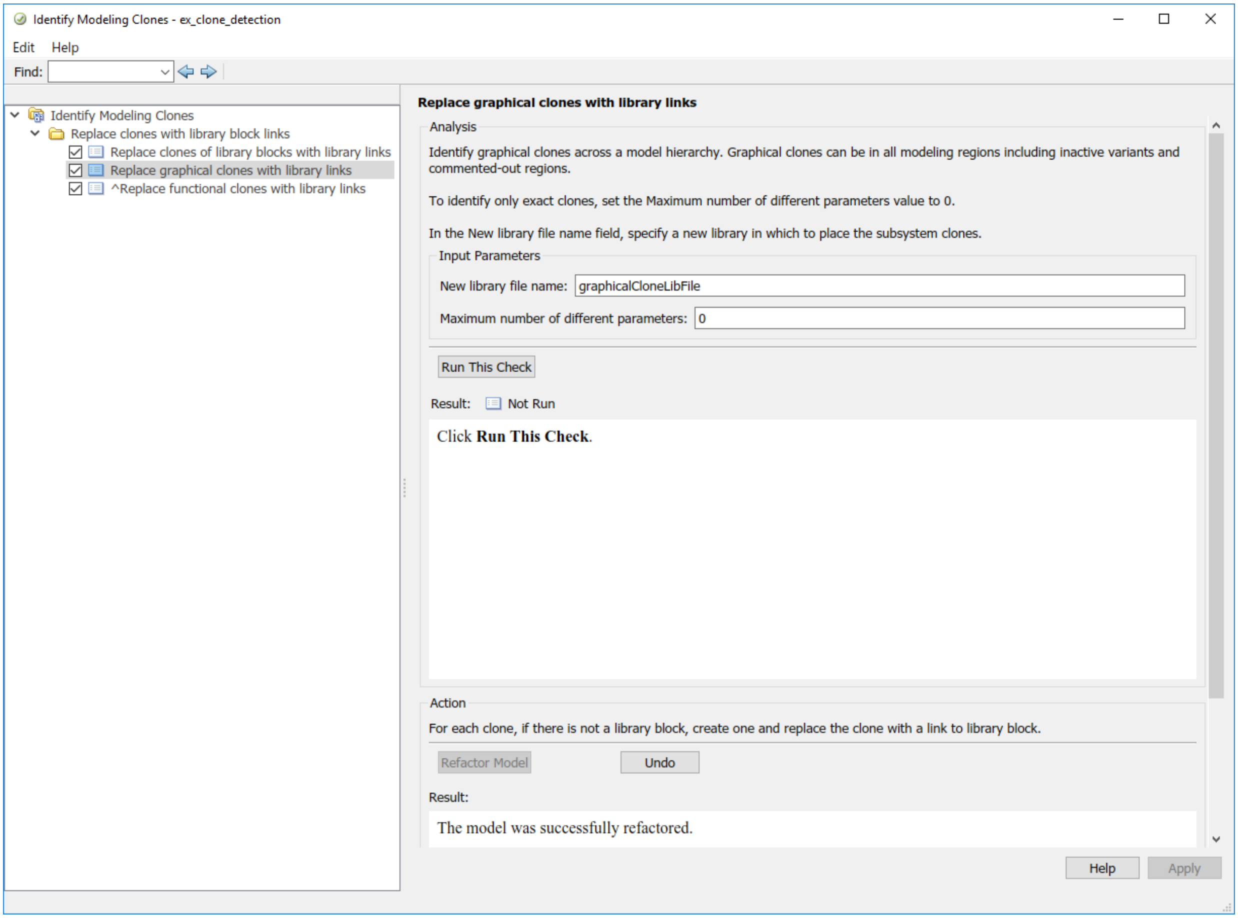


Copyright 2017 The MathWorks Inc.

1. Save the `ex_clone_detection.slx` model to a local working folder.
2. In the Simulink Editor, from the **Analysis** menu, select **Metrics Dashboard**.
3. On the toolstrip, click the **All Metrics** button.
4. In the **ARCHITECTURE** section, the yellow shaded 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.
5. 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 Detection tool by clicking **Open Conversion Tool**.
2. The Clone Detection tool opens and reruns the analysis. The results are in the step **Replace graphical clones with library links**. For more information, see “Enable Component Reuse by Using Clone Detection” on page 3-15.
3. The **Results** table contains hyperlinks to the subsystem clones.
4. Click **Refactor Model**. The Clone Detection tool replaces the clones with links to library blocks. The library blocks are in the library specified by the **New library file name** parameter. The library is on the MATLAB path. It has a default name of `graphicalCloneLibFile`. The **Refactor Model** button is now unavailable, and the **Undo** button is enabled.


Identify Modeling Clones - ex_clone_detection
Edit Help
Find:
Identify Modeling Clones
Replace clones with library block links
☑ Replace clones of library blocks with library links
☑ Replace graphical clones with library links
☑ ^Replace functional clones with library links

Replace graphical clones with library links

Analysis

Identify graphical clones across a model hierarchy. Graphical clones can be in all modeling regions including inactive variants and commented-out regions.

To identify only exact clones, set the Maximum number of different parameters value to 0.

In the New library file name field, specify a new library in which to place the subsystem clones.

Input Parameters

New library file name: graphicalCloneLibFile

Maximum number of different parameters: 0

Result: Not Run

Click **Run This Check**.

Action

For each clone, if there is not a library block, create one and replace the clone with a link to library block.

Result:

The model was successfully refactored.

After you refactor, you can remove the latest changes from the model by clicking the **Undo** button. Each time you refactor a model, the tool creates a back-up model in the folder that has the prefix `m2m_` plus the model name.

Run Model Metrics on Refactored Model

1. Navigate back to the **Metrics Dashboard**. Close the Metrics Dashboard and then open it again.
2. On the toolstrip, click the **All Metrics** button.

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"

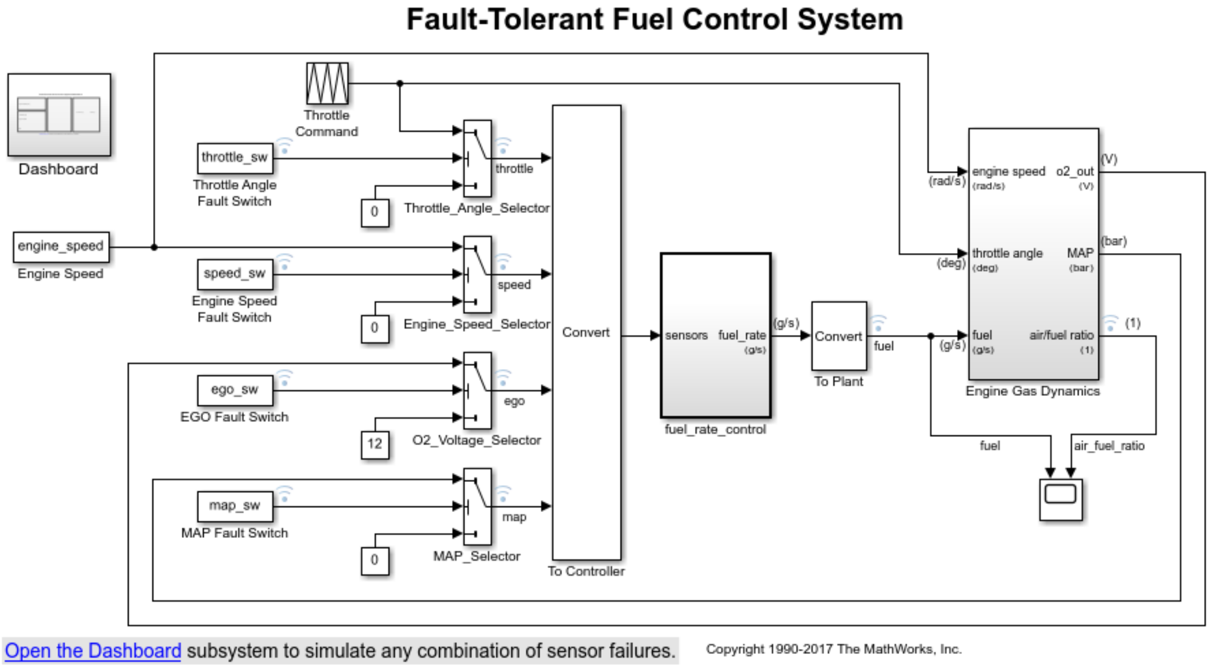
Collect Compliance Data and Explore Results in the Model Advisor

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 Dashboard subsystem to simulate any combination of sensor failures.

Open the Metrics Dashboard

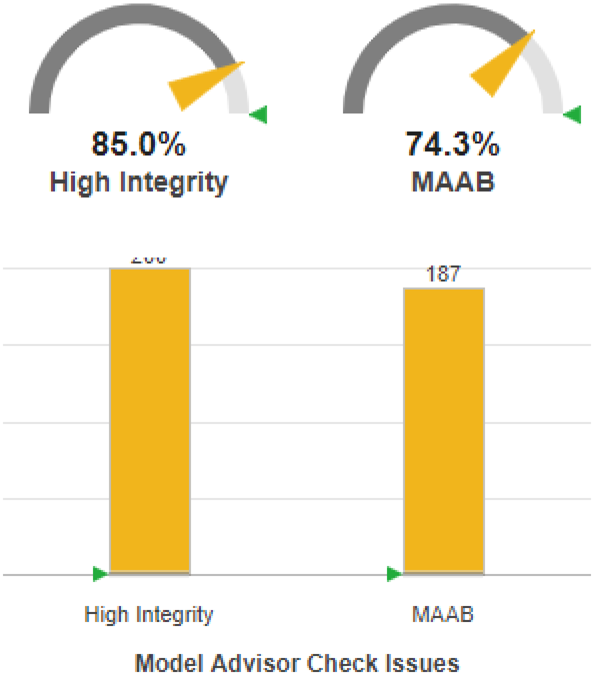
In the model window, open the Metrics Dashboard by selecting **Analysis > Metrics Dashboard**.

Collect Model Metrics

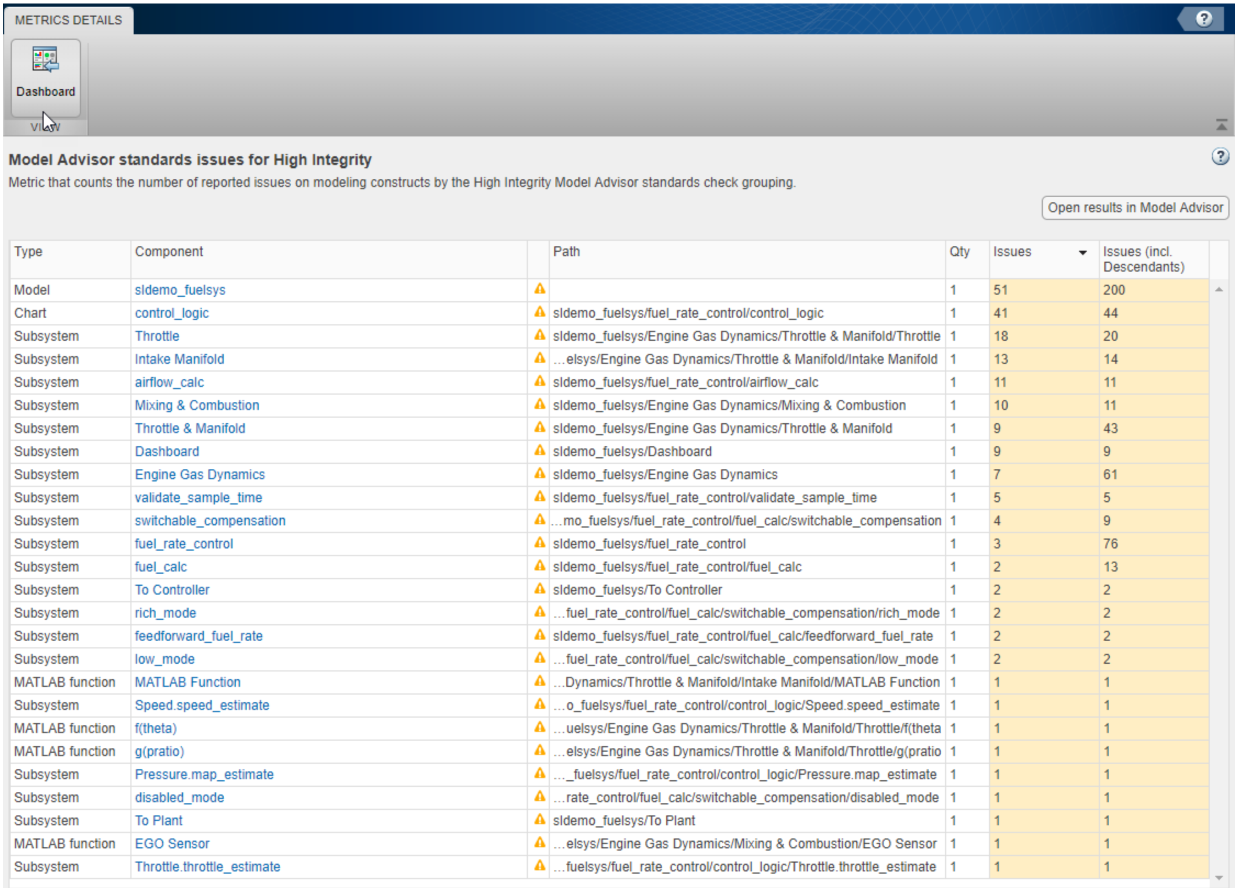
To collect the metric data for this model, click the **All Metrics** icon.

Explore Compliance Results

Locate the **MODELING GUIDELINE COMPLIANCE** section of the dashboard. This section displays the percentage of High Integrity and MAAB 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.



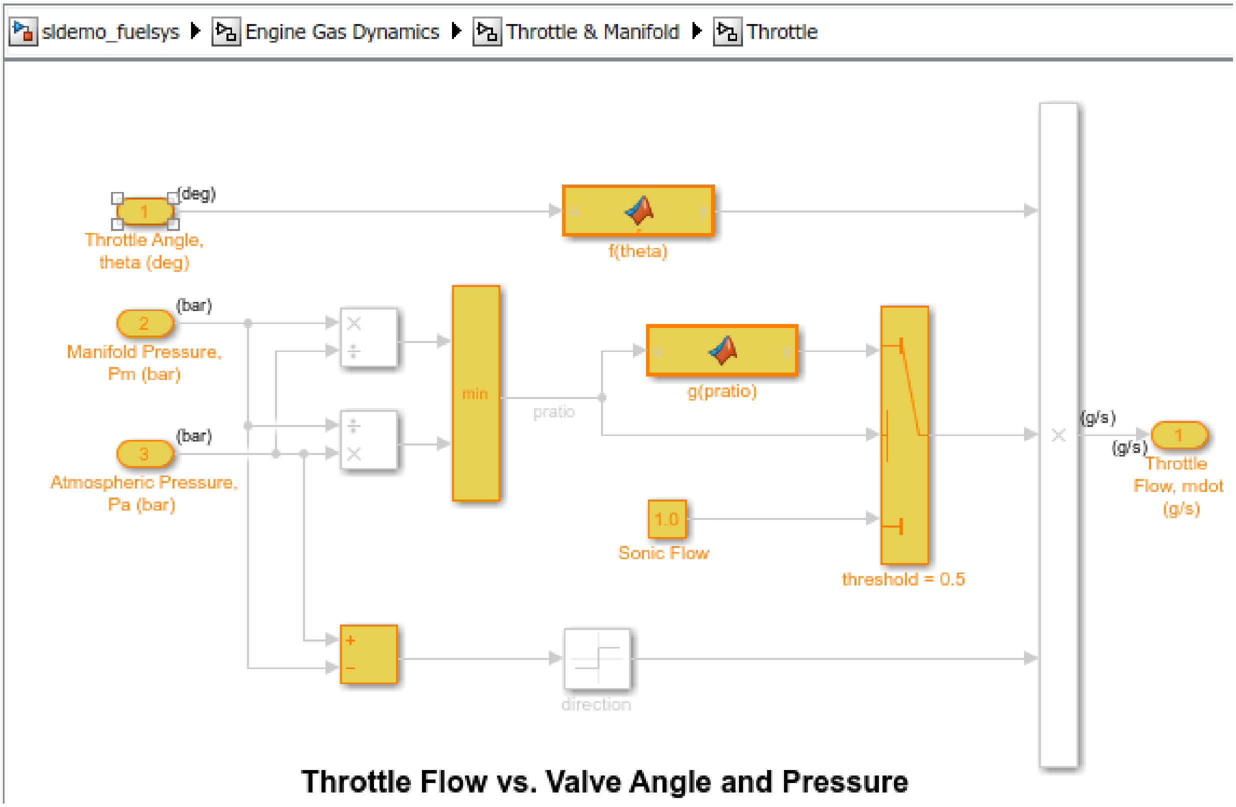
Model Advisor standards issues for High Integrity

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

Open results in Model Advisor

Type	Component		Path	Qty	Issues	Issues (incl. Descendants)
Model	sldemo_fuelsys	⚠		1	51	200
Chart	control_logic	⚠	sldemo_fuelsys/fuel_rate_control/control_logic	1	41	44
Subsystem	Throttle	⚠	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	18	20
Subsystem	Intake Manifold	⚠	...elsys/Engine Gas Dynamics/Throttle & Manifold/Intake Manifold	1	13	14
Subsystem	airflow_calc	⚠	sldemo_fuelsys/fuel_rate_control/airflow_calc	1	11	11
Subsystem	Mixing & Combustion	⚠	sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	10	11
Subsystem	Throttle & Manifold	⚠	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	9	43
Subsystem	Dashboard	⚠	sldemo_fuelsys/Dashboard	1	9	9
Subsystem	Engine Gas Dynamics	⚠	sldemo_fuelsys/Engine Gas Dynamics	1	7	61
Subsystem	validate_sample_time	⚠	sldemo_fuelsys/fuel_rate_control/validate_sample_time	1	5	5
Subsystem	switchable_compensation	⚠	...mo_fuelsys/fuel_rate_control/fuel_calc/switchable_compensation	1	4	9
Subsystem	fuel_rate_control	⚠	sldemo_fuelsys/fuel_rate_control	1	3	76
Subsystem	fuel_calc	⚠	sldemo_fuelsys/fuel_rate_control/fuel_calc	1	2	13
Subsystem	To Controller	⚠	sldemo_fuelsys/To Controller	1	2	2
Subsystem	rich_mode	⚠	...fuel_rate_control/fuel_calc/switchable_compensation/rich_mode	1	2	2
Subsystem	feedforward_fuel_rate	⚠	sldemo_fuelsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate	1	2	2
Subsystem	low_mode	⚠	...fuel_rate_control/fuel_calc/switchable_compensation/low_mode	1	2	2
MATLAB function	MATLAB Function	⚠	...Dynamics/Throttle & Manifold/Intake Manifold/MATLAB Function	1	1	1
Subsystem	Speed.speed_estimate	⚠	...o_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate	1	1	1
MATLAB function	f(theta)	⚠	...uelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle/f(theta	1	1	1
MATLAB function	g(pratio)	⚠	...elsys/Engine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio	1	1	1
Subsystem	Pressure.map_estimate	⚠	..._fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate	1	1	1
Subsystem	disabled_mode	⚠	...rate_control/fuel_calc/switchable_compensation/disabled_mode	1	1	1
Subsystem	To Plant	⚠	sldemo_fuelsys/To Plant	1	1	1
MATLAB function	EGO Sensor	⚠	...elsys/Engine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	1	1
Subsystem	Throttle.throttle_estimate	⚠	...fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate	1	1	1

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 and the compliance percentage is increased.

See Also

More About

- "Collect and Explore Metric Data by Using the Metrics Dashboard" on page 5-2
- "Collect Model Metrics Programmatically" on page 5-20

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');
execute(metric_engine);
```

Determine Model Compliance with MAAB Guidelines

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

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

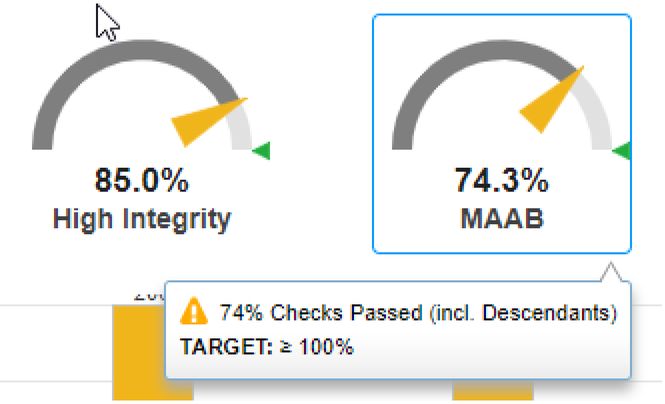
Open the Metrics Dashboard

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.

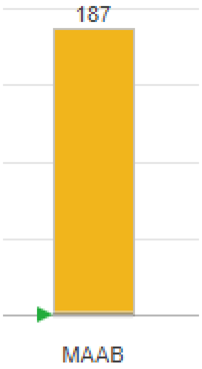
Find the **MODELING GUIDELINE COMPLIANCE** section of the dashboard. For each category of compliance checks, the gauge indicates the percentage of compliance checks that passed.



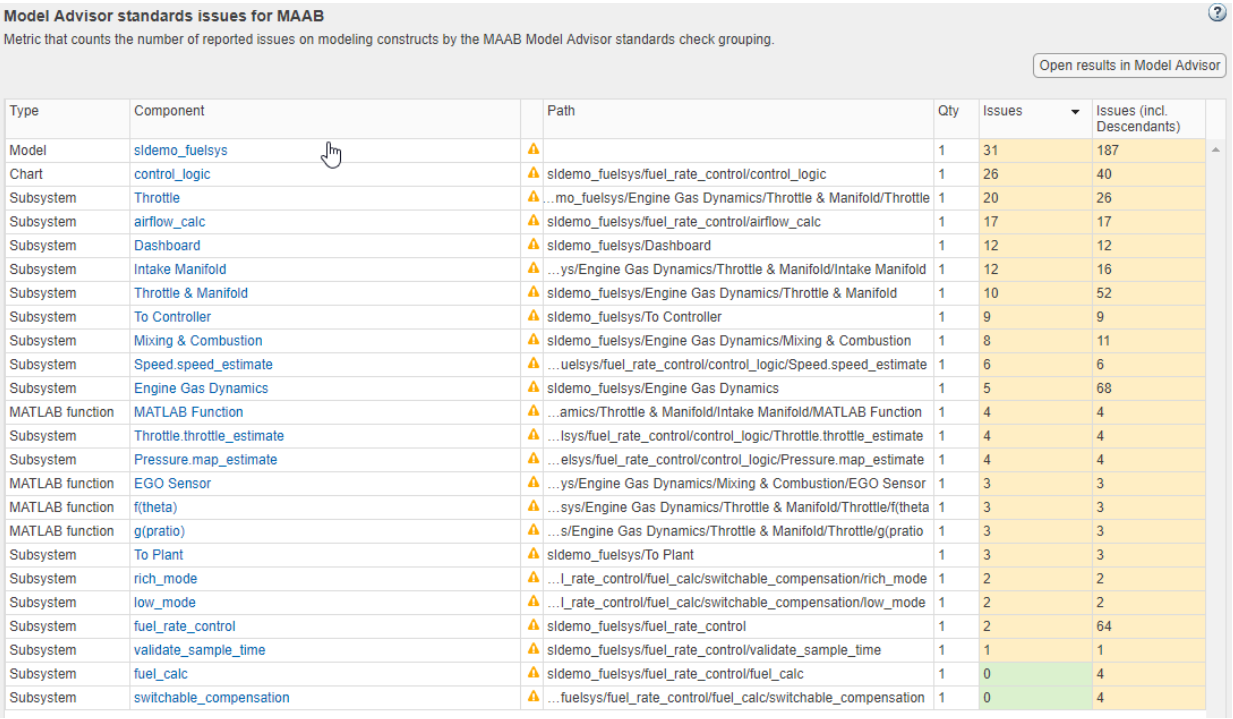
The dashboard reports the same MAAB compliance percentage as the `slmetric.Engine` API reports.

Explore the MAAB Compliance Results

Underneath the percentage gauges, the bar chart indicates the number of compliance check issues. Click anywhere in the MAAB 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.

Type	Component		Path	Qty	Issues	Issues (incl. Descendants)
Model	sldemo_fuelsys	⚠		1	31	187
Chart	control_logic	⚠	sldemo_fuelsys/fuel_rate_control/control_logic	1	26	40
Subsystem	Throttle	⚠	...mo_fuelsys/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	20	26
Subsystem	airflow_calc	⚠	sldemo_fuelsys/fuel_rate_control/airflow_calc	1	17	17
Subsystem	Dashboard	⚠	sldemo_fuelsys/Dashboard	1	12	12
Subsystem	Intake Manifold	⚠	...ys/Engine Gas Dynamics/Throttle & Manifold/Intake Manifold	1	12	16
Subsystem	Throttle & Manifold	⚠	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	10	52
Subsystem	To Controller	⚠	sldemo_fuelsys/To Controller	1	9	9
Subsystem	Mixing & Combustion	⚠	sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	8	11
Subsystem	Speed.speed_estimate	⚠	...uelsys/fuel_rate_control/control_logic/Speed.speed_estimate	1	6	6
Subsystem	Engine Gas Dynamics	⚠	sldemo_fuelsys/Engine Gas Dynamics	1	5	68
MATLAB function	MATLAB Function	⚠	...amics/Throttle & Manifold/Intake Manifold/MATLAB Function	1	4	4
Subsystem	Throttle.throttle_estimate	⚠	...lsys/fuel_rate_control/control_logic/Throttle.throttle_estimate	1	4	4
Subsystem	Pressure.map_estimate	⚠	...elsys/fuel_rate_control/control_logic/Pressure.map_estimate	1	4	4
MATLAB function	EGO Sensor	⚠	...ys/Engine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	3	3
MATLAB function	f(theta)	⚠	...sys/Engine Gas Dynamics/Throttle & Manifold/Throttle/f(theta	1	3	3
MATLAB function	g(pratio)	⚠	...s/Engine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio	1	3	3
Subsystem	To Plant	⚠	sldemo_fuelsys/To Plant	1	3	3
Subsystem	rich_mode	⚠	...l_rate_control/fuel_calc/switchable_compensation/rich_mode	1	2	2
Subsystem	low_mode	⚠	...l_rate_control/fuel_calc/switchable_compensation/low_mode	1	2	2
Subsystem	fuel_rate_control	⚠	sldemo_fuelsys/fuel_rate_control	1	2	64
Subsystem	validate_sample_time	⚠	sldemo_fuelsys/fuel_rate_control/validate_sample_time	1	1	1
Subsystem	fuel_calc	⚠	sldemo_fuelsys/fuel_rate_control/fuel_calc	1	0	4
Subsystem	switchable_compensation	⚠	...fuelsys/fuel_rate_control/fuel_calc/switchable_compensation	1	0	4

See Also

More About

- “Collect Model Metrics Programmatically” on page 5-20
- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2

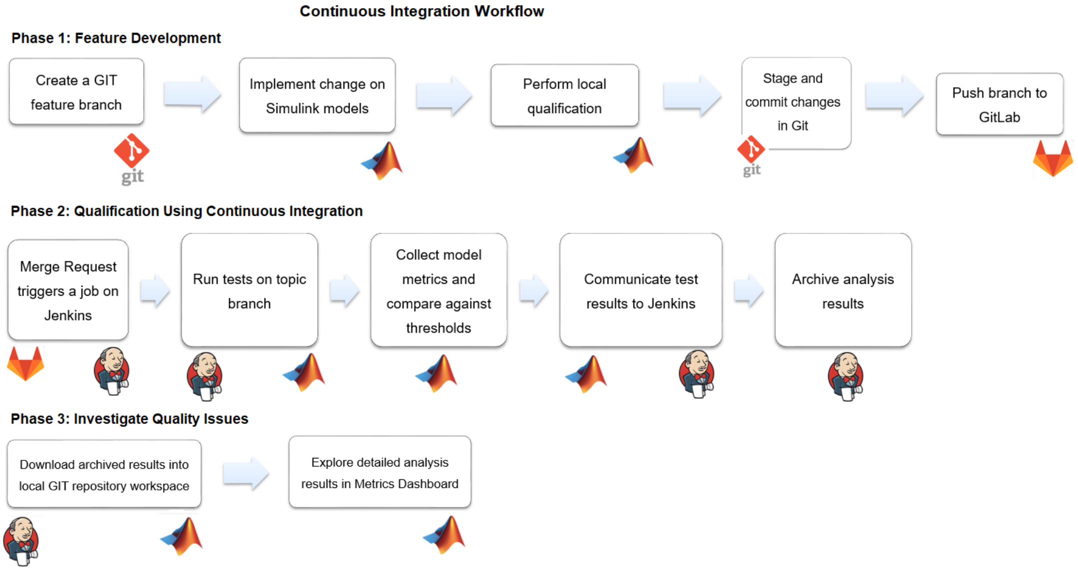
Fix Metric Threshold Violations in a Continuous Integration Systems Workflow

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.


Continuous Integration Workflow
Phase 1: Feature Development
Create a GIT feature branch
Implement change on Simulink models
Perform local qualification
Stage and commit changes in Git
Push branch to GitLab
git
git
Phase 2: Qualification Using Continuous Integration
Merge Request triggers a job on Jenkins
Run tests on topic branch
Collect model metrics and compare against thresholds
Communicate test results to Jenkins
Archive analysis results
Phase 3: Investigate Quality Issues
Download archived results into local GIT repository workspace
Explore detailed analysis results in Metrics Dashboard

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 5-49.

- An `sl_customization.m` file that activates the Model Advisor configuration file to customize the Model Advisor checks. For more information, see “Create and Add Custom Checks - Basic Examples” on page 7-6
- 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

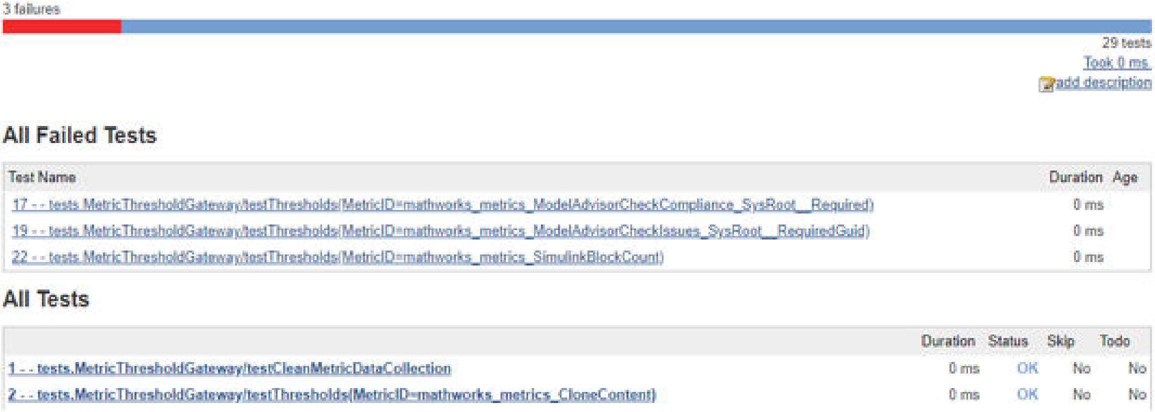
1 If the letters WIP: are not at the beginning of the merge request, the push command triggers a Jenkins build. In the Jenkins Setup part of this example, you configured Jenkins to perform a build when you pushed changes to GitLab. To remove the letters, click **Resolve WIP status**.

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.



3 failures

29 tests

Took 0 ms.

add description

All Failed Tests

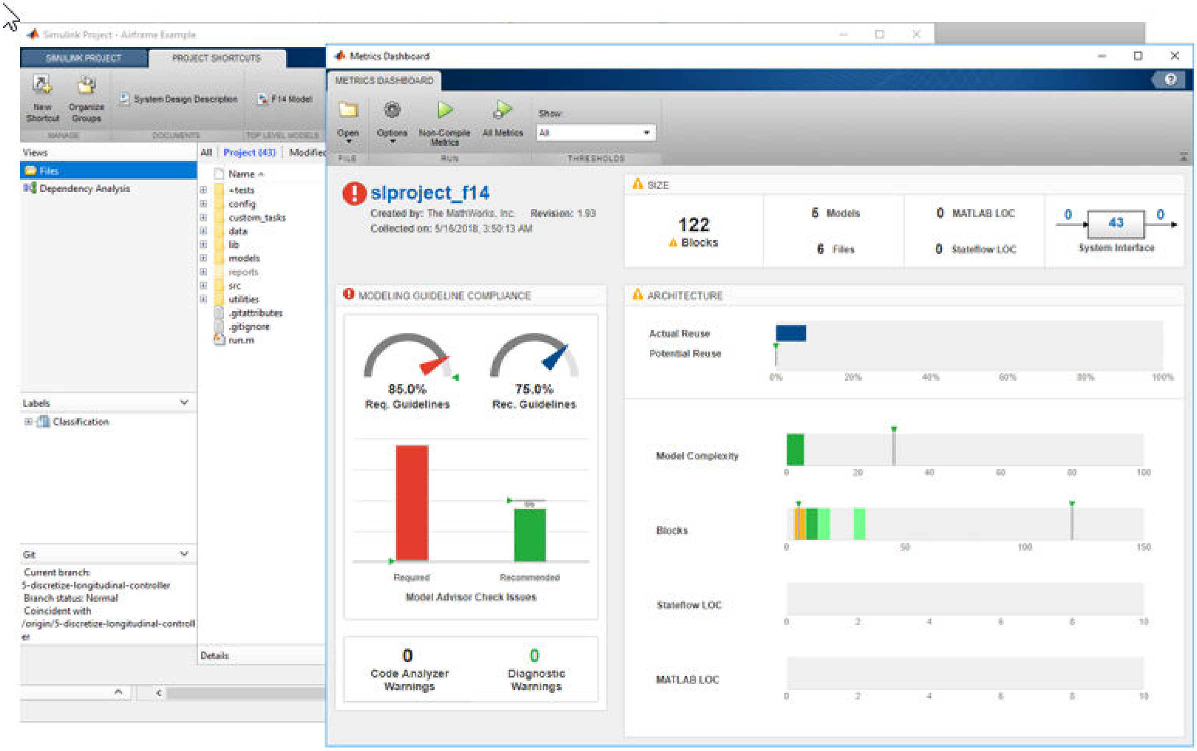
Test Name	Duration	Age
17 - - tests.MetricThresholdGateway/testThresholds(MetricID=mathworks_metrics_ModelAdvisorCheckCompliance_SysRoot__Required)	0 ms	
19 - - tests.MetricThresholdGateway/testThresholds(MetricID=mathworks_metrics_ModelAdvisorCheckIssues_SysRoot__RequiredGuid)	0 ms	
22 - - tests.MetricThresholdGateway/testThresholds(MetricID=mathworks_metrics_SimulinkBlockCount)	0 ms	

All Tests

	Duration	Status	Skip	Todo
1 - - tests.MetricThresholdGateway/testCleanMetricDataCollection	0 ms	OK	No	No
2 - - tests.MetricThresholdGateway/testThresholds(MetricID=mathworks_metrics_CloneContent)	0 ms	OK	No	No

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 “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5. 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.

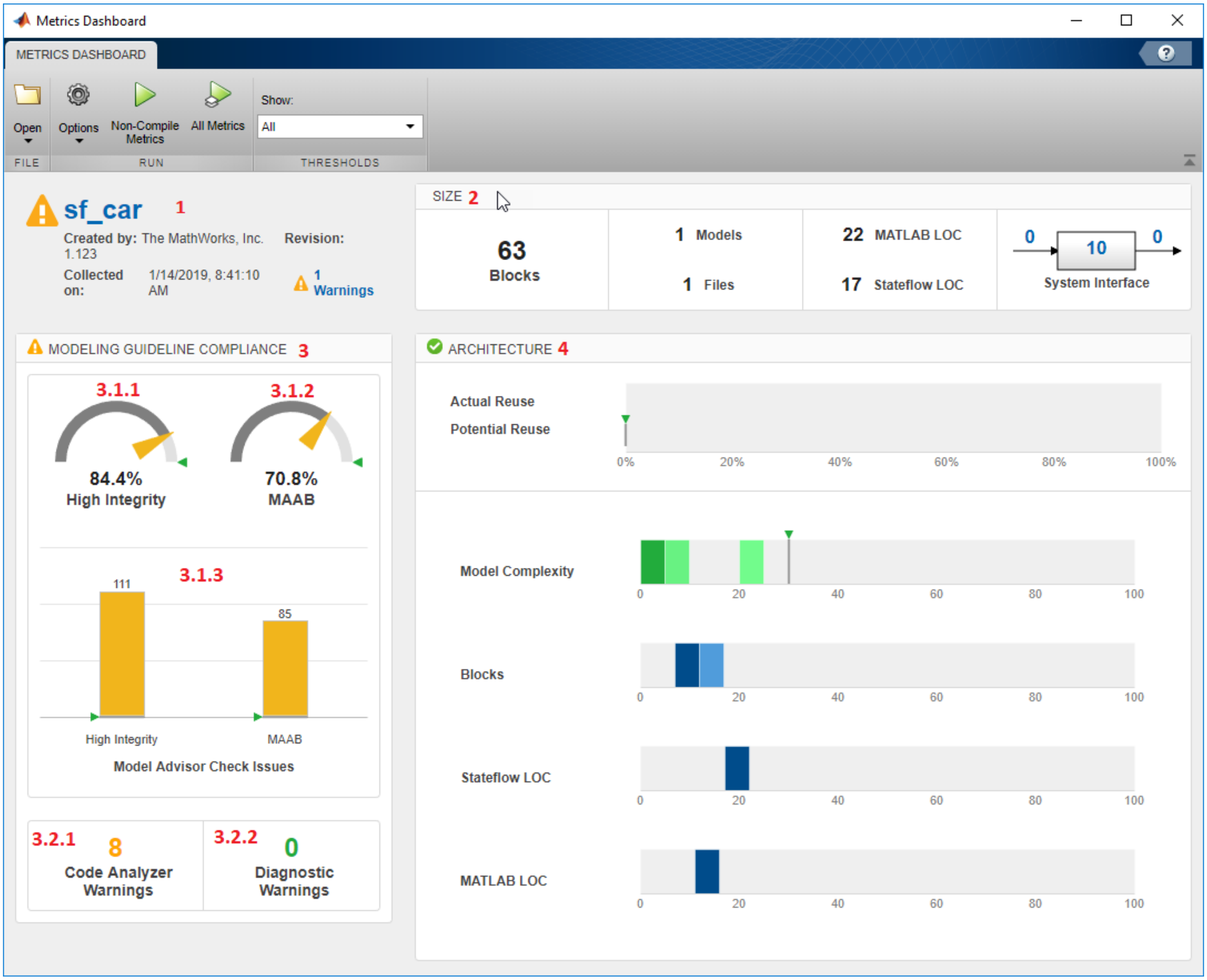
```
layoutWidget = getWidgets(layout);
```

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

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

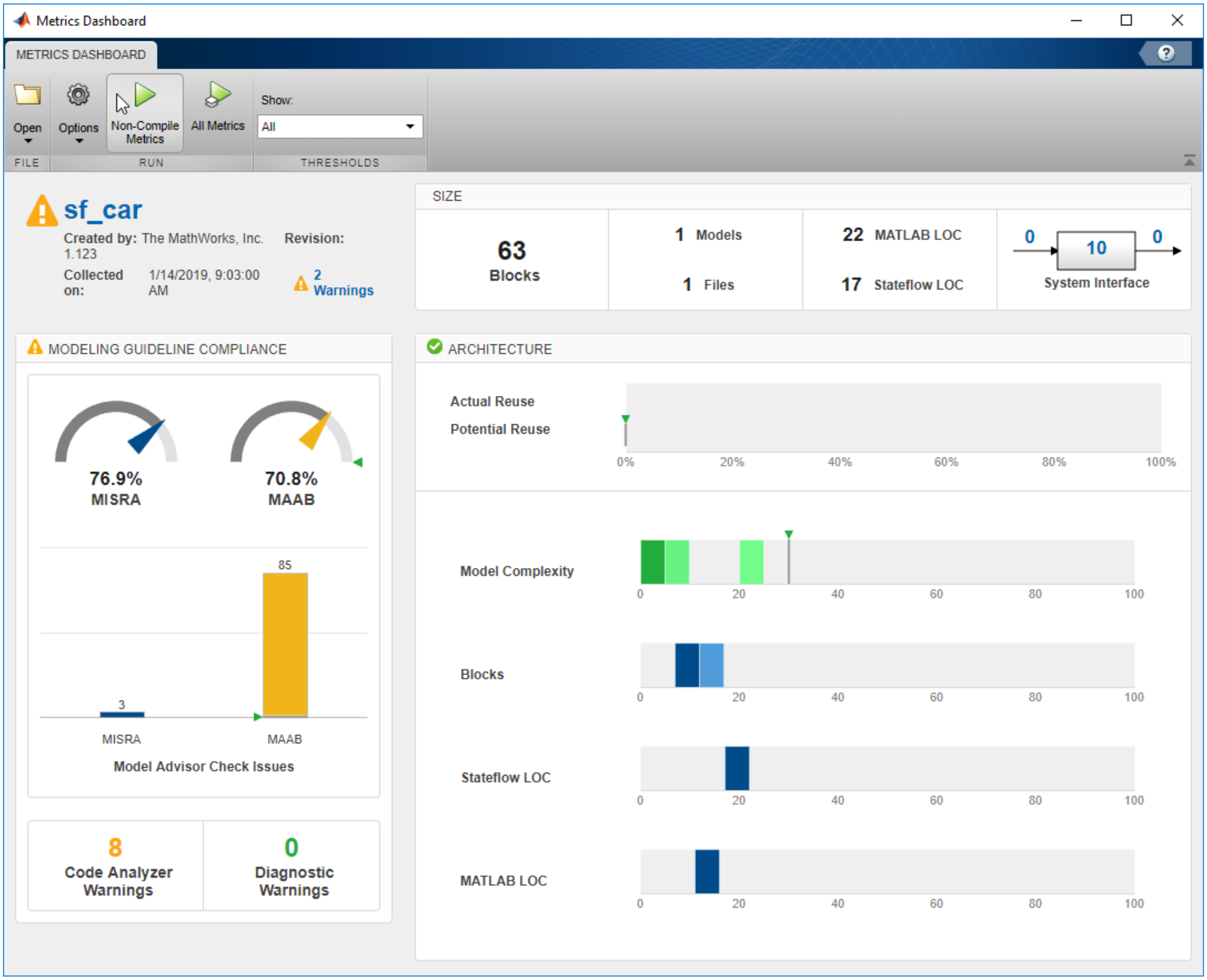
13 Set the active configurations.

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

14 For your model, open the Metrics Dashboard.

```
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 Create a custom metric class.

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

2 Create the nonvirtual block count metric by adding this code to the `nonvirtualblockcount.m` file.

```
classdef nonvirtualblockcount < slmetric.metric.Metric
    %nonvirtualblockcount calculates number of nonvirtual blocks per level.
    % BusCreator, BusSelector and BusAssign are treated as nonvirtual.
    properties
        VirtualBlockTypes = {'Demux','From','Goto','Ground', ...
            'GotoTagVisiblity','Mux','SignalSpecification', ...
            'Terminator','Inport'};
    end

    methods
    function this = nonvirtualblockcount()
        this.ID = 'nonvirtualblockcount';
        this.Name = 'Nonvirtual Block Count';
        this.Version = 1;
        this.CompileContext = 'None';
        this.Description = 'Algorithm that counts nonvirtual blocks per level.';
        this.AggregatedValueName = 'Nonvirtual Blocks (incl. Descendants)';
        this.ValueName = 'Nonvirtual Blocks';
        this.ComponentScope = [Advisor.component.Types.Model, ...
            Advisor.component.Types.SubSystem];
        this.AggregationMode = slmetric.AggregationMode.Sum;
        this.AggregateComponentDetails = true;
        this.ResultChecksumCoverage = true;
        this.SupportsResultDetails = false;

    end

    function res = algorithm(this, component)
        % create a result object for this component
        res = slmetric.metric.Result();

        % set the component and metric ID
        res.ComponentID = component.ID;
        res.MetricID = this.ID;

        % Practice
        D1=slmetric.metric.ResultDetail('identifier 1','Name 1');
        D1.Value=0;
        D1.setGroup('Group1','Group1Name');
        D2=slmetric.metric.ResultDetail('identifier 2','Name 2');
        D2.Value=1;
        D2.setGroup('Group1','Group1Name');



        % use find_system to get all blocks inside this component
        blocks = find_system(getPath(component), ...
            'SearchDepth', 1, ...
            'Type', 'Block');
```

```
isNonVirtual = true(size(blocks));

for n=1:length(blocks)
    blockType = get_param(blocks{n}, 'BlockType');

    if any(strcmp(this.VirtualBlockTypes, blockType))
        isNonVirtual(n) = false;
    else
        switch blockType
            case 'SubSystem'
                % Virtual unless the block is conditionally executed
                % or the Treat as atomic unit check box is selected.
                if strcmp(get_param(blocks{n}, 'IsSubSystemVirtual'), ...
                        'on')
                    isNonVirtual(n) = false;
                end
            case 'Outport'
                % Outport: Virtual when the block resides within
                % SubSystem block (conditional or not), and
                % does not reside in the root (top-level) Simulink window.
                if component.Type ~= Advisor.component.Types.Model
                    isNonVirtual(n) = false;
                end
            case 'Selector'
                % Virtual only when Number of input dimensions
                % specifies 1 and Index Option specifies Select
                % all, Index vector (dialog), or Starting index (dialog).
                nod = get_param(blocks{n}, 'NumberOfDimensions');
                ios = get_param(blocks{n}, 'IndexOptionArray');

                ios_settings = {'Assign all', 'Index vector (dialog)', ...
                    'Starting index (dialog)'};

                if nod == 1 && any(strcmp(ios_settings, ios))
                    isNonVirtual(n) = false;
                end
            case 'Trigger'
                % Virtual when the output port is not present.
                if strcmp(get_param(blocks{n}, 'ShowOutputPort'), 'off')
                    isNonVirtual(n) = false;
                end
            case 'Enable'
                % Virtual unless connected directly to an Outport block.
                isNonVirtual(n) = false;

                if strcmp(get_param(blocks{n}, 'ShowOutputPort'), 'on')
                    pc = get_param(blocks{n}, 'PortConnectivity');

                    if ~isempty(pc.DstBlock) && ...
                            strcmp(get_param(pc.DstBlock, 'BlockType'), ...
                            'Outport')
                        isNonVirtual(n) = true;
                    end
                end
                end
            end
        end

        blocks = blocks(isNonVirtual);

        res.Value = length(blocks);
    end
    end
end
```

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

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

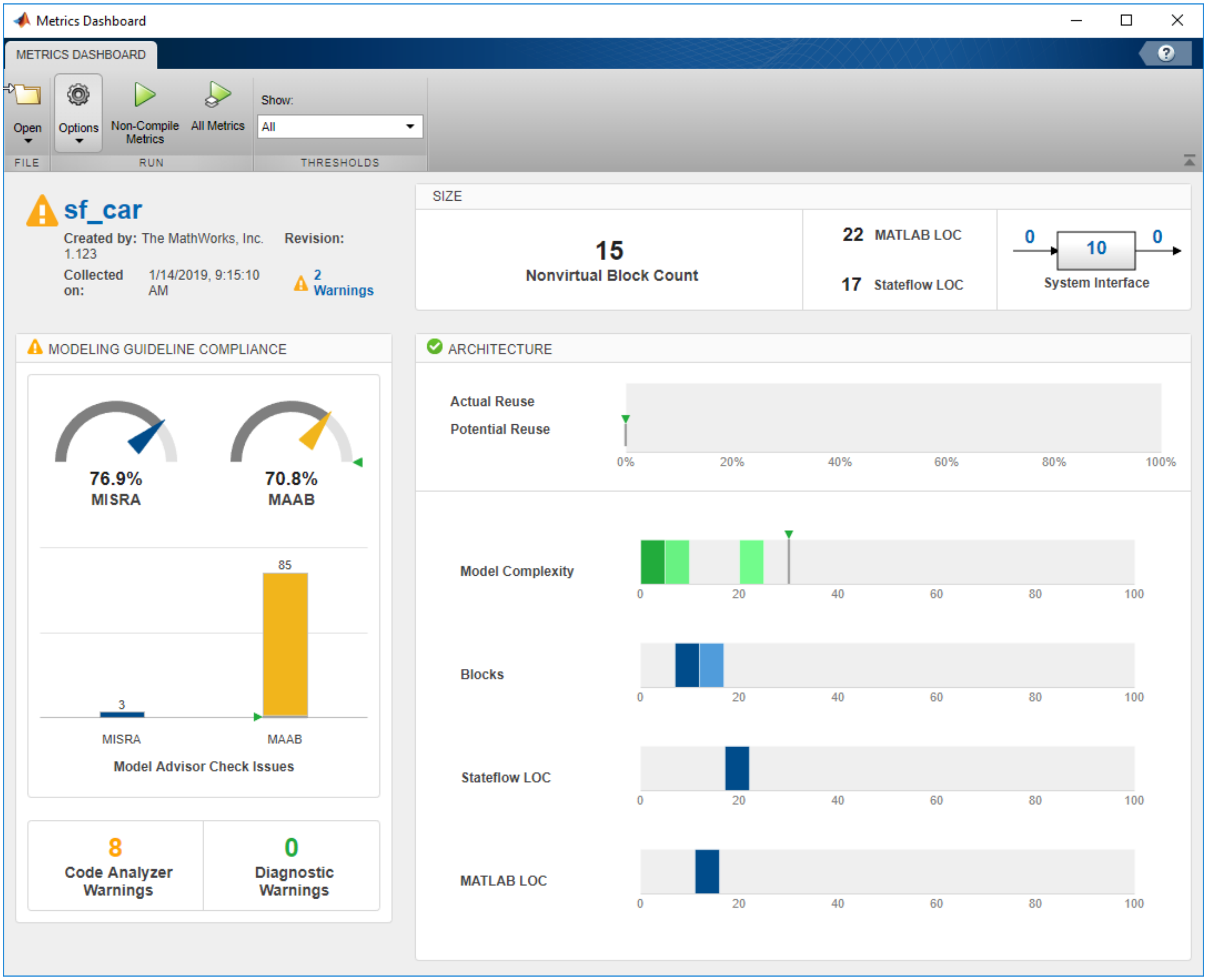
8. Set the active configurations.

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

9 For your model, open the Metrics Dashboard.

```
metricsdashboard sf_car
```

10 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.



Metrics Dashboard

sf_car

Created by: The MathWorks, Inc.

Revision: 1.123

Collected on: 1/14/2019, 9:15:10 AM

2 Warnings

11 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.

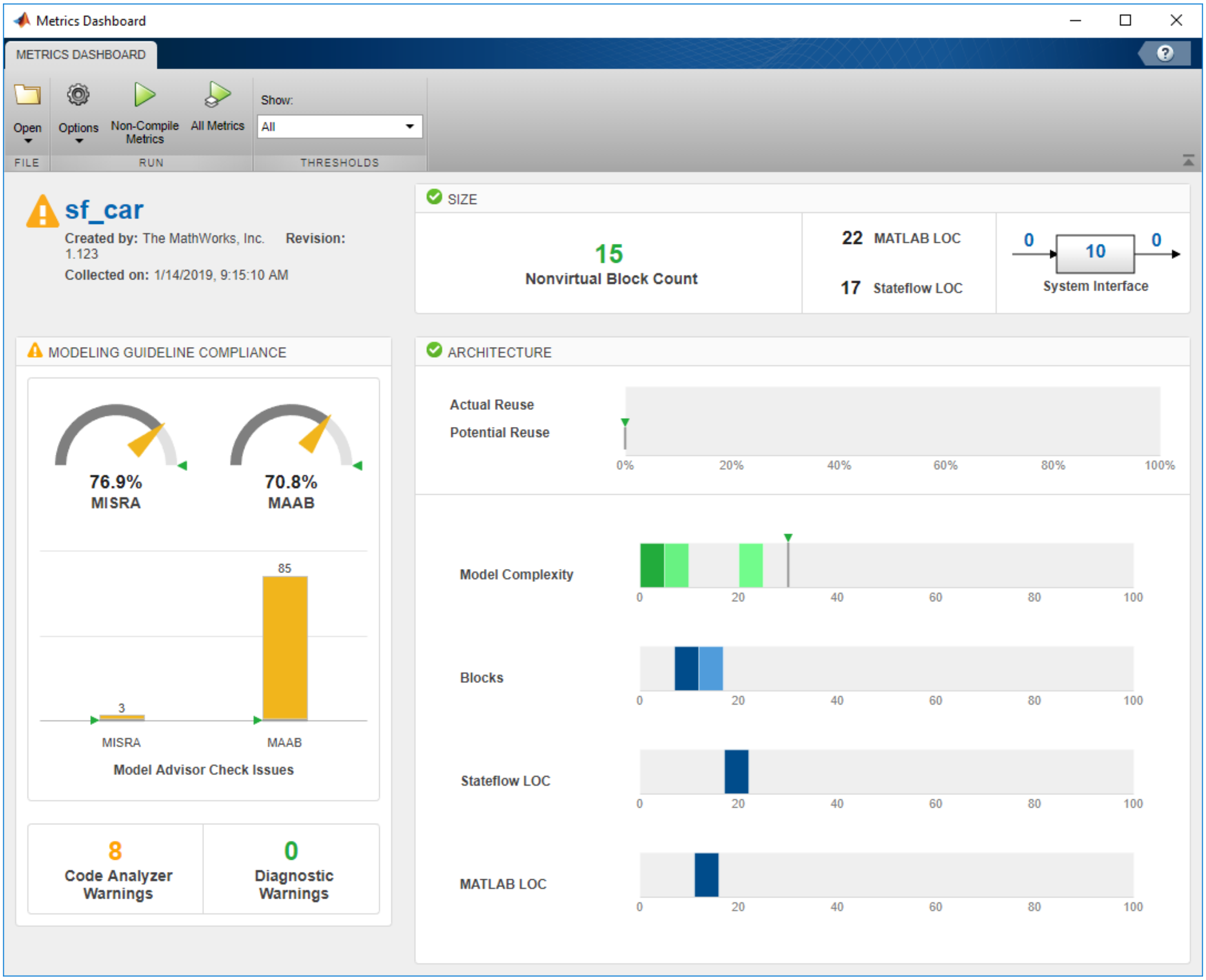
```
save(metricconfig,'FileName','MetricConfig.xml');
save(dashboardconfig,'Filename','DashboardConfig.xml');
```

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
```



sf_car

Created by: The MathWorks, Inc. Revision: 1.123

Collected on: 1/14/2019, 9:15:10 AM

Show: All

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, 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

Overview of Customizing the Model Advisor

Model Advisor Customization

Using Model Advisor APIs and the Model Advisor Configuration Editor, you can:

- Create your own Model Advisor checks.
- Create custom configurations.
- Specify the order in which you make changes to your model.
- Create multiple custom configurations for different projects or modeling guidelines, and switch between these configurations in the Model Advisor.
- Deploy custom configurations to your users.

To	See
Create Model Advisor checks.	“Create Model Advisor Checks”
Format check results.	“Format Check Results” on page 7-86
Create custom Model Advisor configurations.	“Create Custom Configurations” on page 8-2
Specify the order in which you make changes to your model.	“Organize and Deploy Model Advisor Checks”
Deploy custom configurations to your users.	“Organize and Deploy Model Advisor Checks”
Verify that models comply with modeling guidelines.	“Check Model Compliance”

Requirements for Customizing the Model Advisor

Before customizing the Model Advisor:

- If you want to create checks, know how to create a MATLAB script. For more information, see “Create Scripts” (MATLAB).
- Understand how to access model constructs that you want to check. For example, know how to find block and model parameters. For more information on using utilities for creating check callbacks, see “Common Utilities for Creating Checks” on page 7-5.

7

Create Model Advisor Checks

Create Model Advisor Checks Workflow

1. On your MATLAB path, create a recustomization file named `sl_customization.m`. In this file, create a `sl_customization()` function to register the custom checks that you create with the Model Advisor. For detailed information, see “Register Checks” on page 7-43.
2. Define custom checks and where they appear in the Model Advisor. For detailed information, see “Define Custom Checks” on page 7-48.
3. Specify what actions you want the Model Advisor to take for the custom checks by creating a check callback function for each custom check. For detailed information, see “Create Callback Functions and Results” on page 7-56.
4. Optionally, specify what automatic fix operations the Model Advisor performs by creating an action callback function. For detailed information, see “Action Callback Function” on page 7-65.

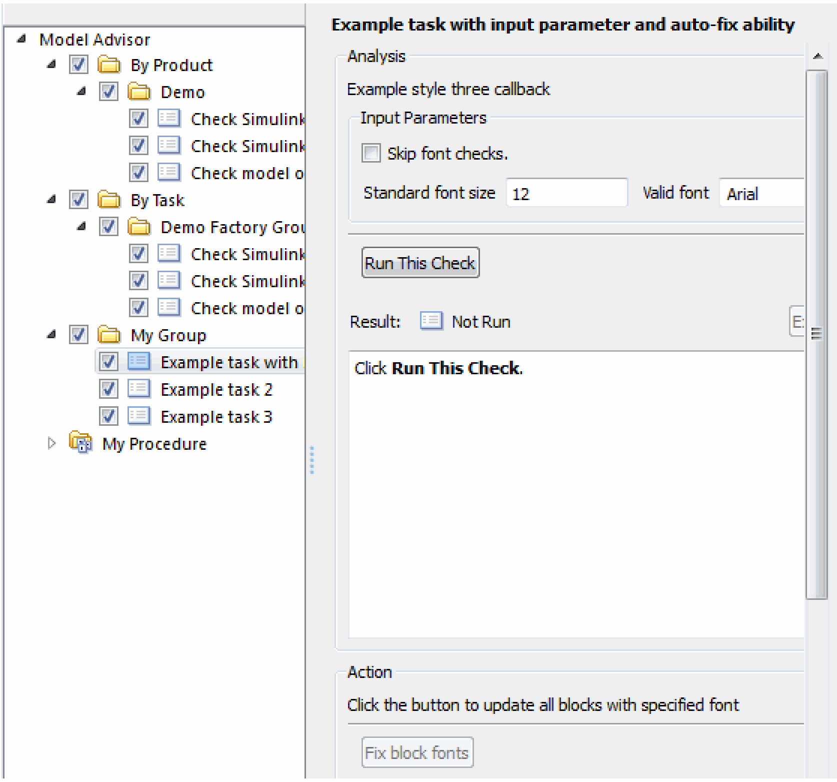
Customization File Overview

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

Function	Description	When Required
`sl_customization()`	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at start-up. See "Register Checks" on page 7-43.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See "Define Custom Checks" on page 7-48.	Required for custom checks and to add custom checks to the **By Product** folder. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings** > **Preferences** dialog box.
Check callback functions	Defines the actions of the custom checks. See "Create Callback Functions and Results" on page 7-56.	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 7-52.	Optional
One or more calls to checklist views	Specifies calls to the Model Advisor Result Explorer for custom checks. See "Define Model Advisor Result Explorer Views" on page 7-53.	Optional

Function	Description	When Required
One or more calls to check actions	Specifies actions the software performs for custom checks. See “Define Check Actions” on page 7-54 and “Action Callback Function” on page 7-65.	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.



Example style three callback

□ Skip font checks.

Standard font size 12 Valid font Arial

Result: Not Run

Click **Run This Check**.

Click the button to update all blocks with specified font

Common Utilities for Creating Checks

When you create a custom check, there are common Simulink utilities that you can use to make the check perform different actions. Following is a list of utilities and when to use them. In the Utility column, click the link for more information about the utility.

Utility	Used For...
`find_system`	Getting handle or path to: • 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 and Add Custom Checks - Basic Examples

To	See
Add a customized check to a Model Advisor **By Product > Demo** subfolder.	"Add Custom Check to by Product Folder" on page 7-6
Create a Model Advisor pass/fail check.	"Create Customized Pass/Fail Check" on page 7-7
Create a Model Advisor pass/fail check with a fix action.	"Create Customized Pass/Fail Check with Fix Action" on page 7-10
Create a Model Advisor pass/fail check with detailed result collections	"Create Customized Pass/Fail Check with Detailed Result Collections" on page 7-14

Add Custom Check to by Product Folder

This example shows how to add a custom check to a Model Advisor **By Product > Demo** subfolder. In this example, the customized check does not check model elements.

1 In your working folder, create the `sl_customization.m` file. This file registers and creates the check registration function `defineModelAdvisorChecks`, which in turn registers the check callback function `SimpleCallback`. The function `defineModelAdvisorChecks` uses a `ModelAdvisor.Root` object to define the check interface.

```
function sl_customization(cm)

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

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Example of a customized check';
rec.TitleTips = 'Added customized check to Product Folder';
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne');
mdladvRoot.publish(rec, 'Demo');

% --- creates SimpleCallback function
function result = SimpleCallback(system);
result={};
```

2 Close the Model Advisor and your model if either are open.

3 In the Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

4 From the MATLAB window, select **New** > **Simulink Model** to open a new Simulink model window.

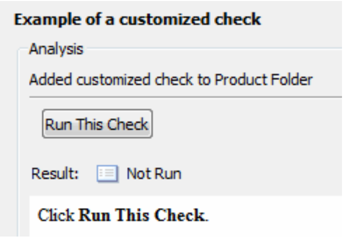
5 From the model window, select **Analysis** > **Model Advisor** > **Model Advisor** to open the Model Advisor.

6 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens.

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

8 In the left pane, expand the **By Product** folder to display the subfolders. The customized check **Example of a customized check** appears in the **By Product > Demo** subfolder.

   The following commands in the `sl_customization.m` file create the right pane of the Model Advisor.



Example of a customized check

Analysis

Added customized check to Product Folder

Result: Not Run

Click **Run This Check**.

```
rec.Title = 'Example of a customized check';
rec.TitleTips = 'Added customized check to Product Folder';
```

Create Customized Pass/Fail Check

This example shows how to create a Model Advisor pass/fail check. In this example, the Model Advisor checks Constant blocks. If a Constant blocks value is numeric, the check fails.

1 In your working folder, update the `sl_customization.m` file. This file registers and creates the check registration function `defineModelAdvisorChecks`, which also registers the check callback function `SimpleCallback`. The function `SimpleCallback` creates a check that finds Constant blocks that have numeric values. `SimpleCallback` uses the Model Advisor format template.

```
function sl_customization(cm)

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

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne')

mdladvRoot.publish(rec, 'Demo');

% --- SimpleCallback function that checks constant blocks
function result = SimpleCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
result    = {};

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');

ft = ModelAdvisor.FormatTemplate('ListTemplate');
ft.setInformation(['This check looks for constant blocks that'...
    ' use numeric values']);
if ~isempty(blk_with_value)
    ft.setSubResultStatusText(['Check has failed. The following '...
        'Constant blocks have numeric values:']);
    ft.setListObj(blk_with_value);
    ft.setSubResultStatus('warn');
    ft.setRecAction('Parameterize the constant block');
    mdladvObj.setCheckResultStatus(false);
else
    ft.setSubResultStatusText(['Check has passed. No constant blocks'...
        ' with numeric values were found.']);
    ft.setSubResultStatus('pass');
    mdladvObj.setCheckResultStatus(true);
end
ft.setSubBar(0);
result{end+1} = ft;
```

2 Close the Model Advisor and your model if either are open.

3 In the Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

4 From the MATLAB window, select **New > Simulink Model** to open a new Simulink model window.

5 In the Simulink model window, create two Constant blocks named Const_one and Const_1:

- Right-click the Const_one block, choose **Constant Parameters**, and assign a **Constant value** of one.
- Right-click the Const_1 block, choose **Constant Parameters**, and assign a **Constant value** of 1.
- Save your model as `example2_qs`

one
Const_one
1
Const_1

6 From the model window, select **Analysis** > **Model Advisor** > **Model Advisor** to open the Model Advisor.

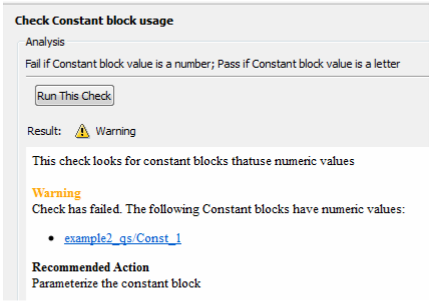
7 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens.

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

9 In the left pane, select **By Product > Demo > Check Constant block usage**.

10 Select **Run This Check**. The Model Advisor check fails for the Const_1 block and displays a **Recommended Action**.

The following commands in the `sl_customization.m` file create the right pane in the Model Advisor.



Check Constant block usage

Analysis

Fail if Constant block value is a number; Pass if Constant block value is a letter

Result: ⚠ Warning

This check looks for constant blocks thatuse numeric values

Warning

Check has failed. The following Constant blocks have numeric values:

- example2_qs/Const_1

Recommended Action

Parameterize the constant block

Check Constant block usage

```
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
```

Recommended Action

```
ft.setInformation(['This check looks for constant blocks that'...
    ' use numeric values']);
ft.setSubResultStatusText(['Check has failed. The following '...
        'Constant blocks have numeric values:']);
ft.setListObj(blk_with_value);
ft.setSubResultStatus('warn');
ft.setRecAction('Parameterize the constant block');
```

11 Follow the **Recommended Action** for fixing the failed Constant block. In the Model Advisor dialog box:

- Double-click the `example2_qs/Const_1` hyperlink.
- Change **Constant Parameters > Constant value** to two, or a nonnumeric value.
- Rerun the Model Advisor check. Both Constant blocks now pass the check.

Create Customized Pass/Fail Check with Fix Action

This example shows how to create a Model Advisor pass/fail check with a fix action. In this example, the Model Advisor checks Constant blocks. If a Constant block value is numeric, the check fails. The Model Advisor is also customized to create a fix action for the failed checks.

1 In your working folder, update the `sl_customization.m` file. This file contains three functions, each of which use the Model Advisor format template:

- `defineModelAdvisorChecks` — Defines the check, creates input parameters, and defines the fix action.
- `simpleCallback` — Creates the check callback function that finds Constant blocks with numeric values.
- `simpleActionCallback` — Creates the fix for Constant blocks that fail the check.

```
function sl_customization(cm)

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

% --- defineModelAdvisorChecks function
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('exampleCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if '...
    'Constant block value is a letter'];
rec.setCallbackFcn(@SimpleCallback,'None','StyleOne')

% --- input parameters
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Text entry example';
inputParam1.Value='VarNm';
inputParam1.Type='String';
inputParam1.Description='sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec.setInputParameters({inputParam1});

% -- set fix operation
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@simpleActionCallback);
myAction.Name='Fix Constant blocks';
myAction.Description=['Click the button to update all blocks with'...
    ' Text entry example'];
rec.setAction(myAction);

mdladvRoot.publish(rec, 'Demo');

% --- SimpleCallback function that checks constant blocks
function result = SimpleCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
result    = {};

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');

ft = ModelAdvisor.FormatTemplate('ListTemplate');
ft.setInformation(['This check looks for constant blocks that'...
    ' use numeric values']);
if ~isempty(blk_with_value)
    ft.setSubResultStatusText(['Check has failed. The following '...
        'Constant blocks have numeric values:']);
    ft.setListObj(blk_with_value);
    ft.setSubResultStatus('warn');
    ft.setRecAction('Parameterize the constant block');
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
else
    ft.setSubResultStatusText(['Check has passed. No constant blocks'...
        'with numeric values were found.']);
    ft.setSubResultStatus('pass');
    mdladvObj.setCheckResultStatus(true);
end
ft.setSubBar(0);
result{end+1} = ft;

% --- creates SimpleActionCallback function that fixes failed check
```

```
function result = simpleActionCallback(taskobj)
mdladvObj = taskobj.MAObj;
result    = {};

system = getfullname(mdladvObj.System);

% Get the string from the input parameter box.
inputParams = mdladvObj.getInputParameters;
textEntryEx = inputParams{1}.Value;

all_constant_blk=find_system(system,'LookUnderMasks','all',...
    'FollowLinks','on','BlockType','Constant');
blk_with_value=find_system(all_constant_blk,'RegExp','On','Value','^[0-9]');
ft = ModelAdvisor.FormatTemplate('TableTemplate');
% Define table col titles
ft.setColTitles({'Block','Old Value','New Value'})
for inx=1:size(blk_with_value)
   oldVal = get_param(blk_with_value{inx},'Value');
   ft.addRow({blk_with_value{inx},oldVal,textEntryEx});
   set_param(blk_with_value{inx},'Value',textEntryEx);
end

ft.setSubBar(0);
result = ft;
mdladvObj.setActionEnable(false);
```

2 Close the Model Advisor and your model if either are open.

3 At the command prompt, enter:

```
Advisor.Manager.refresh_customizations
```

4 From the Command Window, select **New > Simulink Model** to open a new model.

5 In the Simulink model window, create two Constant blocks named Const_one and Const_1:

- Right-click the Const_one block, choose **Constant Parameters**, and assign a **Constant value** of one.
- Right-click the Const_1 block, choose **Constant Parameters**, and assign a **Constant value** of 1.
- Save your model as `example3_qs`.

6 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.

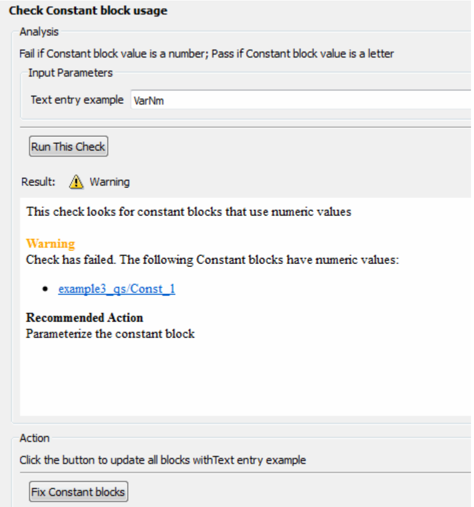
7 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens.

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

9 In the left pane, select **By Product > Demo > Check Constant block usage**.

10 Select **Run This Check**. The Model Advisor check fails for the Const_1 block.

The following commands in the sl_customization.m file create the right pane in the Model Advisor.



Fail if Constant block value is a number; Pass if Constant block value is a letter

Text entry example: VarNm

Result: ⚠ Warning

This check looks for constant blocks that use numeric values

Warning

Check has failed. The following Constant blocks have numeric values:

- example3_qs/Const_1

Recommended Action

Parameterize the constant block

Click the button to update all blocks withText entry example

Check Constant block usage

```
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Fail if Constant block value is a number; Pass if '...
    'Constant block value is a letter'];
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Text entry example';
inputParam1.Value='VarNm';
inputParam1.Type='String';
inputParam1.Description='sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec.setInputParameters({inputParam1});
```

Action

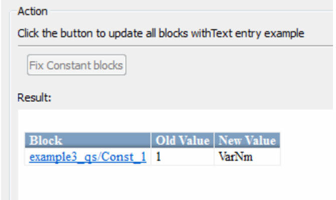
```
myAction.Name='Fix Constant blocks';
myAction.Description=['Click the button to update all blocks with'...
    'Text entry example'];
```

The Model Advisor box has a **Fix Constant blocks** button in the **Action** section of the Model Advisor dialog box.

11 In the Model Advisor dialog box, enter a nonnumeric value in the **Text entry example** parameter field in the **Analysis** section of the Model Advisor dialog box. In this example, the value is `VarNm`.

12 Click **Fix Constant blocks**. The Const_1 **Constant block value** changes from 1 to the nonnumeric value that you entered in step 10. The **Result** section of the dialog box lists the Old Value and New Value of the Const_1 block.

The following commands in the `sl_customization.m` file create the right pane in the Model Advisor.



Click the button to update all blocks withText entry example

Result:

Block	Old Value	New Value
example3_qs/Const_1	1	VarNm

Action

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');

ft.setColTitles({'Block','Old Value','New Value'})
for inx=1:size(blk_with_value)
   oldVal = get_param(blk_with_value{inx},'Value');
   ft.addRow({blk_with_value{inx},oldVal,textEntryEx});
   set_param(blk_with_value{inx},'Value',textEntryEx);
end
```

13 In the Model Advisor dialog box, click **Run This Check**. Both constant blocks now pass the check.

Create Customized Pass/Fail Check with Detailed Result Collections

This example shows how to create a Model Advisor check whose results are collected into a group, such as blocks in a subsystem that violate a check. When a check is not violated, the results contain the check description and result status. When a check is violated, the results contain the check description, result status, and the recommended action to fix the issue. This method is recommended when creating custom Model Advisor checks.

You can review results in the Model Advisor 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, results include a hyperlink to each model element that violates the check. Use these hyperlinks to easily locate areas in your model or subsystem.

To create a customized check with detailed result presented as a collection:

1. In your working folder, update the `sl_customization.m` file as shows in the example. This file contains three functions specific for creating a check whose results are presented on the Model Advisor as a collection:

   - `defineModelAdvisorChecks` — Defines the check and fix actions. In this function, the callback style is '`DetailStyle`', which is the Model Advisor format template that presents the results as a collection in the Model Advisor.
   - `SampleNewCheckStyleCallback` — Creates the check callback function that finds blocks whose name is not located below the block. The function uses name and value pairs to gather the results into collections. See “Check Callback Function for Detailed Result Collections” on page 7-63.
   - `sampleActionCB0` — Creates the fix for blocks whose name is not located below the block. In this example, it moves the name below the block. See “Action Callback Function for Detailed Result Collections” on page 7-66.

   ```
   function sl_customization(cm)

   % -----------------------------
   % Register custom checks
   cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

   % -----------------------------
   % Define Model Advisor check "Check whether block names appear
   % below blocks".
   function defineModelAdvisorChecks
   mdladvRoot = ModelAdvisor.Root;
   rec = ModelAdvisor.Check('com.mathworks.sample.Check0');
   rec.Title = 'Check whether block names appear below blocks
       (recommended check style)';
   rec.TitleTips = 'Example new style callback (recommended
       check style)';
   rec.setCallbackFcn(@SampleNewCheckStyleCallback,'None',
   ```

```
    'DetailStyle');
% set fix operation
myAction0 = ModelAdvisor.Action;
myAction0.setCallbackFcn(@sampleActionCB0);
myAction0.Name='Make block names appear below blocks';
myAction0.Description='Click the button to place block
    names below blocks';
rec.setAction(myAction0);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

% -----------------------------
% Callback function for check "Check whether block names appear
% below blocks".
function SampleNewCheckStyleCallback(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);

% -----------------------------
% Action callback function for check "Check whether block names
% appear below blocks".
function result = sampleActionCB0(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);
```

2 Close the Model Advisor and your model if either are open.

3 In the MATLAB command window, enter:

```
Advisor.Manager.refresh_customizations
```

4 From the MATLAB window, open model `sldemo_fuelsys`.

5 In the Simulink model window:

- In the top model, right-click the `Engine Speed` block and select **Rotate & Flip** > **Flip Block Name**.
- 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 From the model window, select **Analysis** > **Model Advisor** > **Model Advisor** to open the Model Advisor.

7 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens.

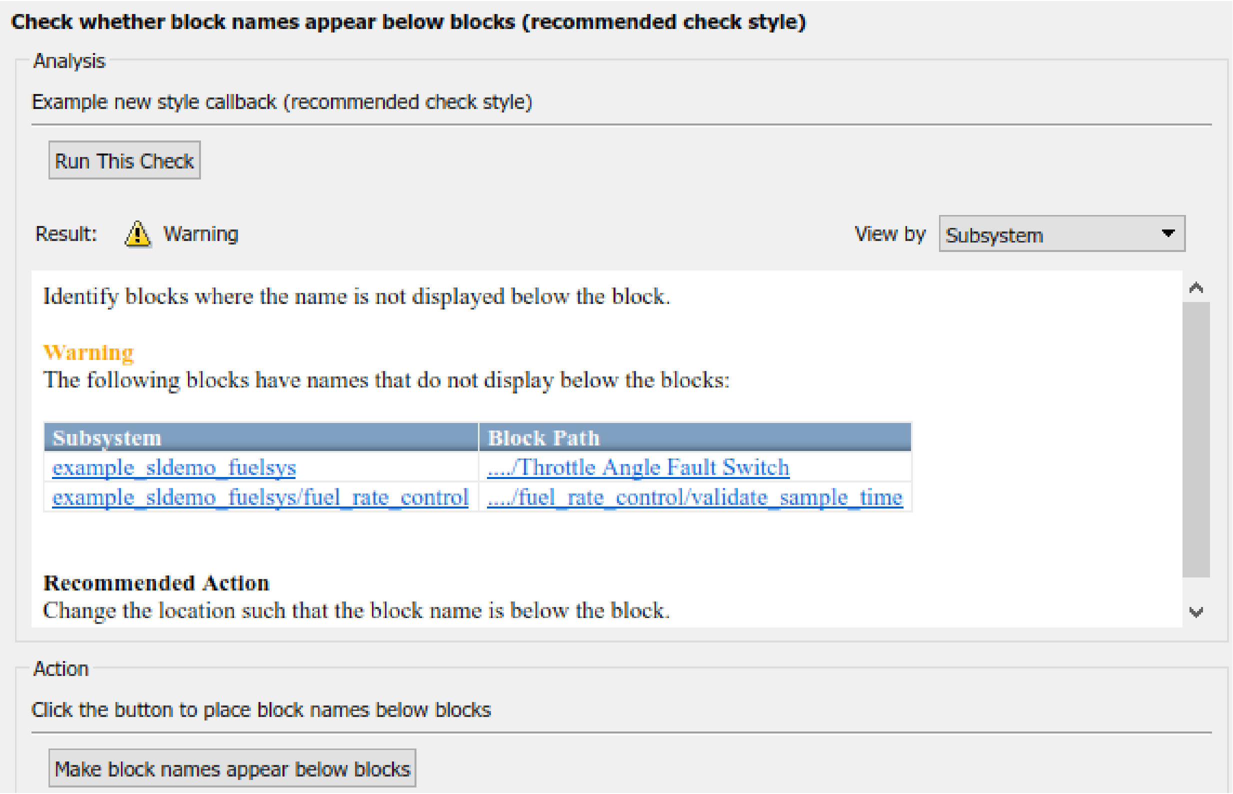
8 In the left pane, select **By Product > Demo > Check whether block names appear below blocks**.

Note If the **By Product** folder is not displayed in the Model Advisor window, select **SettingsPreferencesShow By Product Folder**.

9 Select **Run This Check**. The Model Advisor check fails for the blocks.

10 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,



Check whether block names appear below blocks (recommended check style)

Analysis

Example new style callback (recommended check style)

Result: ⚠ Warning

View by Subsystem

Identify blocks where the name is not displayed below the block.

Warning

The following blocks have names that do not display below the blocks:

Subsystem	Block Path
example_sldemo_fuelsys	..../Throttle Angle Fault Switch
example_sldemo_fuelsys/fuel_rate_control	..../fuel_rate_control/validate_sample_time

Recommended Action

Change the location such that the block name is below the block.

Action

Click the button to place block names below blocks

The following commands in the `sl_customization.m` file create the right pane in the Model Advisor.

Check title and subtitle

```
rec.Title = 'Check whether block names appear below blocks
    (recommended check style)';
rec.TitleTips = 'Example new style callback (recommended
    check style)';
```

Result

```
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.';
```

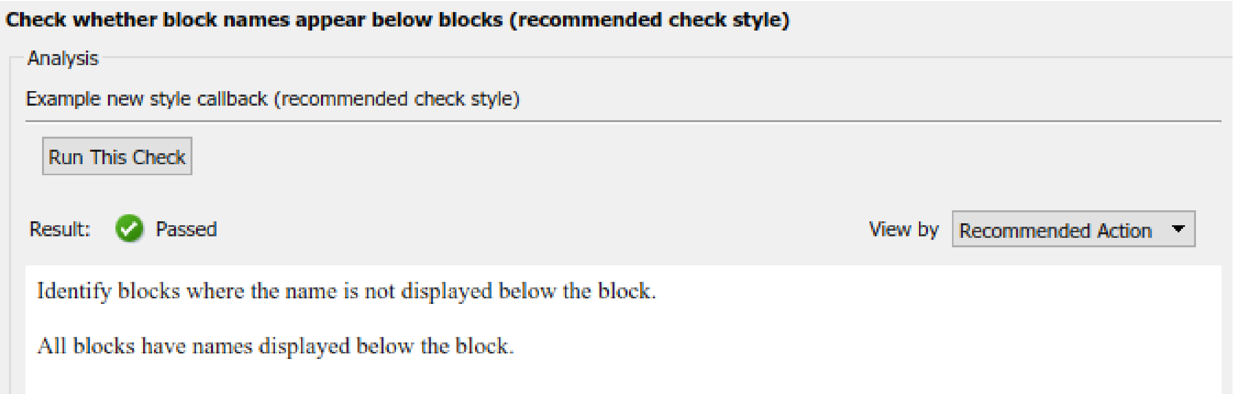
Action

```
myAction0.Name='Make block names appear below blocks';
myAction0.Description='Click the button to place block names
    below blocks';
```

11 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 greyed out after the violations are fixed.

12 Save the model and rerun the Model Advisor check. The check passes.



Check whether block names appear below blocks (recommended check style)

Analysis

Example new style callback (recommended check style)

Result: ✓ Passed

View by: Recommended Action

Identify blocks where the name is not displayed below the block.

All blocks have names displayed below the block.

The following commands in the `sl_customization.m` file create the right pane in the Model Advisor.

Result

```
ElementResults.Description = 'Identify blocks where the name is not
    displayed below the block.';
ElementResults.Status = 'All blocks have names displayed below the block.';
```

See Also

`ModelAdvisor.FormatTemplate` | `ModelAdvisor.Check` | ModelAdvisor.Check.CallbackContext | `ModelAdvisor.FormatTemplate`

More About

- “Register Checks” on page 7-43
- “Define Check Input Parameters” on page 7-52
- “Check Callback Function for Detailed Result Collections” on page 7-63
- “Action Callback Function for Detailed Result Collections” on page 7-66
- “Define, Configure, and Activate Variants” (Simulink)
- “Create and Validate Variant Configurations” (Simulink)
- “Represent Subsystem and Variant Models in Generated Code” (Embedded Coder)
- “Define the Compile Option for Custom Checks” on page 7-72

Create Check for Model Configuration Parameters

To verify the configuration parameters for your model, you can create a configuration parameter check.

Decide which configuration parameter settings to use for your model.

Guidelines	See
MathWorks Automotive Advisory Board (MAAB) Control Algorithm Modeling Guidelines	"MAAB Control Algorithm Modeling" (Simulink)
High-Integrity System Modeling Guideline	"High-Integrity System Modeling" (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 Data File for Diagnostics Pane Configuration Parameter Check

This example shows how to create a data file for **Diagnostics** pane model configuration parameter check that warns when:

- **Algebraic loop** is set to `none`
- **Minimize algebraic loop** is not set to `error`
- **Block Priority Violation** is not set to `error`

In the example, to create the data file, you use the `Advisor.authoring.generateConfigurationParameterDataFile` function.

At the command prompt, type `vdp`.

In the model window, select **Simulation > Model Configuration Parameters**. In the **Diagnostics** pane, set the configuration parameters as follows:

- **Algebraic loop** to none
- **Minimize algebraic loop** to error
- **Block Priority Violation** to error

Use Advisor.authoring.generateConfigurationParameterDataFile to create a data file specifying configuration parameter constraints in the **Diagnostics** pane. 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** (command line: ArtificialAlgebraicLoopMsg) configuration parameter tagging specifies a value of error with a fixvalue of error. When you run the configuration parameter check using ex_DataFile.xml, the check fails if the **Minimize algebraic loop** setting is not error. The check fix action modifies the setting to error.
- The **Block Priority Violation** (command line:BlockPriorityViolationMsg) configuration parameter tagging specifies a value of error with a fixvalue of error. When you run the configuration parameter check using ex_DataFile.xml, the check fails if the **Block Priority Violation** setting is not error. The check fix action modifies the setting to error.

In ex_DataFile.xml, edit the **Algebraic loop** (command line:AlgebraicLoopMsg) parameter tagging so that the check warns if the value is none. Because you are specifying a configuration parameter that you do not want, you need a NegativeModelParameterConstraint. 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 file. First, you register the check using an sl_customization file. Using `ex_DataFile.xml`, the check warns when:

- **Algebraic loop** is set to `none`
- **Minimize algebraic loop** is not set to `error`
- **Block Priority Violation** is not set to `error`

The check fix action modifies the **Minimize algebraic loop** and **Block Priority Violation** settings to `error`.

The check uses the `ex_DataFile.xml` data file created in “Create Data File for Diagnostics Pane Configuration Parameter Check” on page 7-21.

Close the Model Advisor and your model if either are open.

Use the following `sl_customization.m` file to specify and register check **Example: Check model configuration parameters**.

```
function sl_customization(cm)

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

% register items to factory group.
cm.addModelAdvisorTaskFcn(@defineModelAdvisorGroups);


%% defineModelAdvisorChecks
function defineModelAdvisorChecks

 rec = ModelAdvisor.Check('com.mathworks.Check1');
    rec.Title = 'Example: Check model configuration parameters';
    rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
                                    (system)), 'None', 'StyleOne');
    rec.TitleTips = 'Example check for model configuration parameters';

    % --- data file input parameters
    rec.setInputParametersLayoutGrid([1 1]);
    inputParam1 = ModelAdvisor.InputParameter;
    inputParam1.Name = 'Data File';
    inputParam1.Value = 'ex_DataFile.xml';
    inputParam1.Type = 'String';
    inputParam1.Description = 'Name or full path of XML data file.';
    inputParam1.setRowSpan([1 1]);
    inputParam1.setColSpan([1 1]);
    rec.setInputParameters({inputParam1});

    % -- set fix operation
    act = ModelAdvisor.Action;
    act.setCallbackFcn(@(task)(Advisor.authoring.CustomCheck.actionCallback...
                                                 (task)));
    act.Name = 'Modify Settings';
    act.Description = 'Modify model configuration settings.';
    rec.setAction(act);

    mdladvRoot = ModelAdvisor.Root;
    mdladvRoot.register(rec);

%% defineModelAdvisorGroups
function defineModelAdvisorGroups
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group 1
rec = ModelAdvisor.FactoryGroup('com.mathworks.Test.factoryGroup');
rec.DisplayName='Example: My Group';
rec.addCheck('com.mathworks.Check1');
mdladvRoot.publish(rec);
```

Create the **Example: Check model configuration parameters**. At the command prompt, enter:

```
Advisor.Manager.refresh_customizations
```

At the command prompt, type `vdp`.

In the model window, select **Simulation** > **Model Configuration Parameters**. In the **Diagnostics** pane, to trigger check warnings, set the configuration parameters as follows:

- **Algebraic loop** to `none`
- **Minimize algebraic loop** to `warning`
- **Block Priority Violation** to `warning`

From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.

In the left pane, select **By Task > Example: My Group > Example: Check model configuration parameters**. In the right pane, **Data File** is set to `ex_DataFile.xml`.

Click **Run This Check**. The Model Advisor check warns that the configuration parameters are not set to the values specified in ex_DataFile.xml. For configuration parameters with positive constraint tagging (PositiveModelParameterConstraint), the recommended values are obtained from the value tagging. For configuration parameters with negative constraint tagging (NegativeModelParameterConstraint), the values not recommended are obtained from the value tagging.

- **Algebraic loop**(AlgebraicLoopMsg) - the ex_DataFile.xml tagging does not specify a fix action for AlgebraicLoopMsg. The subcheck passes only when the setting is not set to none.
- **Minimize algebraic loop**(ArtificialAlgebraicLoopMsg) - the ex_DataFile.xml tagging specifies a subcheck with a fix action for ArtificialAlgebraicLoopMsg that passes only when the setting is error. The fix action modifies the setting to error.
- **Block priority violation**(BlockPriorityViolationMsg) - the ex_DataFile.xml tagging specifies a subcheck with a fix action for BlockPriorityViolationMsg that does not pass when the setting is warning. The fix action modifies the setting to error.

In the **Action** section of the Model Advisor dialog box, click **Modify Settings**. Model Advisor updates the configuration parameters for **Block priority violation** and **Minimize algebraic loop**.

Run **By Task > Example: My Group > Example: Check model configuration parameters**. The check warns because **Algebraic loop** is set to none.

In the right pane of the Model Advisor window, use the Algebraic loop (AlgebraicLoopMsg) link to open the **Simulation** > **Model Configuration Parameters** > **Diagnostics**. Set **Algebraic loop** to warning or error.

Run **By Task > Example: My Group > Example: Check model configuration parameters**. The check passes.

Data File for Configuration Parameter Check

You use an XML data file to create a configuration parameter check. To create the data file, you can use Advisor.authoring.generateConfigurationParameterDataFile or manually create the file yourself. The data file contains tagging that specifies check behavior. Each model configuration parameter specified in the data file is a subcheck. The structure for the data file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.w3schools.com
MySchema.xsd">
    <messages>
        <Description>Description of check</Description>
        <PassMessage>Pass message</PassMessage>
        <FailMessage>Fail message</FailMessage>
        <RecommendedActions>Recommended action</RecommendedActions>
    </messages>
    <checkdata>
    <!--Command line name of configuration parameter-->
        <PositiveModelParameterConstraint>
            <parameter>Command-line name of configuration parameter</parameter>
            <value>Value that you want configuration parameter to have</value>
            <fixvalue>Specify value for a fix action</fixvalue>
            <dependson>ID of configuration parameter subcheck that must pass
                    before this subcheck runs</value>
        </PositiveModelParameterConstraint>
    <!-- Command line name of configuration parameter-->
        <NegativeModelParameterConstraint>
            <parameter>Command line name of configuration parameter</parameter>
            <value>Value that you do not want configuration parameter to have</value>
            <fixvalue>Specify value for a fix action</fixvalue>
            <dependson>ID of configuration parameter subcheck that must pass
                    before this subcheck runs</value>
        </NegativeModelParameterConstraint>
     </checkdata>
</customcheck>
```

The `<messages>` 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, NegativeModelParameterConstraint constraints warn when the configuration parameter settings for configuration parameter is not error or none .
		`<NegattiveModelParameterConstraint>`
		`    <value>error</value>`
		`    <value>none</value>`
		`</NegativeModelParameterConstraint>`

Type	Format	Example

Structure or object	`<value>`
	`<param1>xyz</param1>`
	`<param2>yza</param2>`
	`</value>`
		In this example, `PositiveModelParameterConstraint` constraints warn 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, `NegativeModelParameterConstraint` constraints warn 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, `NegativeModelParameterConstraint` constraints warn 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>`

<table>
<tr><th>Type</th><th>Format</th><th>Example</th></tr>
<tr><td>Structure or object</td><td><pre>
<fixvalue>
    <param1>xyz</param1>
    <param2>yza</param2>
</fixvalue>
</pre></td><td>In this example, the fix action tag specifies the new configuration parameter setting for a structure.<pre>
<PositiveModelParameterConstraint>
    <value>
        <double>a</double>
        <single>b</single>
    </value>
    <fixvalue>
        <double>c</double>
        <single>d</single>
    </fixvalue>
</PositiveModelParameterConstraint>
</pre></td></tr>
<tr><td>Array</td><td><pre>
<fixvalue>
    <element>value</element>
    <element>value</element>
</fixvalue>
</pre></td><td>In this example, the fix action tag specifies the new configuration parameter setting for an array.<pre>
<NegativeModelParameterConstraint>
    <value>
        <element>A</element>
        <element>B</element>
    </value>
    <fixvalue>
        <element>C</element>
        <element>D</element>
    </fixvalue>
</NegativeModelParameterConstraint>
</pre></td></tr>
</table>

Type	Format	Example

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 Checks for Supported or Unsupported Blocks and Parameters

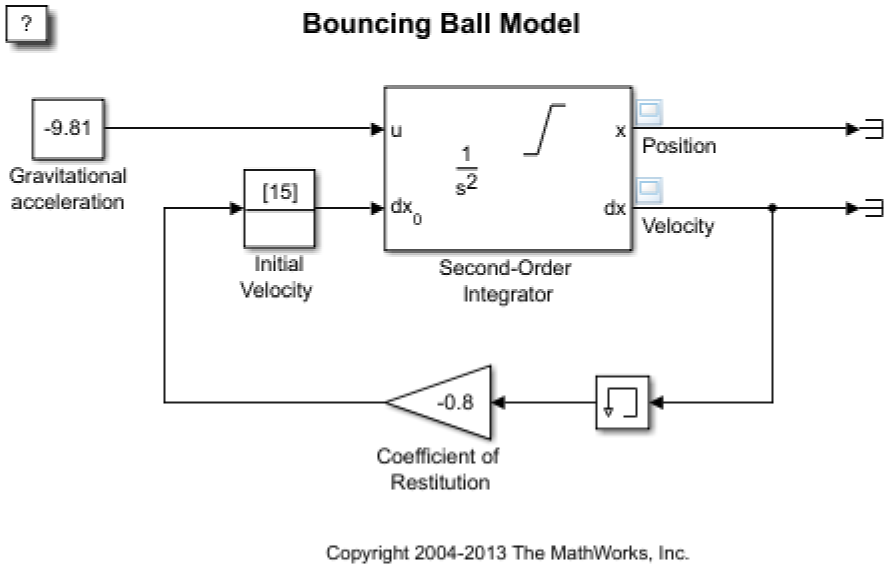
For modeling guidelines, such as MAAB or MISRA, that require you to use a subset of block or parameter values, you can create Model Advisor checks in which you specify these constraints:

- 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. To open the Model Advisor, from the model window, select **Analysis > Model Advisor > 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 Model Compliance by Using the Model Advisor” on page 3-2.

See Also

Advisor.authoring.generateBlockConstraintsDataFile | NegativeBlockParameterConstraint | NegativeBlockTypeConstraint | NegativeModelParameterConstraint | PositiveBlockParameterConstraint | PositiveBlockTypeConstraint | PositiveModelParameterConstraint

Register Checks

Create sl_customization Function

To add checks to the Model Advisor, on your MATLAB path, in the `sl_customization.m` file, create the `sl_customization()` function.

Tip

- You can have more than one `sl_customization.m` file on your MATLAB path.
- Do not place an `sl_customization.m` file that customizes checks and folders in the Model Advisor in your root MATLAB folder or its subfolders, except for the *`matlabroot`*`/work` folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.

The `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. Use these methods to register customizations specific to your application, as described in the following sections.

Register Checks

To register custom checks, the customization manager includes the following method:

- `addModelAdvisorCheckFcn (@`*`checkDefinitionFcn`*`)`

  Registers the checks that you define in *`checkDefinitionFcn`* to the **By Product** folder of the Model Advisor.

  The *`checkDefinitionFcn`* argument is a handle to the function that defines custom checks that you want to add to the Model Advisor as instances of the `ModelAdvisor.Check` class.

This example shows how to register custom checks:

```
function sl_customization(cm)

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

Note If you add custom tasks and folders within the `sl_customization.m` file, include methods for registering the tasks and folders in the `sl_customization` function.

See Also

`ModelAdvisor.Check`

Related Examples

- Registering Tasks and Folders on page 8-12

More About

- “Define Custom Checks” on page 7-48

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.

Argument	I/O Type	Data Type	Description
`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 Model Advisor Check Browser. However, if you use a process callback function in a `sl_customization` file to hide checks and folders, the Model Advisor Configuration Editor and Model Advisor Check Browser do not display the hidden checks and folders. For a complete list of checks and folders, remove process callback functions and update the Simulink environment.
- 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

“Create Model Advisor Checks Workflow” on page 7-2 | “Register Checks” on page 7-43 | “Organize Customization File Checks and Folders” on page 8-11 | “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5

Define Custom Checks

About Custom Checks

You can create a custom check to use in the Model Advisor. Creating custom checks provides you with the ability to specify which conditions and configuration settings the Model Advisor reviews.

You define custom checks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Check` class. Define one instance of this class for each custom check that you want to add to the Model Advisor, and register the custom check.

Tip You can add a check to multiple folders by creating a task.

Contents of Check Definitions

When you define a Model Advisor check, it contains the information listed in the following table.

Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this id to access the check.
Handle to check callback function (required)	Function that specifies the contents of a check.
Check name (recommended)	Creates a name for the check that the Model Advisor displays.
Model compiling (optional)	Specifies whether the model is compiled for check analysis.
Check properties (optional)	Creates a user interface with the check. When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.
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 fixing action.

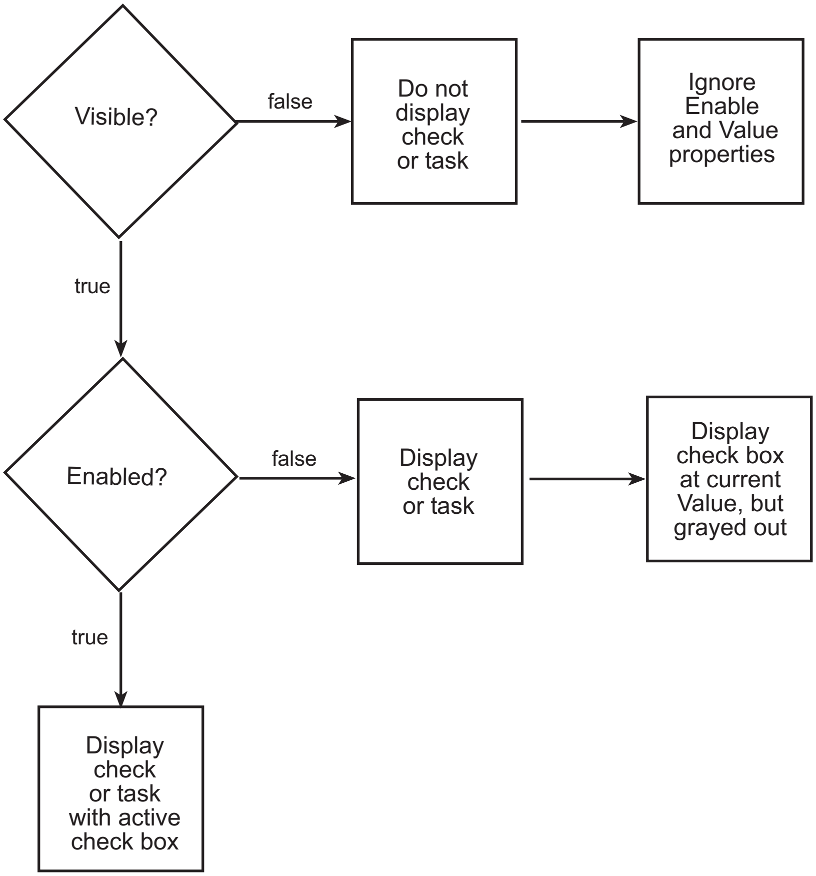
Contents	Description
Explore Result button (optional)	Adds the **Explore Result** button that the user clicks to open the Model Advisor Result Explorer.

Display and Enable Checks

You can create a check and specify how it appears in the Model Advisor. You can define when to display a check, or whether a user can select or clear a check using the `Visible`, `Enable`, and `Value` properties of the `ModelAdvisor.Check` class.

Note When adding checks to the Model Advisor as tasks, specify these properties in the `ModelAdvisor.Task` class. If you specify the properties in both `ModelAdvisor.Check` and `ModelAdvisor.Task`, the `ModelAdvisor.Task` properties take precedence, except for the `Visible` and `LicenseName` properties.

The following chart illustrates how the `Visible`, `Enable`, and `Value` properties interact.


Visible?
false
Do not display check or task
Ignore Enable and Value properties
true
Enabled?
false
Display check or task
Display check box at current Value, but grayed out
true
Display check or task with active check box

Define Where Custom Checks Appear

Specify where the Model Advisor places custom checks using the following guidelines:

- To place a check in a new folder in the **Model Advisor** root, use the `ModelAdvisor.Group` class.
- To place a check in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.
- To place a check in the **By Product** folder, use the `ModelAdvisor.Root.publish` method. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings** > **Preferences** dialog box.

Check Definition Function

This example shows a function that defines the custom checks associated with the callback functions described in “Create Callback Functions and Results” on page 7-56 and the model compile options described in “Define the Compile Option for Custom Checks” on page 7-72. The Model Advisor compiles and simulates the model; the check definition function returns a cell array of custom checks to be added to the Model Advisor.

The check definitions in the example use the tasks described in Defining Custom Groups on page 8-13.

```
% Defines custom Model Advisor checks
function defineModelAdvisorChecks

% Sample Check 0: Check whose Results are Viewed as Detailed Result Collections
rec = ModelAdvisor.Check('com.mathworks.sample.Check0');
rec.Title = 'Check whether block names appear below blocks (recommended check style)';
rec.TitleTips = 'Example new style callback (recommended check style)';
rec.setCallbackFcn(@SampleNewCheckStyleCallback,'None','DetailStyle');
% set fix operation
myAction0 = ModelAdvisor.Action;
myAction0.setCallbackFcn(@sampleActionCB0);
myAction0.Name='Make block names appear below blocks';
myAction0.Description='Click the button to place block names below blocks';
rec.setAction(myAction0);
mdladvRoot.register(rec);

% Sample check 1: Informational check
rec = ModelAdvisor.Check('mathworks.example.configManagement');
rec.Title = 'Informational check for model configuration management';
setCallbackFcn(rec, @modelVersionChecksumCallbackUsingFT,'None','StyleOne');
rec.CallbackContext = 'PostCompile';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample check 2: Basic Check with Pass/Fail Status
rec = ModelAdvisor.Check('mathworks.example.unconnectedObjects');
rec.Title = 'Check for unconnected objects';
setCallbackFcn(rec, @unconnectedObjectsCallbackUsingFT,'None','StyleOne');
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);

% Sample Check 3: Check with Subchecks and Actions
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
rec.Title = 'Check safety-related optimization settings';
setCallbackFcn(rec, @OptmizationSettingCallback,'None','StyleOne');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
setCallbackFcn(modifyAction, @modifyOptmizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
                            ' settings that can impact safety.'];
```

```
modifyAction.Enable = true;
setAction(rec, modifyAction);
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);
```

Define Check Input Parameters

With input parameters, you can request input before running the check. Define input parameters using the `ModelAdvisor.InputParameter` class inside a custom check function. You must define one instance of this class for each input parameter that you want to add to a Model Advisor check.

Specify the layout of input parameters with the following methods.

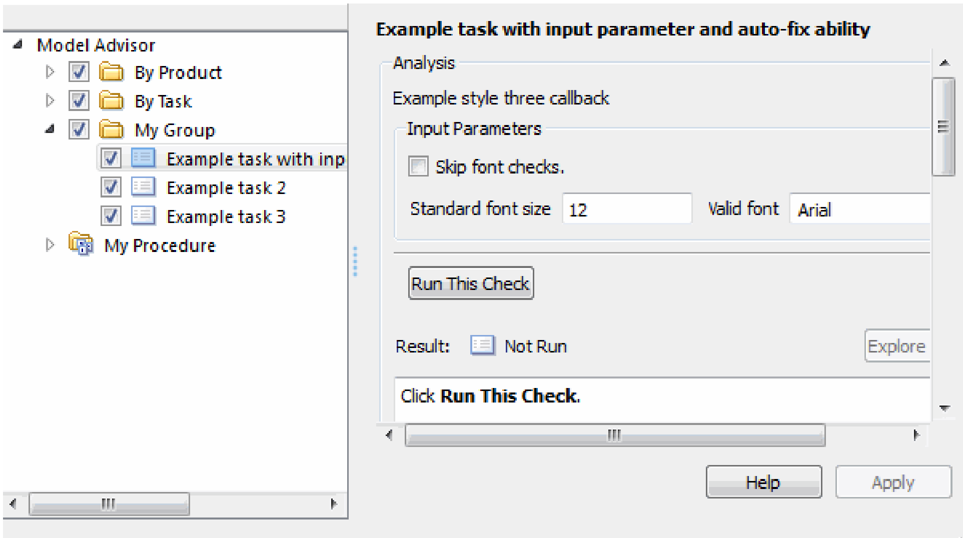
Method	Description
`ModelAdvisor.Check.setInputParametersLayoutGrid`	Specifies the size of the input parameter grid.
`ModelAdvisor.InputParameter.setRowSpan`	Specifies the number of rows the parameter occupies in the Input Parameter layout grid.
`ModelAdvisor.InputParameter.setColSpan`	Specifies the number of columns the parameter occupies in the Input Parameter layout grid.

This example shows how to define input parameters that you add to a custom check. You must include input parameter definitions inside a custom check definition. The following code, when included in a custom check definition, creates three input parameters.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.setInputParametersLayoutGrid([3 2]);
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
```

```
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
```

The Model Advisor displays these input parameters in the right pane, in an **Input Parameters** box.



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

Result: Not Run

Click **Run This Check**.

Define Model Advisor Result Explorer Views

A list view provides a way for users to fix check warnings and failures using the Model Advisor Result Explorer. Creating a list view allows you to:

- Add the **Explore Result** button to the custom check in the Model Advisor window.
- Provide the information to populate the Model Advisor Result Explorer.

This example shows how to define list views. You must make the **Explore Result** button visible using the `ModelAdvisor.Check.ListViewVisible` property inside a custom check function, and include list view definitions inside a check callback function. You must define one instance of this class for each list view that you want to add to a Model Advisor Result Explorer window.

The following code, when included in a check definition function, adds the **Explore Result** button to the check in the Model Advisor.

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
% add 'Explore Result' button
rec.ListViewVisible = true;
```

The following code, when included in a check callback function, provides the information to populate the Model Advisor Result Explorer.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);

% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object')';
myLVParam.Attributes = {'FontName'}; % name is default property
mdladvObj.setListViewParameters({myLVParam});
```

Define Check Actions

An action provides a way for you to specify an action that the Model Advisor performs to fix a Model Advisor check. When you define an action, the Model Advisor window includes an **Action** box below the **Analysis** box.

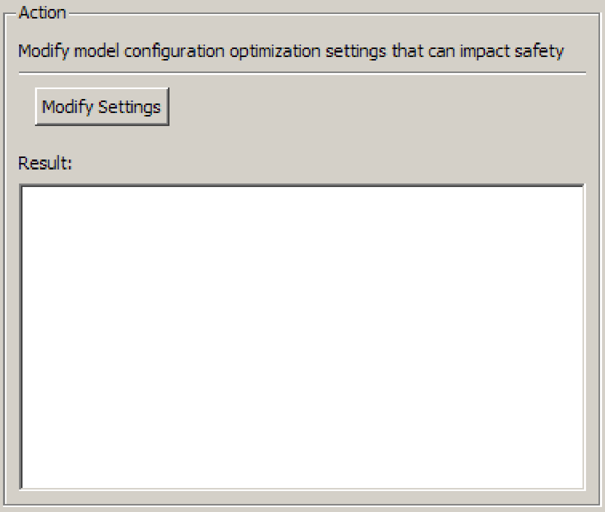
You define actions using the `ModelAdvisor.Action` class inside a custom check function. You must define:

- One instance of this class for each action that you want to take.
- One action callback function for each action.

This example shows the information you need to populate the **Action** box in the Model Advisor. Include this in the check definition function.

```
rec = ModelAdvisor.Check('mathworks.example.optimizationSettings');
% Define an automatic fix action for this check
modifyAction = ModelAdvisor.Action;
modifyAction.setCallbackFcn(@modifyOptmizationSetting);
modifyAction.Name = 'Modify Settings';
modifyAction.Description = ['Modify model configuration optimization' ...
                            ' settings that can impact safety'];
modifyAction.Enable = true;
rec.setAction(modifyAction);
```

The Model Advisor, in the right pane, displays an **Action** box.



Modify model configuration optimization settings that can impact safety

See Also

`ModelAdvisor.Action` | `ModelAdvisor.Check` | `ModelAdvisor.FactoryGroup` | `ModelAdvisor.Group` | `ModelAdvisor.InputParameter` | `ModelAdvisor.Root.publish` | `ModelAdvisor.Task`

Related Examples

- “Organize Customization File Checks and Folders” on page 8-11

More About

- “Batch-Fix Warnings or Failures” (Simulink)
- “Create Callback Functions and Results” on page 7-56
- “Define the Compile Option for Custom Checks” on page 7-72
- Defining Custom Groups on page 8-13
- “Register Checks” on page 7-43

Create Callback Functions and Results

About Callback Functions

A callback function specifies the actions that the Model Advisor performs on a model or subsystem, based on the check or action that the user runs. You must create a callback function for each custom check and action so that the Model Advisor can execute the function when you run the check. All types of callback functions provide one or more return arguments for displaying the results after executing the check or action. In some cases, return arguments are character vectors or cell arrays of character vectors that support embedded HTML tags for text formatting.

Action	More Information
Create an *informational callback function* for a custom check that finds and displays the model configuration and checksum information.	“Informational Check Callback Function” on page 7-57
Create a *simple callback function* that indicates if the model passed a check, or to recommend fixing the issue.	“Simple Check Callback Function” on page 7-58
Create *detailed check callback function* to return and organize results as strings in a layered, hierarchical fashion.	“Detailed Check Callback Function” on page 7-59
Create a callback function that automatically displays hyperlinks for every object returned by the check.	“Check Callback Function with Hyperlinked Results” on page 7-60
Create a callback function that collects results into a group, such as blocks in a subsystem that violate a check. These results are presented on the Model Advisor user interface by using report styles that are viewed by recommended action, subsystem, or block.	“Check Callback Function for Detailed Result Collections” on page 7-63

Action	More Information
Create an *action callback function* that specifies the actions that the Model Advisor performs on a model or subsystem when you click the action button.	“Action Callback Function” on page 7-65
Create a callback function for a custom check with two subchecks.	“Check With Subchecks and Actions” on page 7-66
Create a callback function for a custom basic check with pass/fail status.	“Basic Check with Pass/Fail Status” on page 7-68

Informational Check Callback Function

This example shows how to create a callback function for a custom informational check that finds and displays the model configuration and checksum information. The informational check uses the Result Template API to format the check result.

An informational check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.

An informational check does not include the following items in the results:

- The check status. The Model Advisor displays the overall check status, but the status is not in the result.
- A description of the status.
- The recommended action to take when the check does not pass.
- Subcheck results.
- A line below the results.

```
% Sample Check 1 Callback Function: Informational Check
% Find and display model configuration and checksum information
% Informational checks do not have a passed or warning status in the results

function resultDescription = modelVersionChecksumCallbackUsingFT(system)
resultDescription = [];
system = getfullname(system);
model = bdroot(system);

% Format results in a list using Model Advisor Result Template API
ft = ModelAdvisor.FormatTemplate('ListTemplate');
```

```
% Add See Also section for references to standards
docLinkSfunction{1}     = {['IEC 61508-3, Table A.8 (5)' ...
                            ' ''Software configuration management'' ']};
setRefLink(ft,docLinkSfunction);

% Description of check in results
desc = 'Display model configuration and checksum information.';
% If running the Model Advisor on a subsystem, add note to description
if strcmp(system, model) == false
    desc = strcat(desc, ['<br/>NOTE: The Model Advisor is reviewing a' ...
        ' sub-system, but these results are based on root-level settings.']);
end
setCheckText(ft, desc);

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% If err, use these values
mdlver = 'Error - could not retrieve Version';
mdlauthor = 'Error - could not retrieve Author';
mdldate = 'Error - could not retrieve Date';
mdlsum = 'Error - could not retrieve CheckSum';

% Get model configuration and checksum information
try
    mdlver = get_param(model,'ModelVersion');
    mdlauthor = get_param(model,'LastModifiedBy');
    mdldate = get_param(model,'LastModifiedDate');
    mdlsum = Simulink.BlockDiagram.getChecksum(model);
    mdlsum = [num2str(mdlsum(1)) ' ' num2str(mdlsum(2)) ' ' ...
                     num2str(mdlsum(3)) ' ' num2str(mdlsum(4))];
    mdladvObj.setCheckResultStatus(true); % init to true
catch err
    mdladvObj.setCheckResultStatus(false);
    setSubResultStatusText(ft,err.message);
    resultDescription{end+1} = ft;
    return
end

% Display the results
lbStr ='<br/>';
resultStr = ['Model Version: ' mdlver lbStr 'Author: ' mdlauthor lbStr ...
    'Date: ' mdldate lbStr 'Model Checksum: ' mdlsum];
setSubResultStatusText(ft,resultStr);

% Informational checks do not have subresults, suppress line
setSubBar(ft,false);
resultDescription{end+1} = ft;
```

Simple Check Callback Function

This example shows how to create a simple check callback function. Use a simple check callback function with results formatted using the Result Template API to indicate whether the model passed or failed the check, or to recommend fixing an issue. The keyword for this callback function is `StyleOne`. The check definition requires this keyword.

The check callback function takes the following arguments.

Argument	I/O Type	Description
`system`	Input	Path to the model or subsystem analyzed by the Model Advisor.
`result`	Output	MATLAB character vector that supports Model Advisor Formatting API on page 7-86 calls or embedded HTML tags for text formatting.

Detailed Check Callback Function

This example shows how to create a detailed check callback function. Use the detailed check callback function to return and organize results as strings in a layered, hierarchical fashion. The function provides two output arguments so you can associate text descriptions with one or more paragraphs of detailed information. The keyword for the detailed callback function is `StyleTwo`. The check definition requires this keyword.

The detailed callback function takes the following arguments.

Argument	I/O Type	Description
`system`	Input	Path to the model or system analyzed by the Model Advisor.
`ResultDescription`	Output	Cell array of MATLAB character vectors that supports Model Advisor Formatting API on page 7-86 calls or embedded HTML tags for text formatting. The Model Advisor concatenates the `ResultDescription` character vector with the corresponding array of `ResultDetails` character vectors.
`ResultDetails`	Output	Cell array of cell arrays, each of which contains one or more character vectors.

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

This example shows a detailed check callback function that checks optimization settings for simulation and code generation.

```
% -----------------------------
% Sample StyleTwo callback function, used for check "Check model optimization settings"
% Please refer to Model Advisor API document for more details.
% -----------------------------
function [ResultDescription, ResultDetails] = SampleStyleTwoCallback(system)
ResultDescription ={};
ResultDetails ={};

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

% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '...
    'optimization settings:']);
if strcmp(get_param(model,'BlockReduction'),'off');
    ResultDetails{end+1}     = {ModelAdvisor.Text(['It is recommended to '...
        'turn on Block reduction optimization option.',{'italic'}])};
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
else
    ResultDetails{end+1}     = {ModelAdvisor.Text('Passed',{'pass'})};
end

% Check code generation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check code generation '...
    'optimization settings:']);
ResultDetails{end+1}  = {};
if strcmp(get_param(model,'LocalBlockOutputs'),'off');
    ResultDetails{end}{end+1}     = ModelAdvisor.Text(['It is recommended to'...
        ' turn on Enable local block outputs option.',{'italic'}]);
    ResultDetails{end}{end+1}     = ModelAdvisor.LineBreak;
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if strcmp(get_param(model,'BufferReuse'),'off');
    ResultDetails{end}{end+1}     = ModelAdvisor.Text(['It is recommended to'...
        ' turn on Reuse block outputs option.',{'italic'}]);
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if isempty(ResultDetails{end})
    ResultDetails{end}{end+1}     = ModelAdvisor.Text('Passed',{'pass'});
end
```

Check Callback Function with Hyperlinked Results

This example shows how to create a callback function with hyperlinked results. This callback function automatically displays hyperlinks for every object returned by the check so that you can easily locate problem areas in your model or subsystem. The keyword for this type of callback function is `StyleThree`. The check definition requires this keyword.

This callback function takes the following arguments.

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

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

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

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

```
-----------------------------
% Sample StyleThree callback function, used for check "Check Simulink block font".
% Please refer to Model Advisor API document for more details.
% -----------------------------
function [ResultDescription, ResultDetails] = SampleStyleThreeCallback(system)
ResultDescription ={};
ResultDetails ={};

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
```

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

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

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

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

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font other than ' regularFontName ': ']);
    ResultDetails{end+1}     = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult,'object')';
    myLVParam.Attributes = {'FontName'}; % name is default property
    mdladvObj.setListViewParameters({myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font names '...
        'are identical.']);
    ResultDetails{end+1}     = {};
end

% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
```

```
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font size for blocks to ensure uniform appearance of model. '...
        'The following blocks use a font size other than ' num2str(regularFontSize) ': ']);
    ResultDetails{end+1}     = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font size blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult,'object')';
    myLVParam.Attributes = {'FontSize'}; % name is default property
    mdladvObj.setListViewParameters...
    ({mdladvObj.getListViewParameters{:}, myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font sizes '...
    'are identical.']);
    ResultDetails{end+1}     = {};
end

mdladvObj.setActionEnable(needEnableAction);
mdladvObj.setCheckErrorSeverity(1);
```

In the Model Advisor, if you run **Example task with input parameter and auto-fix ability** for the `slvnvdemo_mdladv` model, you can view the hyperlinked results. Clicking the first hyperlink, `slvnvdemo_mdladv/Input`, displays the Simulink model with the Input block highlighted.

Check Callback Function for Detailed Result Collections

This example shows a check callback function that creates result detail objects that are collected into a group, such as blocks in a subsystem that violate a check. When a check is not violated, the result details contain the check description and result status. When a check is violated, the result details contain the check description, result status, and the recommended action to fix the issue.

The keyword for this callback function is `DetailStyle`. The check definition requires this keyword. See “Check Definition Function” on page 7-51.

The callback function takes the arguments listed in the table.

Argument	I/O Type	Description
`system`	Input	Path to the model or system analyzed by the Model Advisor.
`CheckObj`	Input	`ModelAdvisor.Check` object for the check.

In this example, the callback function reviews the model and identifies blocks whose name is not located below the block. It uses name and value pairs to gather the results into collections.

```
% -----------------------------
% Sample new check style callback function, used for check "Check whether block names appear below blocks".
% Please refer to Model Advisor API document for more details.
% -----------------------------
function SampleNewCheckStyleCallback(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);
```

In the Model Advisor, if you run **Check whether block names appear below blocks (recommended check style)** for the `slvnvdemo_mdladv` model, you can view the 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 there are check violations, click the hyperlink to easily review the issues in your model or subsystem. To create a check using this callback function, see “Create Customized Pass/Fail Check with Detailed Result Collections” on page 7-14.

Action Callback Function

This example shows how to create an action callback function. An action callback function specifies the actions that the Model Advisor performs on a model or subsystem when the user clicks the action button. You must create one callback function for the action that you want to take.

The action callback function takes the following arguments.

Argument	I/O Type	Description
`taskobj`	Input	The `ModelAdvisor.Task` object for the check that includes an action definition.
`result`	Output	MATLAB character vector that supports Model Advisor Formatting API on page 7-86 calls or embedded HTML tags for text formatting.

This example shows an action callback function that fixes the optimization settings that the Model Advisor reviews as defined in “Check With Subchecks and Actions” on page 7-66.

```
% Sample Check 3 Action Callback Function: Check with Subresults and Actions
% Fix optimization settings
function result = modifyOptmizationSetting(taskobj)
% Initialize variables
result =  ModelAdvisor.Paragraph();
mdladvObj = taskobj.MAObj;
system = bdroot(mdladvObj.System);

% 'Block reduction' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'BlockReduction'),'off')
    set_param(system,'BlockReduction','off');
    result.addItem(ModelAdvisor.Text( ...
        'Cleared the ''Block reduction'' check box.',{'Pass'}));
    result.addItem(ModelAdvisor.LineBreak);
end
% 'Conditional input branch execution' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    set_param(system,'ConditionallyExecuteInputs','off');
    result.addItem(ModelAdvisor.Text( ...
       'Cleared the ''Conditional input branch execution'' check box.', ...
   {'Pass'}));
end
```

Action Callback Function for Detailed Result Collections

This example shows the action callback function for check results that are collected into a group, such as blocks in a subsystem that violate a check. From the Model Advisor, you can use this functionality to fix issues flagged by the check.

```
% -----------------------------
% Sample Check 0 Action Callback Function: Check whose Results are Viewed as Detailed Result Collections
% please refer to Model Advisor API document for more details.
% -----------------------------
function result = sampleActionCB0(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);
```

In the Model Advisor, open the `slvnvdemo_mdladv` model. Right-click on a block and select **Rotate & Flip > Flip Block Name**. When you run **Check whether block names appear below blocks (recommended check style)**, the check fails.

You can fix the failed 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 now greyed out.

To create a check using this action callback function, see “Create Customized Pass/Fail Check with Detailed Result Collections” on page 7-14.

Check With Subchecks and Actions

This example shows how to create a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting, and the second reviews the **Conditional input branch execution** optimization setting.

A check with subchecks includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.
- Recommended actions to take when the subcheck does not pass.
- A line between the subcheck results.

```
% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
function ResultDescription = OptmizationSettingCallback(system)
% Initialize variables
system =getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};
system = bdroot(system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,['Check model configuration for optimization settings that'...
    'can impact safety.']);

% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction setting');
setInformation(ft1,'Check whether the ''Block reduction'' check box is cleared.');
% Add See Also section with references to applicable standards
docLinks{1}     = {['Reference DO-178B Section 6.3.4e - Source code ' ...
    'is traceable to low-level requirements']};
% Review 'Block reduction' optimization
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is cleared.');
    ResultStatus = true;
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is selected.');
    setRecAction(ft1,['Clear the ''Optimization > Block reduction''' ...
        ' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check whether the ''Conditional input branch execution'''...
    ' check box is cleared.'])
% Add See Also section and references to applicable standards
docLinks{1} = {['Reference DO-178B Section 6.4.4.2 - Test coverage ' ...
    'of software structure is achieved']};
setRefLink(ft2,docLinks);

% Last subcheck, suppress line
setSubBar(ft2,false);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution''' ...
        'check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution'''...
        ' check box is selected.']);
    setRecAction(ft2,['Clear the ''Optimization > Conditional input branch ' ...
        'execution'' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);
```

Basic Check with Pass/Fail Status

This example shows a callback function for a custom basic check that finds and reports unconnected lines, input ports, and output ports.

A basic check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.

- The status of the check.
- A description of the status.
- Results for the check.
- The recommended actions to take when the check does not pass.

A basic check does not include the following items in the results:

- Subcheck results.
- A line below the results.

```
% Sample Check 2 Callback Function: Basic Check with Pass/Fail Status
% Find and report unconnected lines, input ports, and output ports
function ResultDescription = unconnectedObjectsCallbackUsingFT(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% Initialize variables
mdladvObj.setCheckResultStatus(false);
ResultDescription ={};
ResultStatus = false; % Default check status is 'Warning'
system = getfullname(system);
isSubsystem = ~strcmp(bdroot(system), system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object
ft = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
if isSubsystem
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                                        'output ports in the subsystem.'];
else
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                                        'output ports in the model.'];
end
setCheckText(ft,checkDescStr);

% Add See Also section with references to applicable standards
checkStdRef = 'IEC 61508-3, Table A.3 (3) ''Language subset'' ';
docLinkSfunction{1}     = {checkStdRef};
setRefLink(ft,docLinkSfunction);

% Basic checks do not have subresults, suppress line
setSubBar(ft,false);

% Check for unconnected lines, inputs, and outputs
sysHandle = get_param(system, 'Handle');
uLines = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'line', ...
    'Connected', 'off');
uPorts = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'port', ...
    'Line', -1);

% Use parents of port objects for the correct highlight behavior
if ~isempty(uPorts)
    for i=1:length(uPorts)
        uPorts(i) = get_param(get_param(uPorts(i), 'Parent'), 'Handle');
    end
end

% Create cell array of unconnected object handles
modelObj = {};
searchResult = union(uLines, uPorts);
for i = 1:length(searchResult)
    modelObj{i} = searchResult(i);
end

% No unconnected objects in model
% Set result status to 'Pass' and display text describing the status
if isempty(modelObj)
    setSubResultStatus(ft,'Pass');
    if isSubsystem
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this subsystem.']);
    else
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this model.']);
    end
    ResultStatus            = true;
% Unconnected objects in model
% Set result status to 'Warning' and display text describing the status
else
    setSubResultStatus(ft,'Warn');
    if ~isSubsystem
        setSubResultStatusText(ft,['The following lines, input ports, ' ...
         'or output ports are not properly connected in the system: ' system]);
    else
        setSubResultStatusText(ft,['The following lines, input ports, or ' ...
         'output ports are not properly connected in the subsystem: ' system]);
    end
    % Specify recommended action to fix the warning
    setRecAction(ft,'Connect the specified blocks.');
    % Create a list of handles to problem objects
    setListObj(ft,modelObj);
    ResultStatus = false;
end
% Pass the list template object to the Model Advisor
ResultDescription{end+1} = ft;
% Set overall check status
mdladvObj.setCheckResultStatus(ResultStatus);
```

See Also

`ModelAdvisor.Check` | `ModelAdvisor.FormatTemplate` | `ModelAdvisor.Task`

More About

- Defining Custom Groups on page 8-13
- “Define Custom Checks” on page 7-48
- “Format Check Results” on page 7-86
- “Register Checks” on page 7-43

Define the Compile Option for Custom 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 to ensure that the correct 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';
```

In this situation, the Model Advisor simulates the model. The Model Advisor does not compile the model.

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.

This example shows a check definition that does not require the model to be compiled.

```
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');
```

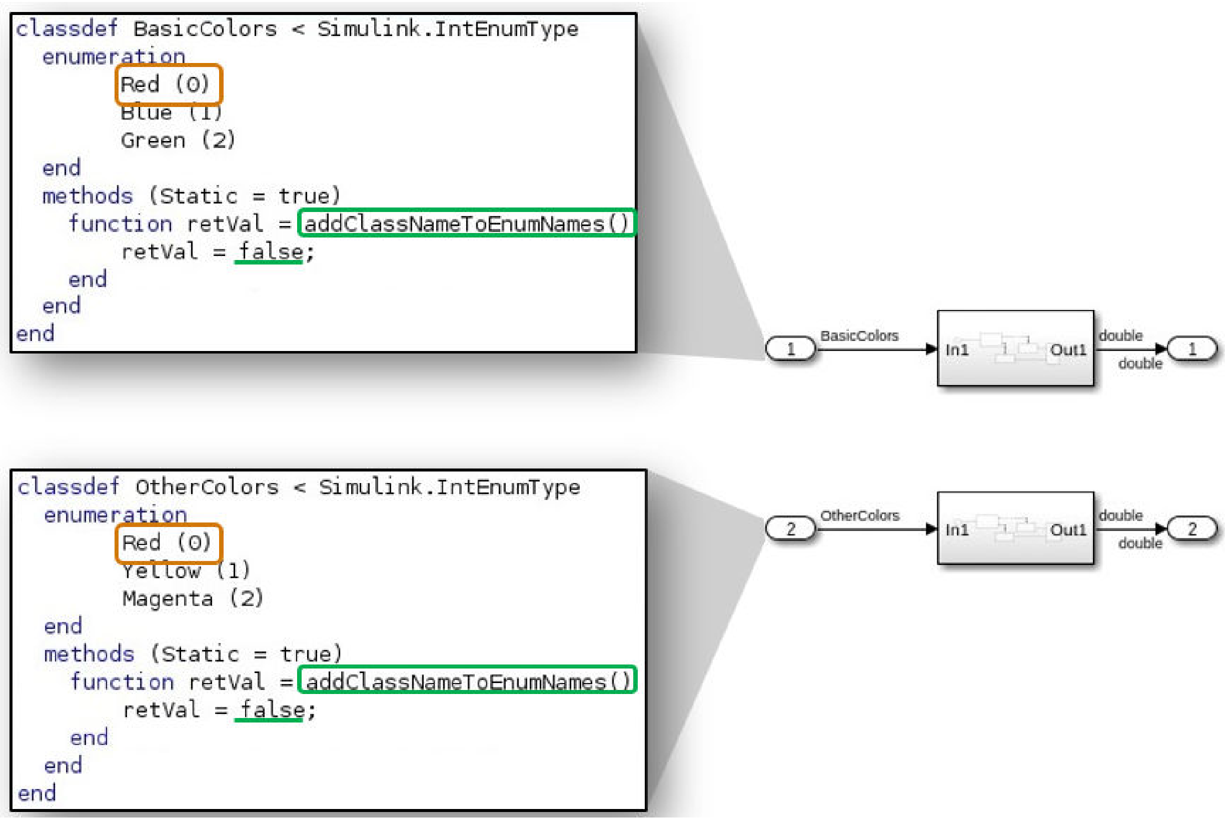
Checks That Evaluate Code Generation Readiness of the Model

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.



```
classdef BasicColors < Simulink.IntEnumType
  enumeration
        Red (0)
        Blue (1)
        Green (2)
  end
  methods (Static = true)
    function retVal = addClassNameToEnumNames()
        retVal = false;
    end
  end
end
```

```
classdef OtherColors < Simulink.IntEnumType
  enumeration
        Red (0)
        Yellow (1)
        Magenta (2)
  end
  methods (Static = true)
    function retVal = addClassNameToEnumNames()
        retVal = false;
    end
  end
end
```

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 7-76.

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

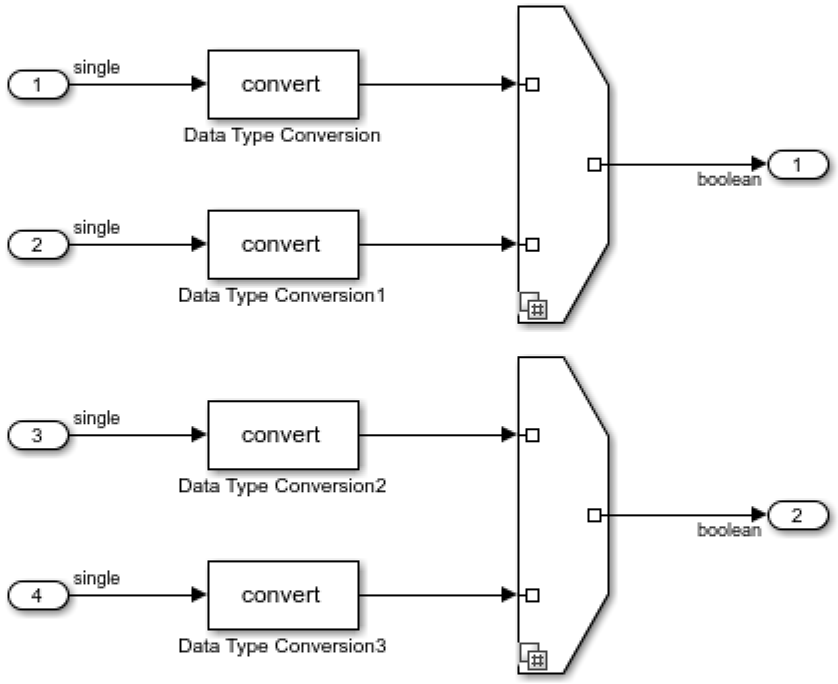
This example shows the creation of a custom Model Advisor check that evaluates active and inactive variant paths from a variant system model. The example provides Model Advisor results that demonstrate why you use `PostCompileForCodegen` versus `PostCompile` as the value for the `ModelAdvisor.Check.CallbackContext` property when generating code from the model is your final objective. See “Define the Compile Option for Custom Checks” on page 7-72.

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, 'boole
        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 bool
    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
```

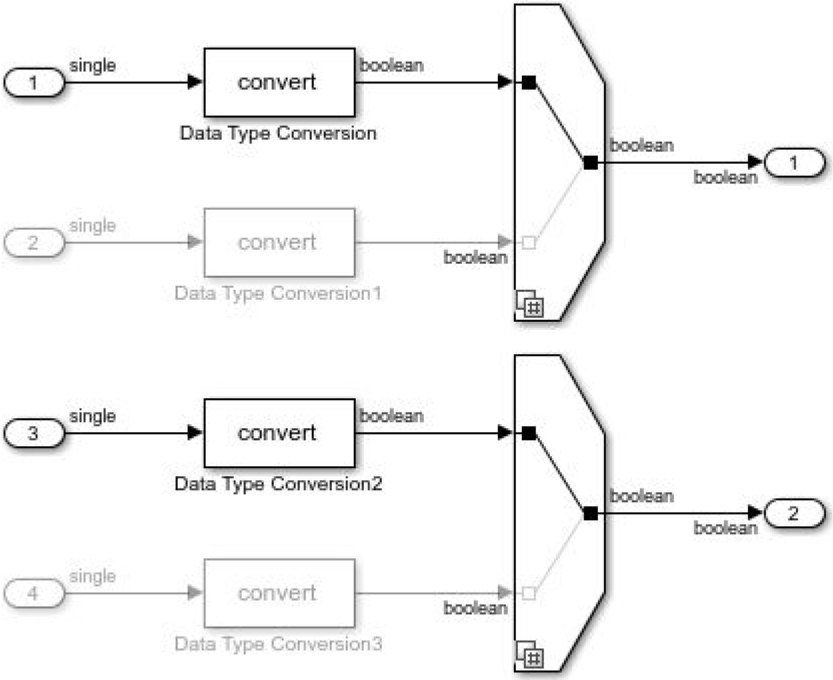
Open Model Advisor and Execute Custom Check

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. From the model window, select **Analysis > Model Advisor > Model Advisor**.
3. A **System Selector — Model Advisor** dialog box opens. Click **OK**. The Model Advisor window opens.
4. 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**.
5. 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.



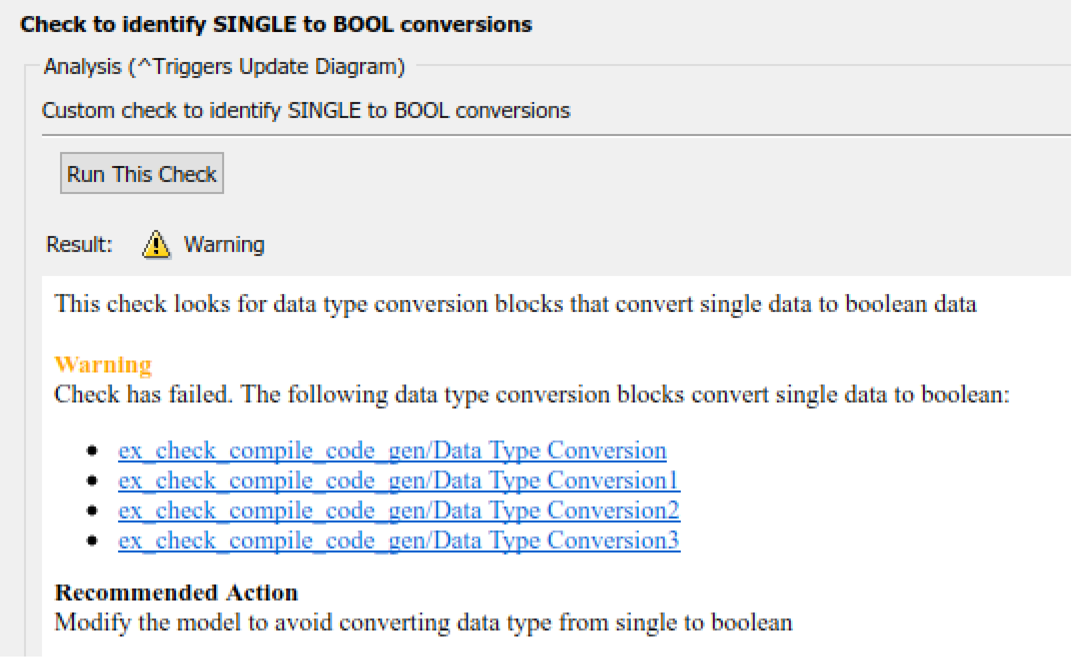
Review the Model Advisor Results

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.



Check to identify SINGLE to BOOL conversions
Analysis (^Triggers Update Diagram)
Run This Check

Custom check to identify SINGLE to BOOL conversions

Result: ⚠ Warning

This check looks for data type conversion blocks that convert single data to boolean data

Warning

Check has failed. The following data type conversion blocks convert single data to boolean:

- ex_check_compile_code_gen/Data Type Conversion
- ex_check_compile_code_gen/Data Type Conversion1
- ex_check_compile_code_gen/Data Type Conversion2
- ex_check_compile_code_gen/Data Type Conversion3

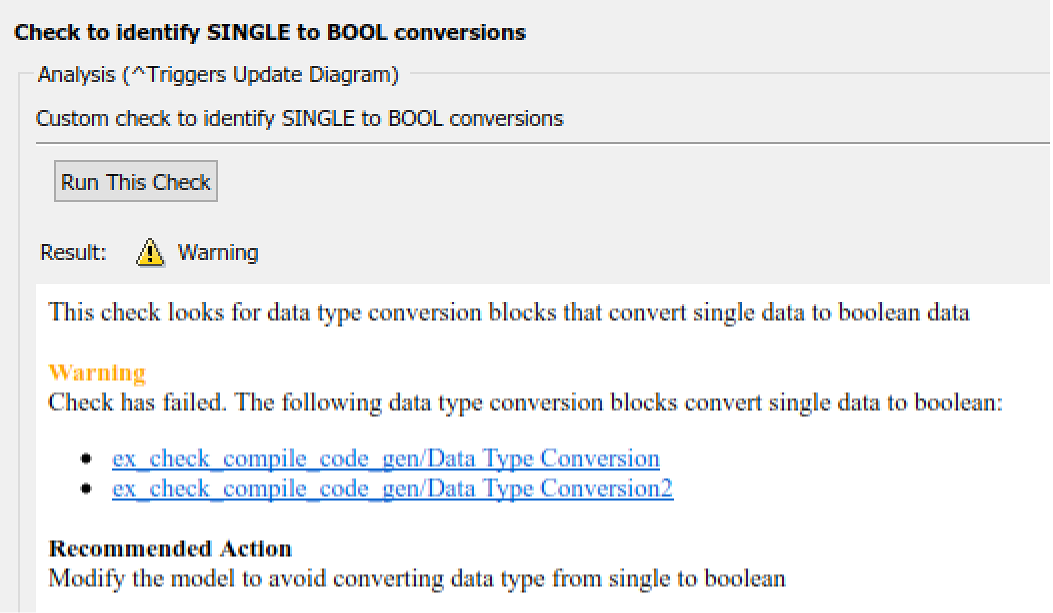
Recommended Action

Modify the model to avoid converting data type from single to boolean

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.



Check to identify SINGLE to BOOL conversions

Analysis (^Triggers Update Diagram)

Custom check to identify SINGLE to BOOL conversions

Result: ⚠ Warning

This check looks for data type conversion blocks that convert single data to boolean data

Warning
Check has failed. The following data type conversion blocks convert single data to boolean:

- ex_check_compile_code_gen/Data Type Conversion
- ex_check_compile_code_gen/Data Type Conversion2

Recommended Action
Modify the model to avoid converting data type from single to boolean

See Also

`ModelAdvisor.Check` | ModelAdvisor.Check.CallbackContext

More About

- “Define Custom Checks” on page 7-48
- “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 in 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 `sl_customization.m` file.

Update the `sl_customization.m` File

1. To open the example model, at the command prompt, type `slvnvdemo_mdladv`.
2. In the model window, double-click **View demo sl_customization.m**.
3. To exclude the custom check **Check Simulink block font** from blocks during Model Advisor analysis, make three modifications to the `sl_customization.m` file.

   a. Enable the **Check Simulink block font** check to support check exclusions by using the `ModelAdvisor.Check.supportExclusion` property. You can now exclude the check from model blocks. After `rec.setInputParametersLayoutGrid([3 2]);`, add `rec.supportExclusion = true;`. The `check 1` section of the `function defineModelAdvisorChecks` now looks like:

   ```
   % --- sample check 1
   rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
   rec.Title = 'Check Simulink block font';
   rec.TitleTips = 'Example style three callback';
   rec.setCallbackFcn(@SampleStyleThreeCallback,'None','StyleThree');
   rec.setInputParametersLayoutGrid([3 2]);
   rec.supportExclusion = true;
   ```

   b. Use the `Simulink.ModelAdvisor.filterResultWithExclusion` function to filter model objects causing a check warning or failure with checks that have exclusions enabled. To do this, there are two locations in the `sl_customization.m` file to modify, both in the `[ResultDescription, ResultDetails] = SampleStyleThreeCallback(system)` function:

   - After both instances of

     ```
     searchResult = setdiff(allBlks, regularBlks);
     ```

     add

```
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
```

- In the first location, the function now looks like:

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

  % look for different font blocks in the system
  searchResult = setdiff(allBlks, regularBlks);
  searchResult = mdladvObj.filterResultWithExclusion(searchResult);
  if ~isempty(searchResult)
  ```

- In the second location, the function now looks like:

  ```
  % find regular font size blocks
  regularBlks = find_system(allBlks,'FontSize',regularFontSize);
  % look for different font size blocks in the system
  searchResult = setdiff(allBlks, regularBlks);
  searchResult = mdladvObj.filterResultWithExclusion(searchResult);
  if ~isempty(searchResult)
  ```

4. Save the `sl_customization.m` file. If you are asked if it is ok to overwrite the file, click **OK**.

Create and Save Exclusions

1. In the model window, double-click **Launch Model Advisor**.
2. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings** > **Preferences** dialog box.
3. In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check fails.
4. In the Model Advisor window, click the **Enable highlighting** button (). The blocks causing the **Check Simulink block font** check failure are highlighted in yellow.
5. In the model window, right-click the X block and select **Model Advisor** > **Exclude block only** > **Check Simulink block font**.
6. In the Model Advisor Exclusion Editor, click **OK** to create the exclusion file.
7. In the model window, right-click the Input block and select **Model Advisor** > **Exclude block only** > **Check Simulink block font**.

Review Exclusions

1. In the Model Advisor Exclusion Editor, click **OK** to update the exclusion file.
2. In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check now passes. In the right-pane of the Model Advisor window, you can see the **Check Exclusion Rules** that the Model Advisor during the analysis.
3. Close `slvnvdemo_mdladv`.

See Also

`Simulink.ModelAdvisor` | ModelAdvisor.Check.supportExclusion

Related Examples

- Example of Excluding Gain and Outport Blocks From Checks on page 3-36
- Excluding Blocks From Model Advisor Checks on page 3-27

More About

- “Select and Run Model Advisor Checks” (Simulink)
- “Address Model Check Results with Highlighting” (Simulink)

Format Check Results

Format Results

You can make the analysis results of your custom checks appear similar to each other with minimal scripting using the `ModelAdvisor.FormatTemplate` class.

If this format template does not meet your needs, or if you want to format action results, use the Model Advisor Formatting API to produce formatted outputs in the Model Advisor. The following constructors of the `ModelAdvisor` class allow you to format the output.

Constructor	Description
`ModelAdvisor.Text`	Create Model Advisor text output.
`ModelAdvisor.List`	Create list.
`ModelAdvisor.Table`	Create table.
`ModelAdvisor.Paragraph`	Create and format paragraph.
`ModelAdvisor.LineBreak`	Insert line break.
`ModelAdvisor.Image`	Include image in Model Advisor output.

Format Text

Text is the simplest form of output. You can format text in many different ways. The default text formatting is:

- Empty
- Default color (black)
- Unformatted (not bold, italicized, underlined, linked, subscripted, or superscripted)

To change text formatting, use the `ModelAdvisor.Text` constructor. When you want one type of formatting for all text, use this syntax:

`ModelAdvisor.Text(content, {`*`attributes`*`})`

When you want multiple types of formatting, you must build the text.

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
```

```
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' for a uniform appearance in the model.');

result = [t1, t2, t3, t4, t5];
```

Add ASCII and Extended ASCII characters using the MATLAB `char` command. For more information, see the `ModelAdvisor.Text` class page.

Format Lists

You can create two types of lists: numbered and bulleted. The default list formatting is bulleted. Use the `ModelAdvisor.List` constructor to create and format lists. You can create lists with indented subsections, formatted as either numbered or bulleted.

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));

topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1',{'keyword','bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2',{'keyword','bold'}), subList]);
```

Format Tables

The default table formatting is:

- Default color (black)
- Left justified
- Bold title, row, and column headings

Change table formatting using the `ModelAdvisor.Table` constructor.

This example creates a subtable within a table.

```
table1 = ModelAdvisor.Table(1,1);
table2 = ModelAdvisor.Table(2,3);
table2.setHeading('Table 2');
table2.setHeadingAlign('center');
table2.setColHeading(1, 'Header 1');
table2.setColHeading(2, 'Header 2');
table2.setColHeading(3, 'Header 3');
table1.setHeading('Table 1');
table1.setEntry(1,1,table2);
```

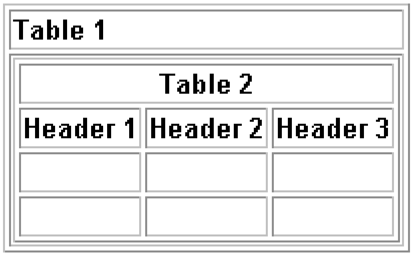


Table 1

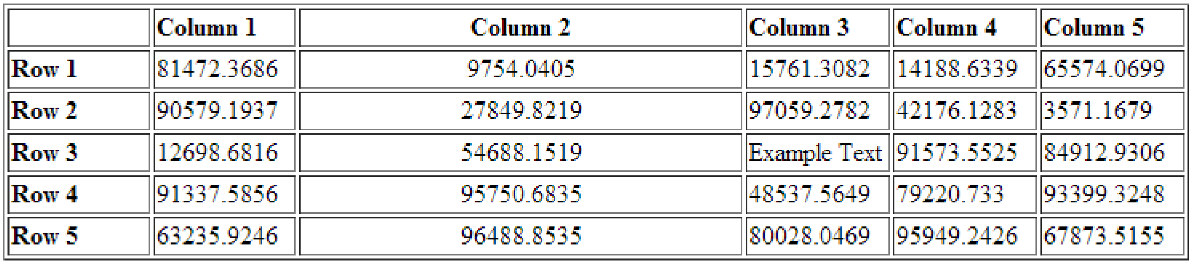
Table 2

Header 1	Header 2	Header 3

This example creates a table with five rows and five columns containing randomly generated numbers. Use the MATLAB code in a callback function to create the table. The Model Advisor displays `table1` in the results.

```
% ModelAdvisor.Table example

matrixData = rand(5,5) * 10^5;

% initialize a table with 5 rows and 5 columns (heading rows not counting)
table1 = ModelAdvisor.Table(5,5);

% set column headings
for n=1:5
    table1.setColHeading(n, ['Column ', num2str(n)]);
end

% set alignment of second column heading
table1.setColHeadingAlign(2, 'center');

% set column width of second column
table1.setColWidth(2, 3);

% set row headings
for n=1:5
    table1.setRowHeading(n, ['Row ', num2str(n)]);
end

% set Table content
for rowIndex=1:5
    for colIndex=1:5
        table1.setEntry(rowIndex, colIndex, ...
            num2str(matrixData(rowIndex, colIndex)));

        % set alignment of entries in second row
        if colIndex == 2
            table1.setEntryAlign(rowIndex, colIndex, 'center');
        end
    end
end

% overwrite content of cell 3,3 with a ModelAdvisor.Text
```

```
text = ModelAdvisor.Text('Example Text');
table1.setEntry(3,3, text)
```



	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	81472.3686	9754.0405	15761.3082	14188.6339	65574.0699
Row 2	90579.1937	27849.8219	97059.2782	42176.1283	3571.1679
Row 3	12698.6816	54688.1519	Example Text	91573.5525	84912.9306
Row 4	91337.5856	95750.6835	48537.5649	79220.733	93399.3248
Row 5	63235.9246	96488.8535	80028.0469	95949.2426	67873.5155

Format Paragraphs

You must handle paragraphs explicitly because most markup languages do not support line breaks. The default paragraph formatting is:

- Empty
- Default color (black)
- Unformatted, (not bold, italicized, underlined, linked, subscripted, or superscripted)
- Aligned left

If you want to change paragraph formatting, use the `ModelAdvisor.Paragraph` class.

Formatted Output

The following is the example from “Simple Check Callback Function” on page 7-58, reformatted using the Model Advisor Formatting API.

```
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
if strcmp(get_param(bdroot(system), 'ScreenColor'),'white')
    result = ModelAdvisor.Text('Passed',{'pass'});
    mdladvObj.setCheckResultStatus(true);
else
    msg1 = ModelAdvisor.Text(...
        ['It is recommended to select a Simulink window screen color'...
        ' of white to ensure a readable and printable model. Click ']);
    msg2 = ModelAdvisor.Text('here');
    msg2.setHyperlink('matlab: set_param(bdroot,''ScreenColor'',''white'')');
    msg3 = ModelAdvisor.Text(' to change screen color to white.');
    result = [msg1, msg2, msg3];
    mdladvObj.setCheckResultStatus(false);
end
```

Format Linebreaks

You can add a line break between two lines of text with the `ModelAdvisor.LineBreak` constructor.

```
result = ModelAdvisor.Paragraph;
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

Format Images

To include an image in Model Advisor output, use the `ModelAdvisor.Image` constructor. To create an `Image` object, use this syntax.

```
image_obj = ModelAdvisor.Image;
```

See Also

`ModelAdvisor.Check` | `ModelAdvisor.FormatTemplate` | `ModelAdvisor.Task`

Related Examples

- “Simple Check Callback Function” on page 7-58

More About

- Defining Custom Groups on page 8-13
- “Define Custom Checks” on page 7-48

8

Create Custom Configurations by Organizing Checks and Folders

Create Custom Configurations

You can use the Model Advisor APIs and Model Advisor Configuration Editor available with Simulink Check to do the tasks listed in the following table.

To	See
Create custom configurations by organizing Model Advisor checks and folders.	“Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5
Specify the order in which you make changes to your model.	“Create Procedural-Based Configurations” on page 9-5
Deploy custom configuration to your users.	“How to Deploy Custom Configurations” on page 10-3

Create Configurations by Organizing Checks and Folders

To customize the Model Advisor with MathWorks and custom checks, perform the following tasks:

1. Review the information in “Requirements for Customizing the Model Advisor” on page 6-2.
2. Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
3. Organize the checks into new and existing folders to create custom configurations. See “Organize and Deploy Model Advisor Checks”.

   a. Identify which checks you want to include in your custom Model Advisor configuration. You can use MathWorks checks and/or custom checks.
   b. Create the custom configurations using either of the following:

      - Model Advisor Configuration Editor - “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5.
      - A customization file - “Organize Customization File Checks and Folders” on page 8-11.
   c. Verify the custom configuration. See “Verify and Use Custom Configurations” on page 8-17.
4. Optionally, deploy the custom configurations to your users. See “Organize and Deploy Model Advisor Checks”.
5. Verify that models comply with modeling guidelines. See “Select and Run Model Advisor Checks” (Simulink).

Create Procedural-Based Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

To create a procedural-based configuration, perform the following tasks:

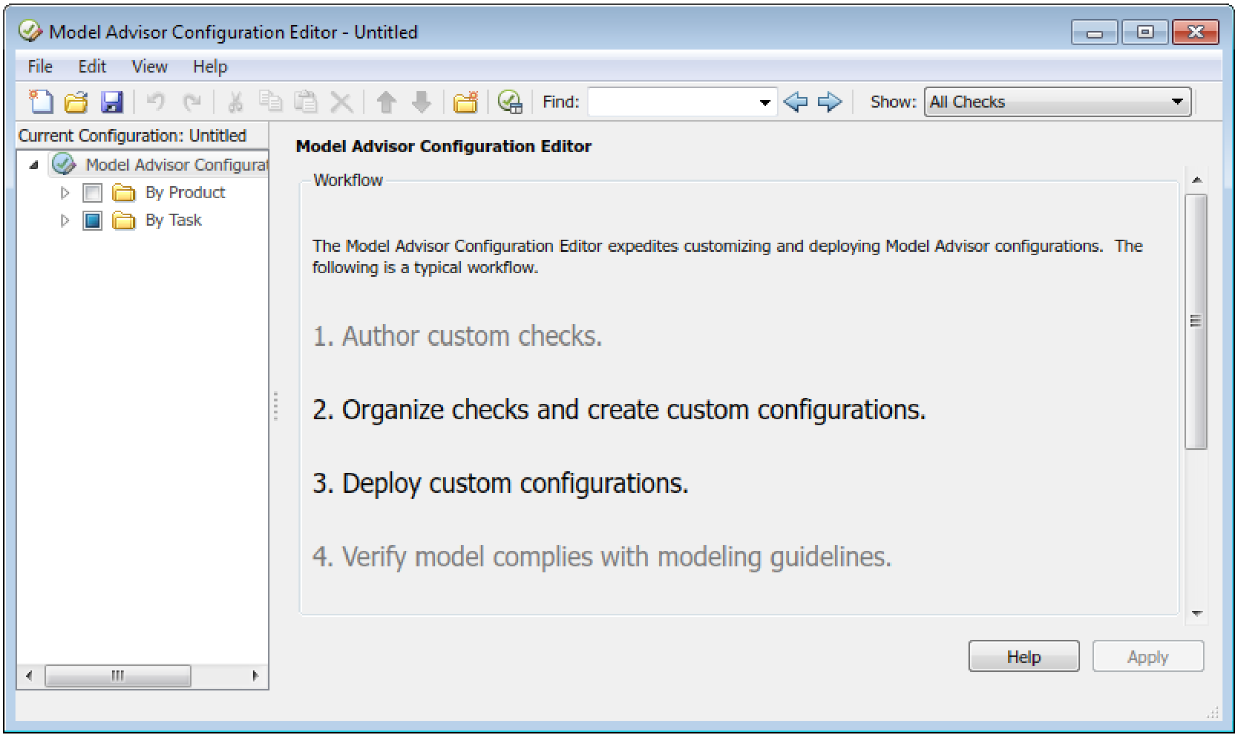
1. Review the information in “Requirements for Customizing the Model Advisor” on page 6-2.
2. Decide on order of changes to your model.
3. Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.
4. Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
5. Organize the checks into procedures for a procedural-based configuration. See “Create Procedural-Based Configurations” on page 9-5.

   a. Create procedures using the procedure API. For detailed information, see “Create Procedures Using the Procedures API” on page 9-2.
   b. Create the custom configuration by using a customization file. See “Organize Customization File Checks and Folders” on page 8-11.
   c. Verify the custom configuration as described in “Verify and Use Custom Configurations” on page 8-17.
6. Optionally, deploy the custom configurations to your users. For detailed information, see “Organize and Deploy Model Advisor Checks”.
7. Verify that models comply with modeling guidelines. For detailed information, see “Select and Run Model Advisor Checks” (Simulink).

Organize Checks and Folders Using the Model Advisor Configuration Editor

Overview of the Model Advisor Configuration Editor

When you start the Model Advisor Configuration Editor, two windows open; the Model Advisor Configuration Editor and the Model Advisor Check Browser. The Configuration Editor window consists of two panes: the Model Advisor Configuration Editor hierarchy and the Workflow. The Model Advisor Configuration Editor hierarchy lists the checks and folders in the current configuration. The Workflow on the right shows the common workflow you use to create a custom configuration.



Model Advisor Configuration Editor

Workflow

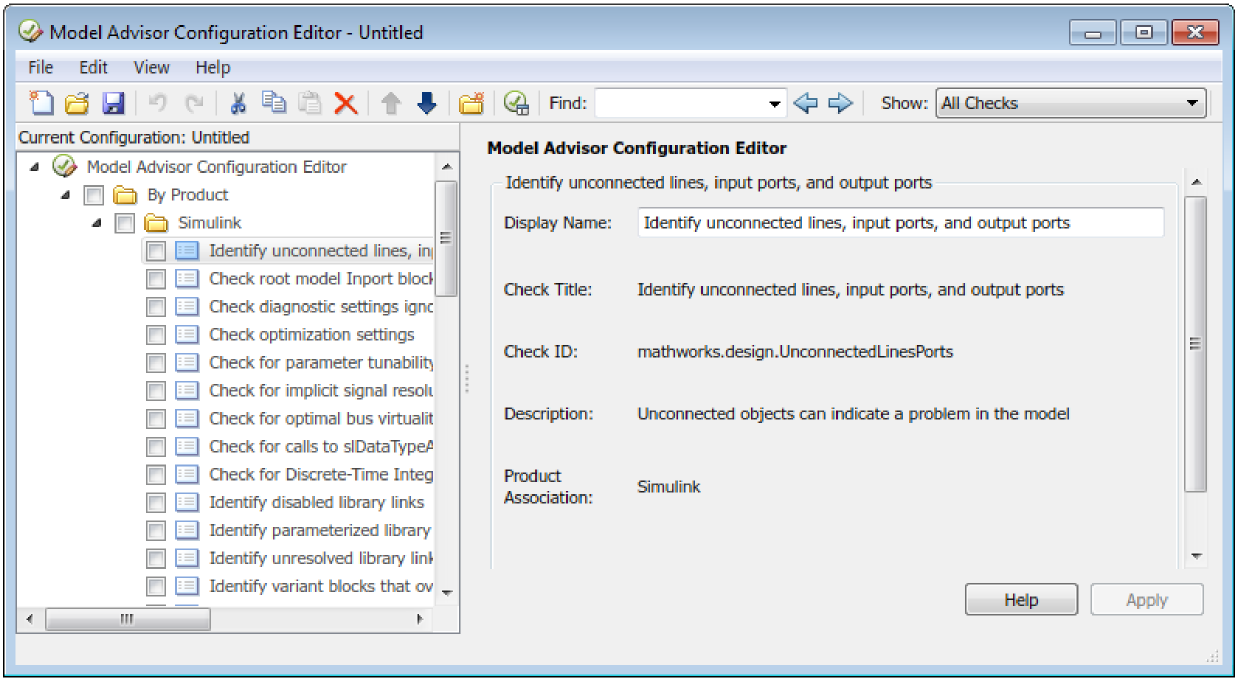
The Model Advisor Configuration Editor expedites customizing and deploying Model Advisor configurations. The following is a typical workflow.

1. Author custom checks.
2. Organize checks and create custom configurations.
3. Deploy custom configurations.
4. Verify model complies with modeling guidelines.

Model Advisor Configuration Editor

If you want the Model Advisor Configuration Editor hierarchy to list only the checks configured for edit-time checking, in the **Show** field, select `Edit-Time Supported Checks`. Or, in the model window, select **Analysis** > **Model Advisor** > **Configure Advisor Edit-Time Checks**.

When you select a folder or check in the Model Advisor Configuration Editor hierarchy, the Workflow pane changes to display information about the check or folder. You can change the display name of the check or folder in this pane.


Model Advisor Configuration Editor - Untitled
File
Edit
View
Help
Find:
Show: All Checks
Current Configuration: Untitled
Model Advisor Configuration Editor
By Product
Simulink
Identify unconnected lines, in
Check root model Inport block
Check diagnostic settings ignc
Check optimization settings
Check for parameter tunability
Check for implicit signal resolu
Check for optimal bus virtualit
Check for calls to slDataTypeA
Check for Discrete-Time Integ
Identify disabled library links
Identify parameterized library
Identify unresolved library link
Identify variant blocks that ov
Model Advisor Configuration Editor
Help
Apply

Identify unconnected lines, input ports, and output ports

Display Name: Identify unconnected lines, input ports, and output ports

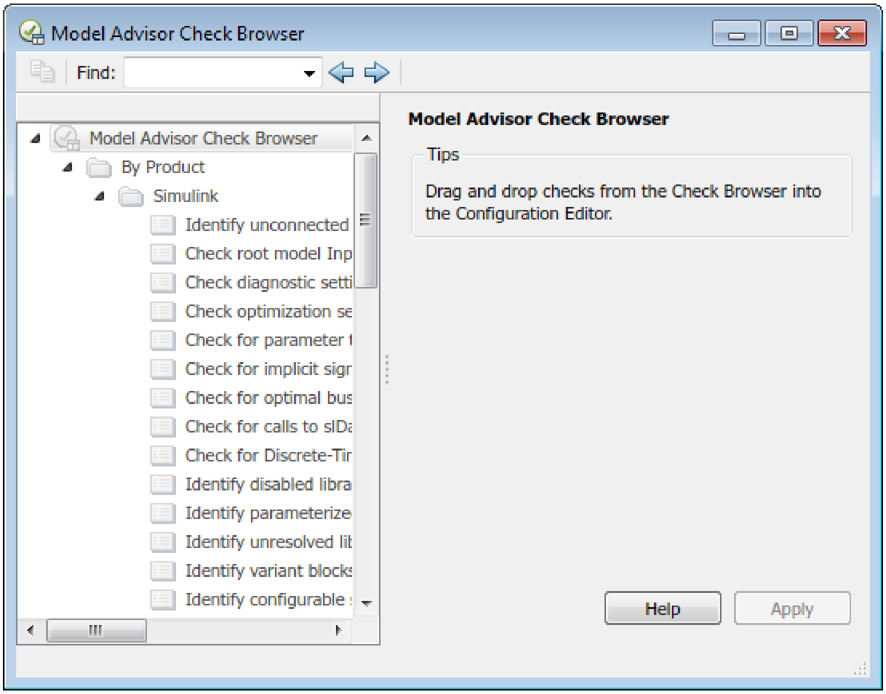
Check Title: Identify unconnected lines, input ports, and output ports

Check ID: mathworks.design.UnconnectedLinesPorts

Description: Unconnected objects can indicate a problem in the model

Product Association: Simulink

The Model Advisor Check Browser window includes a read-only list of available checks. If you delete a check in the Model Advisor Configuration Editor, you can retrieve a copy of it from the Model Advisor Check Browser.



Drag and drop checks from the Check Browser into the Configuration Editor.

Model Advisor Check Browser

Using the Model Advisor Configuration Editor, you can perform the following actions.

To...	Select...
Create new configurations	**File > New**
Find checks and folders in the Model Advisor Check Browser	**View > Check Browser**
Add checks and folders to the configuration	**Edit > Copy** **Edit > Paste** **Edit > New folder** The check or folder and drag and drop
Remove checks and folders from the configuration	**Edit > Delete** **Edit > Cut**

To...	Select...
Reorder checks and folders	**Edit > Move up** **Edit > Move down** The check or folder and drag and drop
Rename checks and folders **Note** MathWorks folder display names are restricted. When you rename a folder, you cannot use the restricted display names.	The check or folder and edit **Display Name** in right pane.
Allow or gray out the check box control for checks and folders **Tip** This capability is equivalent to enabling checks, described in “Display and Enable Checks” on page 7-49.	**Edit > Enable** **Edit > Disable**
Save the configuration as a MAT file for use and distribution	**File > Save** **File > Save As**
Set the configuration so it opens by default in the Model Advisor	**File > Set Current Configuration as Default**
Restore the MathWorks default configuration	**File > Restore Default Configuration**
Load and edit saved configurations	**File > Open**

Start the Model Advisor Configuration Editor

Before starting the Model Advisor Configuration Editor, verify that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.

Note

- The Model Advisor Configuration Editor uses the `slprj` folder in the code generation folder (Simulink). If the `slprj` folder does not exist in the code generation folder, the Model Advisor Configuration Editor creates it.

1 To include custom checks in the new Model Advisor configuration, update the Simulink environment to include your `sl_customization.m` file.

2 Start the Model Advisor Configuration Editor.

To start the Model Advisor Configuration Editor...	**Do this:**
Programmatically	At the MATLAB command line, enter `Simulink.ModelAdvisor.openConfigUI`.
From the Model Advisor	**a** Start the Model Advisor. **b** Select **Settings** > **Open Configuration Editor**.

The Model Advisor Configuration Editor and Model Advisor Check Browser windows open.

3 Optionally, to edit an existing configuration in the Model Advisor Configuration Editor window:

a Select **File** > **Open**.

b In the Open dialog box, navigate to the configuration file that you want to edit.

c Click **Open**.

Organize Checks and Folders Using the Model Advisor Configuration Editor

The following tutorial steps you through creating a custom configuration.

1 Open the Model Advisor Configuration Editor at the MATLAB command line by entering `Simulink.ModelAdvisor.openConfigUI` .

2 In the Model Advisor Configuration Editor, in the left pane, delete the **By Product** and **By Task** folders, to start with a blank configuration.

3 Select the root node which is labeled Model Advisor Configuration Editor.

4 In the toolbar, click the **New Folder** button to create a folder.

5 In the left pane, select the new folder.

6 In the right pane, edit **Display Name** to rename the folder. For the purposes of this tutorial, rename the folder to **Review Optimizations**.

7. In the Model Advisor Check Browser window, in the **Find** field, enter `optimization` to find **Simulink** > **Check optimization settings**.
8. Drag and drop **Check optimization settings** into **Review Optimizations**.
9. In the Model Advisor Check Browser window, find **Simulink Check** > **Modeling Standards** > **DO-178C/DO-331Checks** > **Check safety-related optimization settings**.
10. Drag and drop **Check safety-related optimization settings** into **Review Optimizations**.
11. In the Model Advisor Configuration Editor window, expand **Review Optimizations**.
12. Rename **Check optimization settings** to **Check Simulink optimization settings**.
13. Select **File** > **Save As** to save the configuration.
14. Name the configuration `optimization_configuration.mat`.
15. Close the Model Advisor Configuration Editor window.

Tip To move a check to the first position in a folder:

1. Drag the check to the second position.
2. Right-click the check and select **Move up**.

See Also

`Simulink.ModelAdvisor` | `ModelAdvisor.Check`

Related Examples

- “Update the Environment to Include Your sl_customization File” on page 8-17

Organize Customization File Checks and Folders

Customization File Overview

The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

Function	Description	Required or Optional
`sl_customization()`	Registers custom checks and tasks, folders with the Simulink customization manager at startup. See "Register Checks" on page 7-43.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See "Define Custom Checks" on page 7-48.	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 8-13.	Required to add custom checks to the Model Advisor, except when adding the checks to the **By Product** folder. Write one task for each check that you add to the Model Advisor.
One or more groups	Defines custom groups. See "Define Custom Tasks" on page 8-13.	Required to add custom tasks to new folders in the Model Advisor, except when adding a new subfolder to the **By Product** folder. Write one group definition for each new folder.

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

Register Tasks and Folders

Create sl_customization Function

To add tasks and folders to the Model Advisor, create the `sl_customization.m` file on your MATLAB path. Then create the `sl_customization()` function in the `sl_customization.m` file on your MATLAB path.

Tip

- You can have more than one `sl_customization.m` file on your MATLAB path.
- Do not place an `sl_customization.m` file that customizes the Model Advisor in your root MATLAB folder or its subfolders, except for the *`matlabroot`*`/work` folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

The customization manager object includes methods for registering custom checks, tasks, 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 Model Advisor, select **View** > **Source** tab.
2. Navigate to the folder that contains the MathWorks check.
3. In the right pane, click **Source**. The Model Advisor displays the **Title**, **TitleID**, and **Source** information for each check in the folder.
4. Select and copy the **TitleID** of the check that you want to add as a task.

Display and Enable Tasks

The `Visible`, `Enable`, and `Value` properties interact the same way for tasks as they do for checks.

Define Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor using the following guidelines:

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Task Definition Function

The following example shows a task definition function. This function defines three tasks.

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.

Customization Example

The Simulink Check software provides an example that shows how to customize the Model Advisor by adding:

- Custom checks
- Check input parameters
- Check actions
- Check list views to call the Model Advisor Result Explorer
- Custom tasks to include the custom checks in the Model Advisor
- Custom folders for grouping the checks
- Custom procedures

The example also provides the source code of the `sl_customization.m` file that executes the customizations.

To run the example:

1. At the MATLAB command line, type`slvnvdemo_mdladv`.
2. Follow the instructions in the model.

See Also

`ModelAdvisor.Check` | `ModelAdvisor.FactoryGroup` | `ModelAdvisor.Group` | `ModelAdvisor.Root.publish` | `ModelAdvisor.Task`

Related Examples

- “Update the Environment to Include Your sl_customization File” on page 8-17

More About

- “Define Custom Checks” on page 7-48
- “Display and Enable Checks” on page 7-49
- “Register Checks” on page 7-43

Verify and Use Custom Configurations

Update the Environment to Include Your sl_customization File

When you start Simulink, it reads customization (`sl_customization.m`) files. If you change the contents of your customization file, update your environment by performing these tasks:

1. If you previously started the Model Advisor:

   a. Close the model from which you started the Model Advisor

   b. Clear the data associated with the previous Model Advisor session by removing the `slprj` folder from your code generation folder (Simulink).

2. At the MATLAB command line, enter:

   ```
   sl_refresh_customizations
   ```

   If you have created custom checks, at the MATLAB command line, then also enter:

   ```
   Advisor.Manager.refresh_customizations
   ```

3. Open your model.
4. Start the Model Advisor.

Verify Custom Configurations

To verify a custom configuration:

1. If you created custom checks, or created the custom configuration using the `sl_customization` method, update the Simulink environment.
2. Open a model.
3. From the model window, start the Model Advisor.
4. Select **Settings** > **Load Configuration**. If you see a warning that the Model Advisor report corresponds to a different configuration, click **Load** to continue.
5. In the Open dialog box, navigate to and select your custom configuration.
6. When the Model Advisor reopens, verify that the configuration contains the new folders and checks. For example, the **Review Optimizations** folder and the **Check Simulink optimization settings** and **Check safety-related optimization settings** checks.

7. Optionally, run the checks.

See Also

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5

Customize Model Advisor Check for Nondefault Block Attributes

You can customize the list of nondefault block parameters that are flagged by the Model Advisor MAAB check **Check for Nondefault Block Attributes** (**Check ID**: `mathworks.maab.db_0140`).

1. In the Model Advisor, select **Settings** > **Open Configuration Editor**.
2. In the **Find** field, enter `db_0140` and press **Enter**.
3. The Model Advisor Configuration Editor window displays the check **Check for Nondefault Block Attributes**. On the right pane, under **Input Parameters** > **Standard**, select **Custom**. The **Custom** setting enables editing of the parameter list.
4. In the table, find the block type for which you want to change the nondefault parameter list. Under **Parameter**, select a cell to edit. The parameters are separated with spaces.
5. Delete or add a parameter name that corresponds to the **BlockType**. For example, to remove the rounding method parameter from the check for each gain block, find `Gain` under **BlockType**. Under **Parameter**, delete the parameter name `RndMeth`. **Check ID**: `mathworks.maab.db_0140` no longer checks for the display of nondefault rounding methods from gain blocks' annotations.

See Also

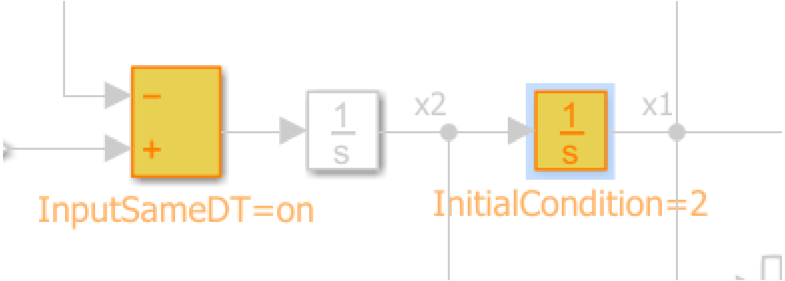
More About

- "Check for nondefault block attributes"
- "Customize Model Advisor Check for Nondefault Block Attributes" on page 8-19
- db_0140: Display of basic block parameters

Automatically Fix Display of Nondefault Block Parameters

To conform with Model Advisor MAAB check **Check for Nondefault Block Attributes** (**Check ID**: `mathworks.maab.db_0140`), you can use the **Add nondefault values into block annotation** button to automatically add descriptive text to the model editor window.

1. At the command prompt, type `vdp` and press **Enter**. The Van der Pol equation model opens in the Simulink editor window.
2. The model has two blocks which do not display nondefault values as annotations. Run the Model Advisor from **Analysis > Model Advisor > Model Advisor**.
3. On the left pane, select **By Product > Simulink Check > Modeling Standards > MathWorks Automotive Advisory Board Checks**. On the right pane, run the check by selecting **Run Selected Checks**.
4. The Model Advisor runs the check and displays a warning for the integrator block that has a nonzero initial condition not currently displayed. On the Model Advisor toolbar, select **Enable highlighting** ( ) to highlight the blocks causing the warning.
5. In the right pane of the Model Advisor window, select **Add nondefault values into block annotation** to automatically add the nondefault attribute and value to the integrator block's annotation. Model Advisor displays `InitialCondition = 2`.



6. Run the check again to clear the warning.

See Also

More About

- “Check for nondefault block attributes”
- “Automatically Fix Display of Nondefault Block Parameters” on page 8-20
- db_0140: Display of basic block parameters
- “Select and Run Model Advisor Checks” (Simulink)

Create Procedural-Based Model Advisor Configurations

Create Procedures

What Is a Procedure?

A procedure is a series of checks. The checks in a procedure depend on passing the previous checks. If Check A is the first check in a procedure and Check B follows, the Model Advisor does not run Check B until Check A passes. Checks A and B can be either custom or provided by MathWorks.

You create procedures with the `ModelAdvisor.Procedure` class API. You first add the checks to tasks, which are wrappers for the checks. The tasks are added to procedures.

When creating procedural checks, be aware of potential conflicts with the checks. Verify that it is possible to pass both checks.

Create Procedures Using the Procedures API

You use the `ModelAdvisor.Procedure` class to create procedural checks.

1. Add each check to a task using the `ModelAdvisor.Task.setCheck` method. The task is a wrapper for the check. You cannot add checks directly to procedures.
2. Add each task to a procedure using the `ModelAdvisor.Procedure.addTask` method.

Define Procedures

You define procedures in a procedure definition function that specifies the properties of each instance of the `ModelAdvisor.Procedure` class. Define one instance of the procedure class for each procedure that you want to add to the Model Advisor. Then register the procedure using the `ModelAdvisor.Root.register` method.

Add Subprocedures and Tasks to Procedures

You can add subprocedures or tasks to a procedure. The tasks are wrappers for checks.

- Use the `ModelAdvisor.Procedure.addProcedure` method to add a subprocedure to a procedure.
- Use the `ModelAdvisor.Procedure.addTask` method to add a task to a procedure.

Define Where Procedures Appear

You can specify where the Model Advisor places a procedure using the `ModelAdvisor.Group.addProcedure` method.

Procedure Definition

The following code example adds procedures to a group:

```
%Create three procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure3');

%Create a group
MAG = ModelAdvisor.Group('com.mathworks.sample.myGroup');

%Add the three procedures to the group
addProcedure(MAG, MAP1);
addProcedure(MAG, MAP2);
addProcedure(MAG, MAP3);

%register the group and procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAG);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

The following code example adds subprocedures to a procedure:

```
%Create a 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);
```

See Also

`ModelAdvisor.Procedure` | `ModelAdvisor.Procedure.addProcedure` | `ModelAdvisor.Procedure.addTask` | `ModelAdvisor.Root.register` | `ModelAdvisor.Task.setCheck`

Related Examples

- “Create Procedural-Based Configurations” on page 9-5

More About

- “Define Custom Tasks” on page 8-13

Create Procedural-Based Configurations

Overview of Procedural-Based Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

To create a procedural-based configuration, perform the following tasks:

1. Review the information in “Requirements for Customizing the Model Advisor” on page 6-2.
2. Decide on order of changes to your model.
3. Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.
4. Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
5. Organize the checks into procedures for a procedural-based configuration. See “Create Procedural-Based Configurations” on page 9-5.

   a. Create procedures using the procedure API. For detailed information, see “Create Procedures Using the Procedures API” on page 9-2.
   b. Create the custom configuration by using a customization file. See “Organize Customization File Checks and Folders” on page 8-11.
   c. Verify the custom configuration as described in “Verify and Use Custom Configurations” on page 8-17.
6. Optionally, deploy the custom configurations to your users. For detailed information, see “Organize and Deploy Model Advisor Checks”.
7. Verify that models comply with modeling guidelines. For detailed information, see “Run Model Checks” (Simulink).

Create a Procedural-Based Configuration

In this example, you examine a procedural-based configuration.

1. At the MATLAB command line, type`slvnvdemo_mdladv`.
2. In the model window, select **View demo sl_customization.m**. The `sl_customization.m` file opens in the MATLAB Editor window.

   The file contains four checks created in the function `defineModelAdvisorChecks`:

   - `ModelAdvisor.Check('com.mathworks.sample.Check1')` - Checks Simulink block fonts.
   - `ModelAdvisor.Check('com.mathworks.sample.Check2')` - Checks Simulink window screen color.
   - `ModelAdvisor.Check('com.mathworks.sample.Check3')` - Checks model optimization settings.
   - `ModelAdvisor.Check('com.mathworks.sample.Check4')` - Checks Gain block usage.

   Each check has a set of fix actions.
3. In the `sl_customization.m` file, examine the function `defineTaskAdvisor`.

   - The `ModelAdvisor.Procedure` class API creates procedures `My Procedure` and `My sub_Procedure`:

     ```
     % Define procedures
     MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');
     MAP.DisplayName='My Procedure';

     MAP_sub = ModelAdvisor.Procedure('com.mathworks.sample.sub_ProcedureSample');
     MAP_sub.DisplayName='My sub_Procedure';
     ```
   - The `ModelAdvisor.Task` class API creates tasks `MAT4`, `MAT5`, `MAT6`, and `MAT7`. The `ModelAdvisor.Task.setCheck` method adds the checks to the tasks:

     ```
     % Define tasks
     MAT4 = ModelAdvisor.Task('com.mathworks.sample.TaskSample4');
     MAT4.DisplayName='Check Simulink block font';
     MAT4.setCheck('com.mathworks.sample.Check1');
     mdladvRoot.register(MAT4);

     MAT5 = ModelAdvisor.Task('com.mathworks.sample.TaskSample5');
     MAT5.DisplayName='Check Simulink window screen color';
     MAT5.setCheck('com.mathworks.sample.Check2');
     mdladvRoot.register(MAT5);
     ```

```
MAT6 = ModelAdvisor.Task('com.mathworks.sample.TaskSample6');
MAT6.DisplayName='Check model optimization settings';
MAT6.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT6);

MAT7 = ModelAdvisor.Task('com.mathworks.sample.TaskSample7');
MAT7.DisplayName='Check gain block usage';
MAT7.setCheck('com.mathworks.sample.Check4');
mdladvRoot.register(MAT7);
```

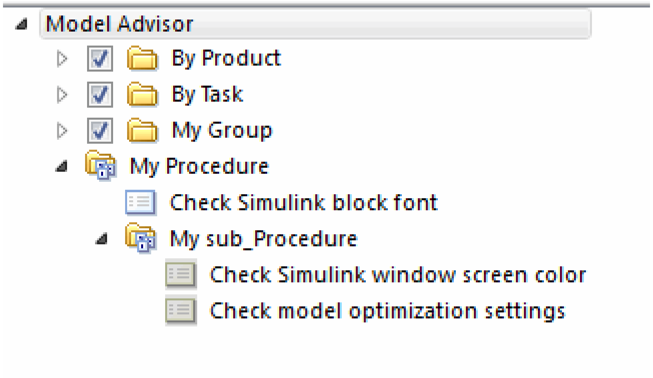
- The `ModelAdvisor.Procedure.addTask` method adds task `MAT4` to `My Procedure` and tasks `MAT5`, `MAT6`, and `MAT7` to `My sub_Procedure`. The `ModelAdvisor.Procedure.addProcedure` method adds `My sub_Procedure` to `My Procedure`:

```
% Add tasks to procedures:
% Add Task4 to MAP
MAP.addTask(MAT4);
% Now Add Task5 and Task6 to MAP_sub
MAP_sub.addTask(MAT5);
MAP_sub.addTask(MAT6);
MAP_sub.addTask(MAT7);
% Include the Sub-Procedure in the Procedure
MAP.addProcedure(MAP_sub);
```

4 From the model window, select **Analysis** > **Model Advisor** > **Model Advisor** to open the Model Advisor.

5 A **System Selector — Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens.

6 In the left pane, expand **My Procedure > My sub_Procedure**. The Check Simulink block font check is in the My Procedure folder. My sub_Procedure contains Check Simulink window screen color and Check model optimization settings.


Model Advisor
☑ By Product
☑ By Task
☑ My Group
My Procedure
Check Simulink block font
My sub_Procedure
Check Simulink window screen color
Check model optimization settings

7 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run All**. The Model Advisor Check Simulink block font check fails. The Model Advisor does not check the remaining two checks in the My sub_Procedure folder. Running the checks in the My sub_Procedure folder depends on passing the Check Simulink block font check.

8 In the **Action** section of the Model Advisor dialog box, click **Fix block fonts**.

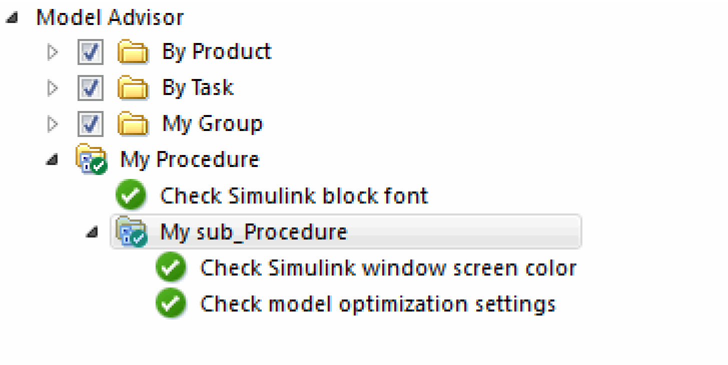
9 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run All**. The Check Simulink block font check passes. The Model Advisor runs the Check Simulink window screen color check. This check fails and the Model Advisor stops checking.

10 In the **Action** section of the Model Advisor dialog box, click **Fix window screen color**.

11 In the left pane of the Model Advisor, select My sub_Procedure. In the right pane of the Model Advisor, click **Run All**. The Check Simulink window screen color check passes. The Model Advisor runs the Check model optimization settings check. This check warns.

12 In the **Action** section of the Model Advisor dialog box, click **Fix model optimization settings**.

13 In the left pane of the Model Advisor, select Check model optimization settings. In the right pane of the Model Advisor, click **Run This Task**. The Check model optimization settings check passes.


Model Advisor
☑ By Product
☑ By Task
☑ My Group
My Procedure
✓ Check Simulink block font
My sub_Procedure
✓ Check Simulink window screen color
✓ Check model optimization settings

See Also

`ModelAdvisor.Procedure` | `ModelAdvisor.Procedure.addProcedure` | `ModelAdvisor.Procedure.addTask` | `ModelAdvisor.Root.register` | `ModelAdvisor.Task.setCheck`

More About

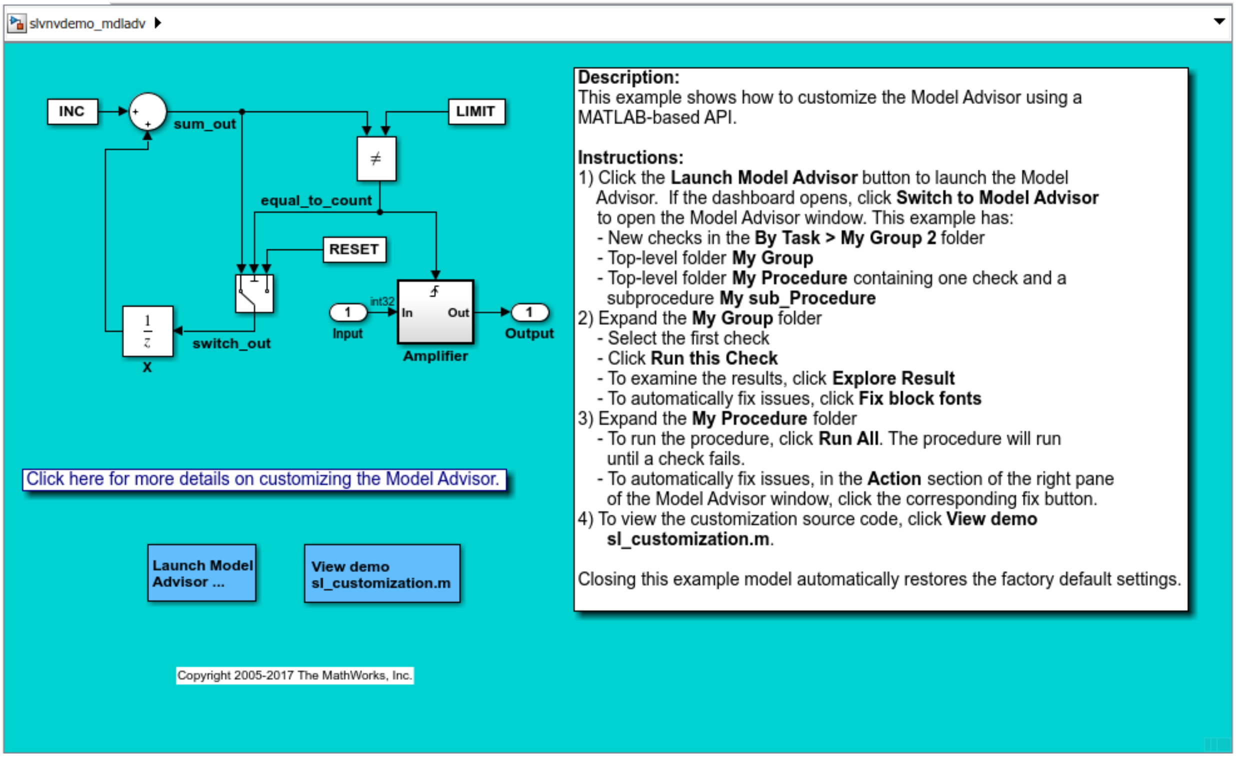
- “Create Procedures” on page 9-2
- “Define Custom Checks” on page 7-48

Add Checks and Tasks to the Model Advisor

This example shows how to customize the Model Advisor using a MATLAB-based API.

Open the Example Model

Open the example model `slvnvdemo_mdladv`.



Click here for more details on customizing the Model Advisor.

Description:
This example shows how to customize the Model Advisor using a MATLAB-based API.

Instructions:
1) Click the **Launch Model Advisor** button to launch the Model Advisor. If the dashboard opens, click **Switch to Model Advisor** to open the Model Advisor window. This example has:
   - New checks in the **By Task > My Group 2** folder
   - Top-level folder **My Group**
   - Top-level folder **My Procedure** containing one check and a subprocedure **My sub_Procedure**
2) Expand the **My Group** folder
   - Select the first check
   - Click **Run this Check**
   - To examine the results, click **Explore Result**
   - To automatically fix issues, click **Fix block fonts**
3) Expand the **My Procedure** folder
   - To run the procedure, click **Run All**. The procedure will run until a check fails.
   - To automatically fix issues, in the **Action** section of the right pane of the Model Advisor window, click the corresponding fix button.
4) To view the customization source code, click **View demo sl_customization.m**.

Closing this example model automatically restores the factory default settings.

Copyright 2005-2017 The MathWorks, Inc.

Open the Model Advisor

From the model editor window, launch the Model Advisor by clicking the Model Advisor icon.

In the **System Selector** dialog, click **OK**. For this example, the Model Advisor contains:

- Hidden shipping checks
- New checks in the **By Task > My Group 2** folder
- Top-level folder **My Group**
- Top-level folder **My Procedure** that contains once check and a subprocedure **My sub_Procedure**

Explore the Custom Folder

Explore the custom **My Group** folder:

- In the left pane, click the **My Group** folder.
- Select the first check, **Example task with input parameter and auto-fix ability**.
- In the right pane, click **Run This Check**.
- To examine the results in the Model Advisor Results Explorer dialog, click **Explore Result**.
- To automatically fix the issues, return to the Model Advisor and, in the right pane, click **Fix block fonts**.

Explore the Custom Procedure Folder

Explore the custom **My Procedure** folder:

- In the left pane, click the **My Procedure** folder.
- To run the procedure, in the right pane, click **Run All**. The procedure runs until a check fails.
- To automatically fix issues, in the **Action** section of the right pane, click the corresponding fix button.

View the Customization Code

To implement these customizations, on the MATLAB path, create an `sl_customization.m` file with the following:

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

% Copyright 2005-2017 The MathWorks, Inc.
% register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);
% register custom factory group
cm.addModelAdvisorTaskFcn(@defineModelAdvisorTasks);
% register custom tasks.
cm.addModelAdvisorTaskAdvisorFcn(@defineTaskAdvisor);

% -----------------------------
% defines Model Advisor Tasks
% -----------------------------
function defineModelAdvisorTasks
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='My Group 2';
rec.Description='Demo Factory Group';
rec.addCheck('com.mathworks.sample.Check1');
rec.addCheck('com.mathworks.sample.Check2');
rec.addCheck('com.mathworks.sample.Check3');
mdladvRoot.publish(rec); % publish inside By Group list


% -----------------------------
% defines Model Advisor Checks
% -----------------------------
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;

% --- sample check 1
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback,'None','StyleThree');
rec.setInputParametersLayoutGrid([3 2]);
% set input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
```

```
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
% set fix operation
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
myAction.Description='Click the button to update all blocks with specified font';
rec.setAction(myAction);
rec.ListViewVisible = true;
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

% --- sample check 2
rec = ModelAdvisor.Check('com.mathworks.sample.Check2');
rec.Title = 'Check Simulink window screen color';
rec.TitleTips = 'Example style one callback';
rec.setCallbackFcn(@SampleStyleOneCallback,'None','StyleOne');
% set fix operation
myAction2 = ModelAdvisor.Action;
myAction2.setCallbackFcn(@sampleActionCB2);
myAction2.Name='Fix window screen color';
myAction2.Description='Click the button to change Simulink window screen color to white
rec.setAction(myAction2);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

% --- sample check 3
rec = ModelAdvisor.Check('com.mathworks.sample.Check3');
rec.Title = 'Check model optimization settings';
rec.TitleTips = 'Example style two callback';
rec.setCallbackFcn(@SampleStyleTwoCallback,'None','StyleTwo');
% set fix operation
myAction3 = ModelAdvisor.Action;
```

```
myAction3.setCallbackFcn(@sampleActionCB3);
myAction3.Name='Fix model optimization settings';
myAction3.Description='Click the button to turn on model optimization settings';
rec.setAction(myAction3);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.


% -----------------------------
% defines Model Advisor tasks
% please refer to Model Advisor API document for more details.
% -----------------------------
function defineTaskAdvisor
mdladvRoot = ModelAdvisor.Root;

MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task with input parameter and auto-fix ability';
MAT1.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT1);

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';
MAT2.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT2);

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
MAT3.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT3);

MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.DisplayName='My Group';
MAG.addTask(MAT1);
MAG.addTask(MAT2);
MAG.addTask(MAT3);
mdladvRoot.publish(MAG); % publish under Root

% Define procedures
MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');
MAP.DisplayName='My Procedure';

MAP_sub = ModelAdvisor.Procedure('com.mathworks.sample.sub_ProcedureSample');
MAP_sub.DisplayName='My sub_Procedure';

% Define tasks
```

```
MAT4 = ModelAdvisor.Task('com.mathworks.sample.TaskSample4');
MAT4.DisplayName='Check Simulink block font';
MAT4.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT4);

MAT5 = ModelAdvisor.Task('com.mathworks.sample.TaskSample5');
MAT5.DisplayName='Check Simulink window screen color';
MAT5.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT5);

MAT6 = ModelAdvisor.Task('com.mathworks.sample.TaskSample6');
MAT6.DisplayName='Check model optimization settings';
MAT6.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT6);

% Add tasks to procedures:
% Add Task4 to MAP
MAP.addTask(MAT4);
% Now Add Task5 and Task6 to MAP_sub
MAP_sub.addTask(MAT5);
MAP_sub.addTask(MAT6);
% Include the Sub-Procedure in the Procedure
MAP.addProcedure(MAP_sub);

mdladvRoot.register(MAP_sub); % publish under Root
mdladvRoot.publish(MAP); % publish under Root


% -----------------------------
% Sample StyleThree callback function,
% please refer to Model Advisor API document for more details.
% -----------------------------
function [ResultDescription, ResultDetails] = SampleStyleThreeCallback(system)
ResultDescription ={};
ResultDetails ={};

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(true);
needEnableAction = false;
% get input parameters
inputParams = mdladvObj.getInputParameters;
skipFontCheck = inputParams{1}.Value;
regularFontSize = inputParams{2}.Value;
regularFontName = inputParams{3}.Value;
```

```
if skipFontCheck
    ResultDescription{end+1} = ModelAdvisor.Paragraph('Skipped.');
    ResultDetails{end+1}     = {};
    return
end
regularFontSize = str2double(regularFontSize);
if regularFontSize<1 || regularFontSize>=99
    mdladvObj.setCheckResultStatus(false);
    ResultDescription{end+1} = ModelAdvisor.Paragraph('Invalid font size. Please enter
    ResultDetails{end+1}     = {};
end

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

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

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

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to use same
        'The following blocks use a font other than ' regularFontName ': ']);
    ResultDetails{end+1}     = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
    myLVParam.Data = get_param(searchResult,'object')';
    myLVParam.Attributes = {'FontName'}; % name is default property
    mdladvObj.setListViewParameters({myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph('All block font names are identi
    ResultDetails{end+1}     = {};
end

% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
```

```
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to use same f
        'The following blocks use a font size other than ' num2str(regularFontSize) ':
    ResultDetails{end+1}     = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font size blocks'; % the name appeared at pull down filte
    myLVParam.Data = get_param(searchResult,'object')';
    myLVParam.Attributes = {'FontSize'}; % name is default property
    mdladvObj.setListViewParameters({mdladvObj.getListViewParameters{:}, myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph('All block font sizes are identic
    ResultDetails{end+1}     = {};
end

mdladvObj.setActionEnable(needEnableAction);
mdladvObj.setCheckErrorSeverity(1);

% -----------------------------
% Sample StyleOne callback function,
% please refer to Model Advisor API document for more details.
% -----------------------------
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

if strcmp(get_param(bdroot(system),'ScreenColor'),'white')
    result = ModelAdvisor.Text('Passed',{'pass'});
    mdladvObj.setCheckResultStatus(true); % set to pass
    mdladvObj.setActionEnable(false);
else
    result = ModelAdvisor.Text('It is recommended to select a Simulink window screen co
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
    mdladvObj.setCheckErrorSeverity(1);
end

% -----------------------------
% Sample StyleTwo callback function,
% please refer to Model Advisor API document for more details.
% -----------------------------
function [ResultDescription, ResultDetails] = SampleStyleTwoCallback(system)
ResultDescription ={};
ResultDetails ={};
```

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

% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph('Check Simulation optimization settin
if strcmp(get_param(model,'BlockReduction'),'off')
    ResultDetails{end+1}     = {ModelAdvisor.Text('It is recommended to turn on Block r
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
else
    ResultDetails{end+1}     = {ModelAdvisor.Text('Passed',{'pass'})};
end

% Check code generation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph('Check code generation optimization s
ResultDetails{end+1}  = {};
if strcmp(get_param(model,'LocalBlockOutputs'),'off')
    ResultDetails{end}{end+1}     = ModelAdvisor.Text('It is recommended to turn on Ena
    ResultDetails{end}{end+1}     = ModelAdvisor.LineBreak;
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if strcmp(get_param(model,'BufferReuse'),'off')
    ResultDetails{end}{end+1}     = ModelAdvisor.Text('It is recommended to turn on Reu
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if isempty(ResultDetails{end})
    ResultDetails{end}{end+1}     = ModelAdvisor.Text('Passed',{'pass'});
end


% -----------------------------
% Sample action callback function,
% please refer to Model Advisor API document for more details.
% -----------------------------
function result = sampleActionCB(taskobj)
mdladvObj = taskobj.MAObj;
system = getfullname(mdladvObj.System);

% get input parameters
inputParams = mdladvObj.getInputParameters;
```

```
regularFontSize = inputParams{2}.Value;
regularFontName = inputParams{3}.Value;

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

% find regular font name blocks
regularBlks = find_system(allBlks,'FontName',regularFontName);
% look for different font blocks in the system
fixBlks = setdiff(allBlks, regularBlks);
% fix them one by one
for i=1:length(fixBlks)
    set_param(fixBlks{i},'FontName',regularFontName);
end
% save result
resultText1 = ModelAdvisor.Text([num2str(length(fixBlks)), ' blocks has been updated wi

% find regular font size blocks
regularBlks = find_system(allBlks,'FontSize',str2double(regularFontSize));
% look for different font size blocks in the system
fixBlks = setdiff(allBlks, regularBlks);
% fix them one by one
for i=1:length(fixBlks)
    set_param(fixBlks{i},'FontSize',regularFontSize);
end
% save result
resultText2 = ModelAdvisor.Text([num2str(length(fixBlks)), ' blocks has been updated wi
result = ModelAdvisor.Paragraph;
result.addItem([resultText1 ModelAdvisor.LineBreak resultText2]);
mdladvObj.setActionEnable(false);

% -----------------------------
% Sample action callback function for Check Simulink window screen color
% please refer to Model Advisor API document for more details.
% -----------------------------
function result = sampleActionCB2(taskobj)
mdladvObj = taskobj.MAObj;
system = mdladvObj.System;
set_param(bdroot(system),'ScreenColor','white');
result = ModelAdvisor.Text('Simulink window screen color has been updated to white colo
mdladvObj.setActionEnable(false);
```

```
% -----------------------------
% Sample action callback function for model optimization settings
% please refer to Model Advisor API document for more details.
% -----------------------------
function result = sampleActionCB3(taskobj)
mdladvObj = taskobj.MAObj;
model = bdroot(mdladvObj.System);
set_param(model,'BlockReduction','on');
set_param(model,'LocalBlockOutputs','on');
set_param(model,'BufferReuse','on');
result = ModelAdvisor.Text('Model optimization options "Block reduction", "Enable local
mdladvObj.setActionEnable(false);
```

10

Deploy Custom Configurations

Overview of Deploying Custom Configurations

About Deploying Custom Configurations

When you create a custom configuration, often you deploy the custom configuration to your development group. Deploying the custom configuration allows your development group to review models using the same checks.

After you create a custom configuration, you can use it in the Model Advisor, or deploy the configuration to your users. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

Deploying Custom Configurations Workflow

When you deploy custom configurations, you:

1. Optionally author custom checks, as described in “Create Model Advisor Checks”.
2. Organize checks and folders to create custom configurations, as described in “Create Custom Configurations” on page 8-2.
3. Deploy the custom configuration to your users, as described in “How to Deploy Custom Configurations” on page 10-3.

How to Deploy Custom Configurations

To deploy a custom configuration:

1. Determine which files to distribute. You might need to distribute more than one file.

If You...	Using the...	Distribute...
Created custom checks	Customization file	• `sl_customization.m` • Files containing check and action callback functions (if separate)
Organized checks and folders	Model Advisor Configuration Editor	Configuration MAT file
	Customization file	`sl_customization.m`

2. Distribute the files and tell the user to include these files on the MATLAB path.
3. Instruct the user to load the custom configuration.

See Also

Related Examples

- “Manually Load and Set the Default Configuration” on page 10-4

Manually Load and Set the Default Configuration

When you use the Model Advisor, you can load any configuration. Once you load a configuration, you can set it so that the Model Advisor use that configuration every time you open the Model Advisor.

1. Open the Model Advisor.
2. Select **Settings** > **Load Configuration**.
3. In the Open dialog box, navigate to and select the configuration file that you want to edit.
4. Click **Open**.

   Simulink reloads the Model Advisor using the new configuration.
5. Optionally, when the Model Advisor opens, set the current configuration as the default by selecting **File** > **Set Current Configuration as Default**.

See Also

Related Examples

- "Update the Environment to Include Your sl_customization File" on page 8-17

More About

- "Organize Checks and Folders Using the Model Advisor Configuration Editor" on page 8-5

Model Slicer

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.

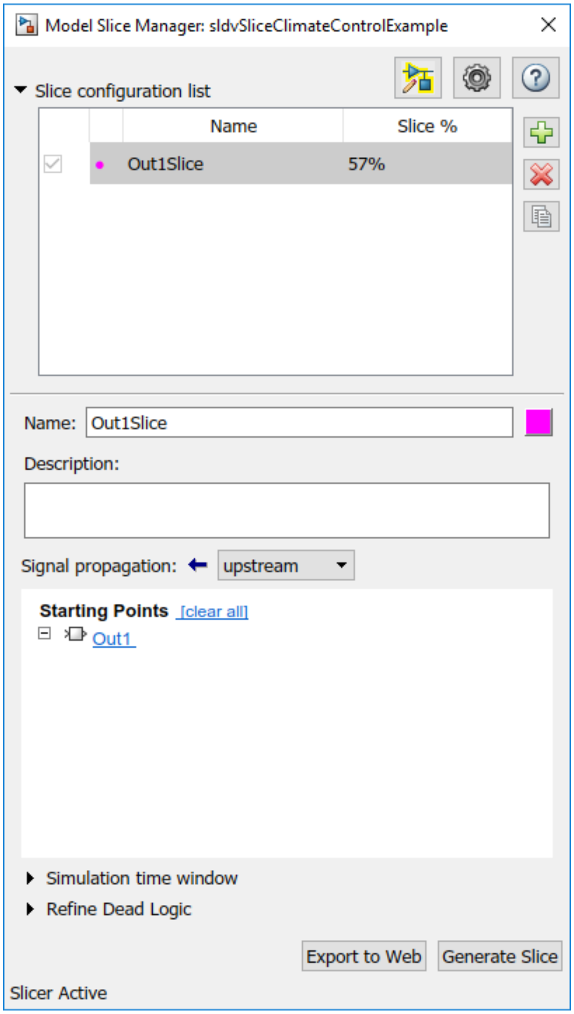
   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   ```
2. Open the `sldvSliceClimateControlExample` model.

   ```
   sldvSliceClimateControlExample
   ```
3. Select **Analysis** > **Model Slicer** to open the Model Slice Manager.

   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**.



Model Slice Manager: sldvSliceClimateControlExample

Slice configuration list

	Name	Slice %
☑	● Out1Slice	57%

Name: Out1Slice

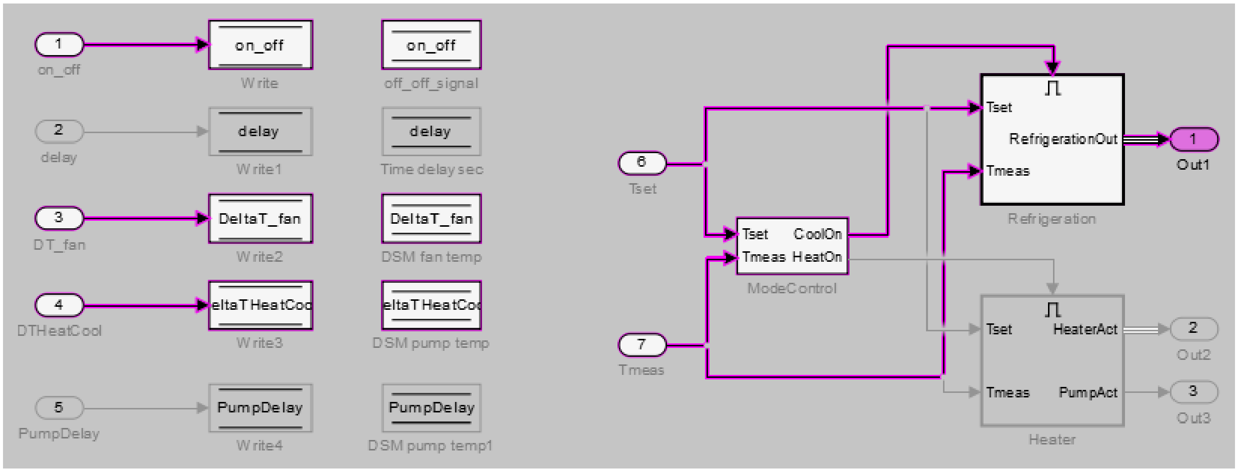
Description:

Signal propagation: upstream

Starting Points [clear all]

Out1

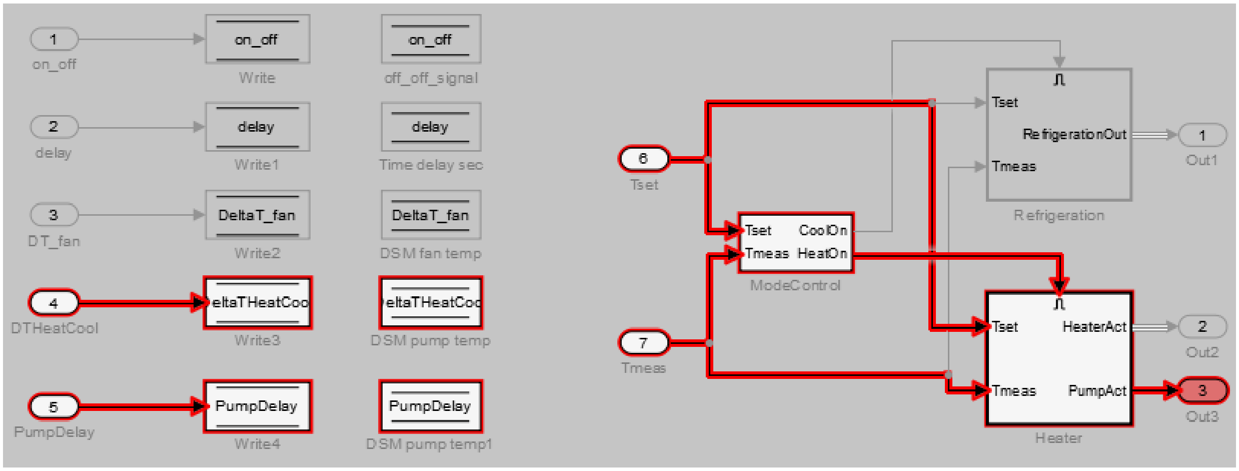
The Model Slicer now highlights the upstream constructs that affect `Out1`.



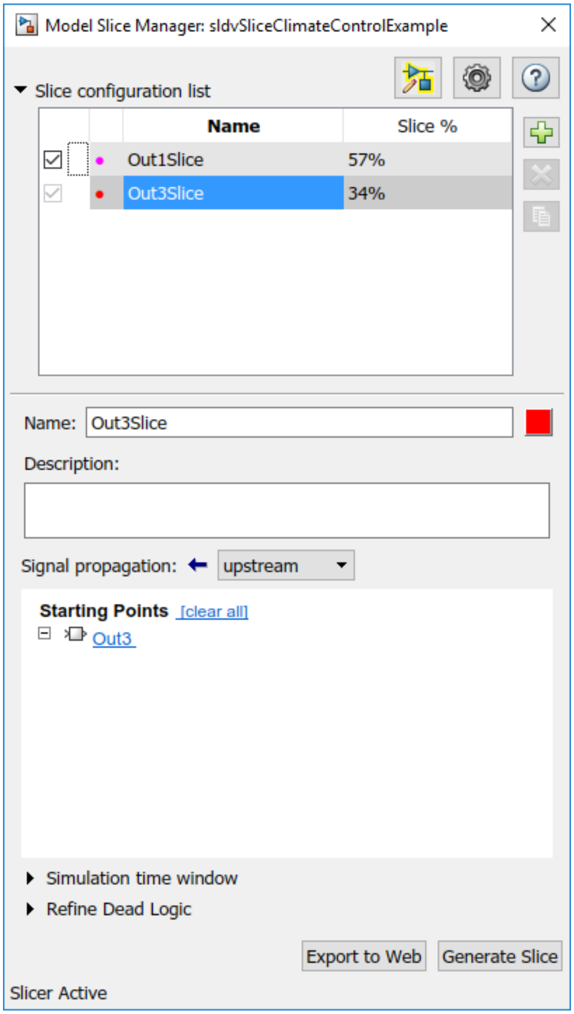
If you create two slice configurations, you can highlight the intersecting portions of their highlights. Create a new slice configuration and view the intersecting portions of the slice configuration you created above and the new slice configuration:

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 Slice Manager: sldvSliceClimateControlExample

Slice configuration list

		Name	Slice %
☑	●	Out1Slice	57%
☑	●	Out3Slice	34%

Name: Out3Slice 🟥

Description:

Signal propagation: ← upstream

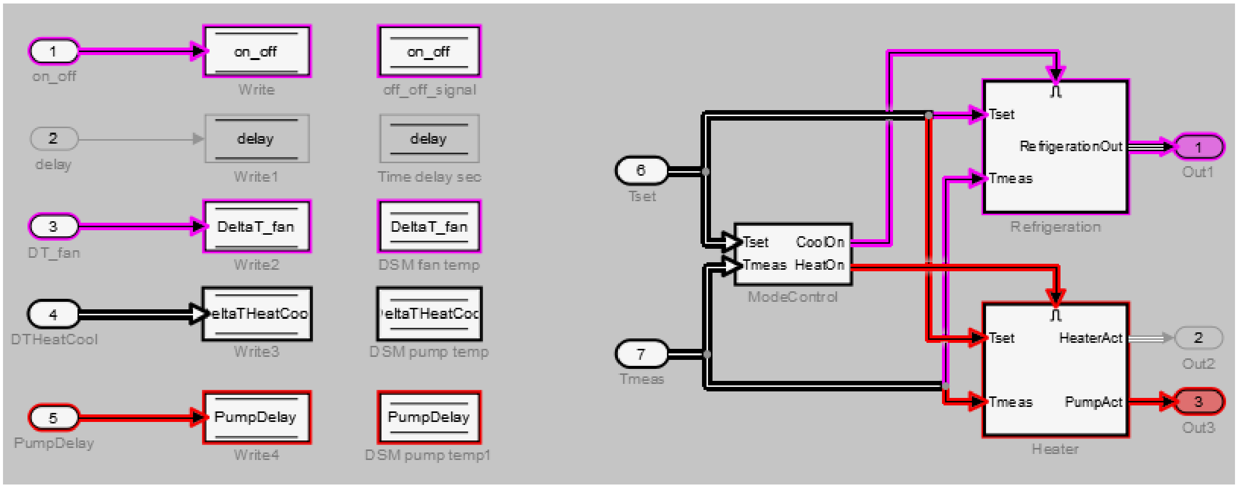
Starting Points [clear all]

⊟ Out3

Model Slicer highlights portions of the model as follows:

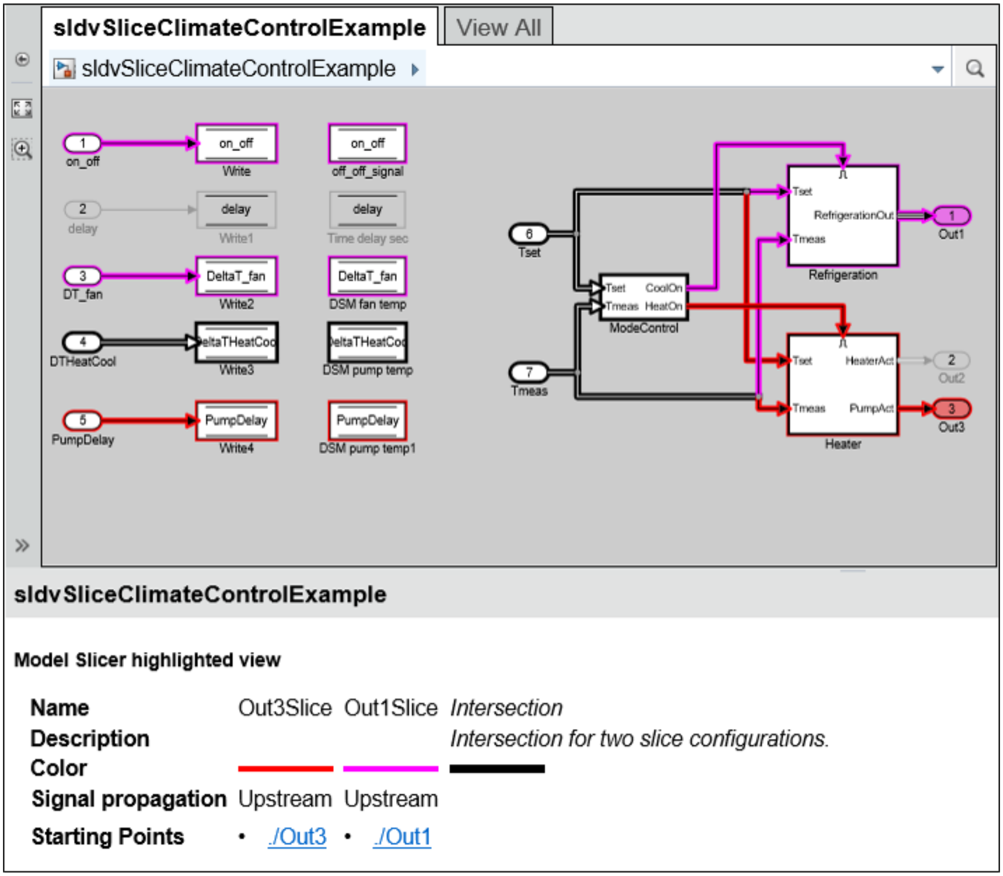
- The portions of the model that are exclusively upstream of `Out1` are highlighted in cyan.
- The portions of the model that are exclusively upstream of `Out3` are highlighted in red.

- 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`.



sldvSliceClimateControlExample

Model Slicer highlighted view

Name	Out3Slice	Out1Slice	*Intersection*
Description			*Intersection for two slice configurations.*
Color	🟥	🟪	⬛
Signal propagation	Upstream	Upstream	
Starting Points	• ./Out3	• ./Out1	

See Also

More About

- “Refine Highlighted Model” on page 11-13
- “Create a Simplified Standalone Model” on page 11-33
- “Model Slicer Considerations and Limitations” on page 11-53

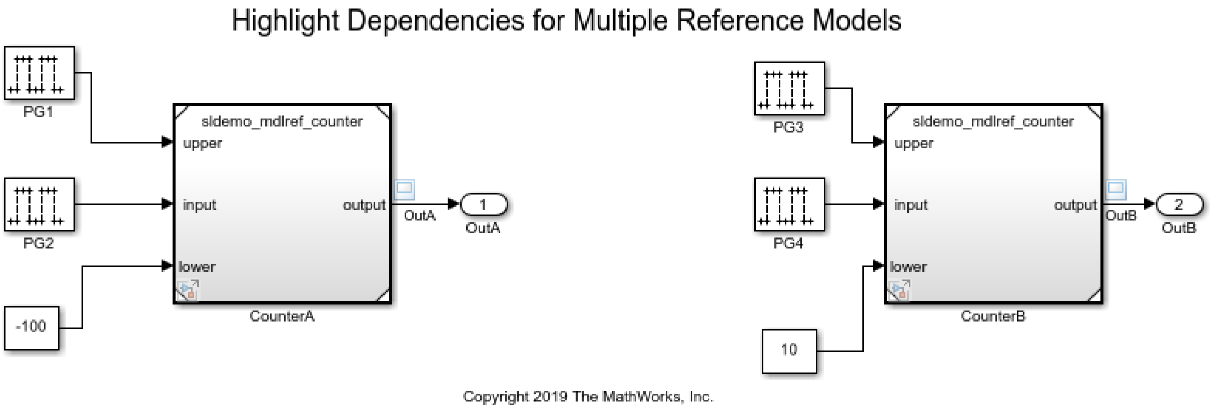
Highlight Dependencies for Multiple Instance Reference Models

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');
```

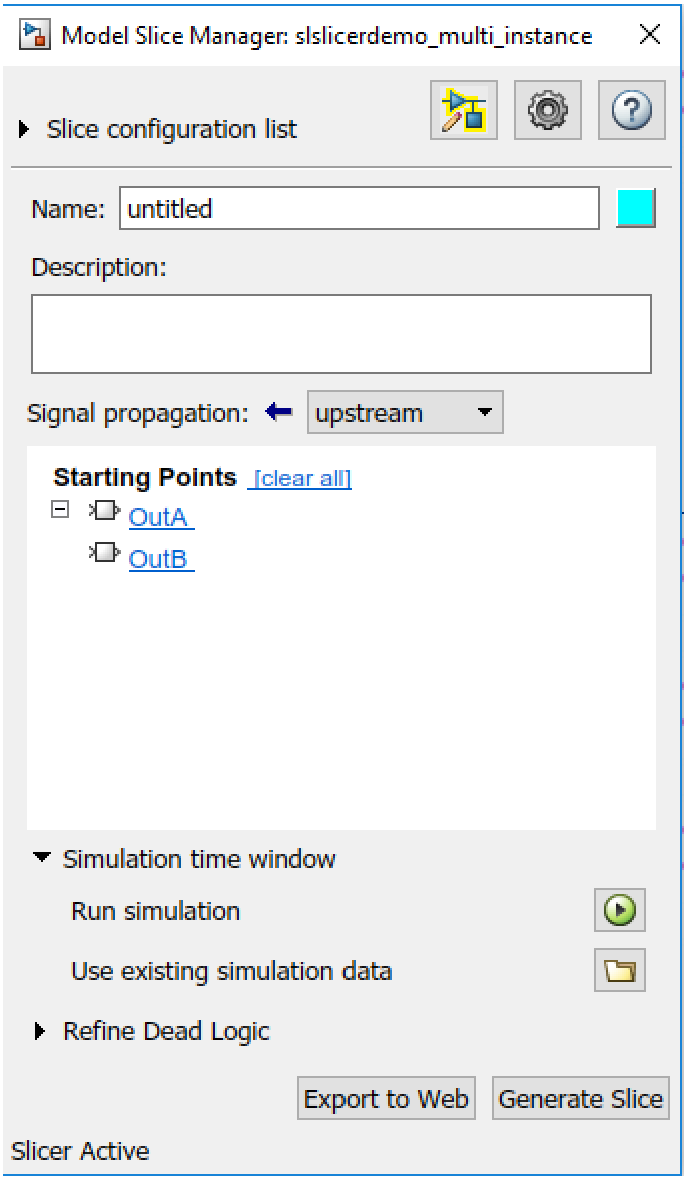


2. Click **Analysis > Model Slicer**.

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**.



Model Slice Manager: slslicerdemo_multi_instance

Slice configuration list

Name: untitled

Description:

Signal propagation: upstream

Starting Points [clear all]

OutA

OutB

Simulation time window

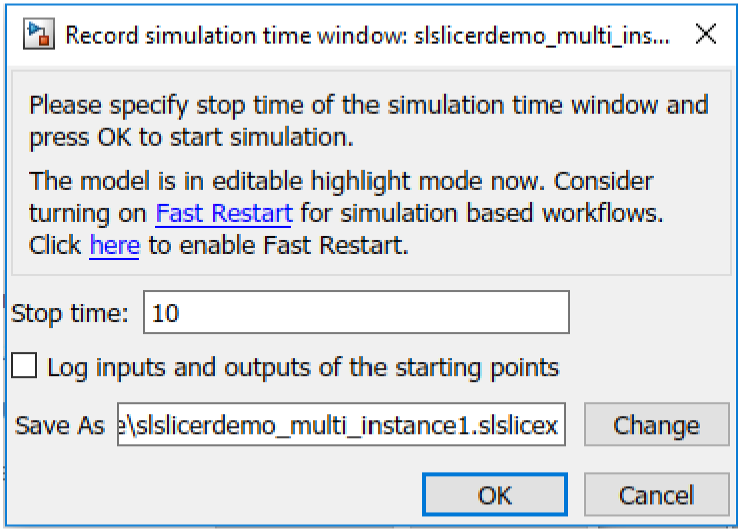
Run simulation

Use existing simulation data

Refine Dead Logic

Slicer Active

6. In the simulation time window, click **OK**. The model simulation starts.



Record simulation time window: slslicerdemo_multi_ins...

Please specify stop time of the simulation time window and press OK to start simulation.

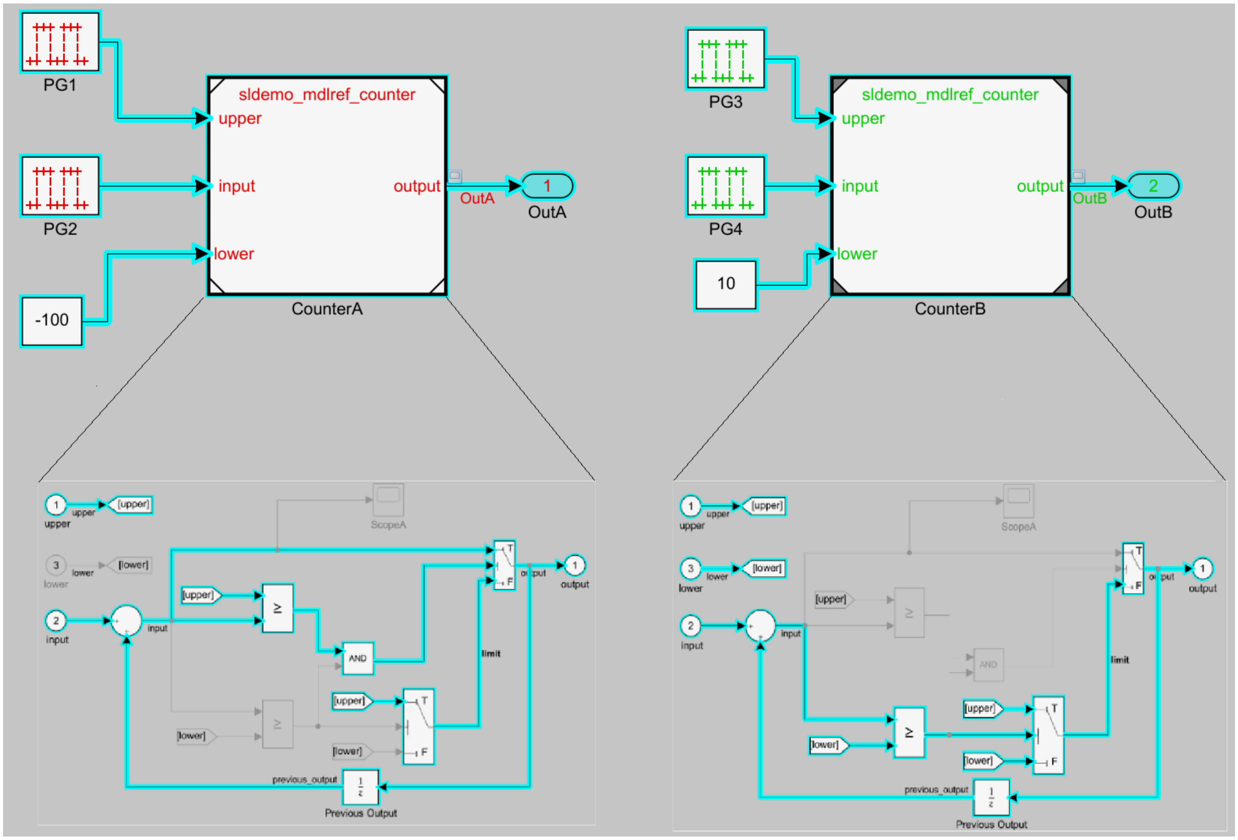
The model is in editable highlight mode now. Consider turning on Fast Restart for simulation based workflows. Click here to enable Fast Restart.

Stop time: 10

□ Log inputs and outputs of the starting points

Save As ɛ\slslicerdemo_multi_instance1.slslicex

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 11-2
- “Model Slicer Considerations and Limitations” on page 11-53

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

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.

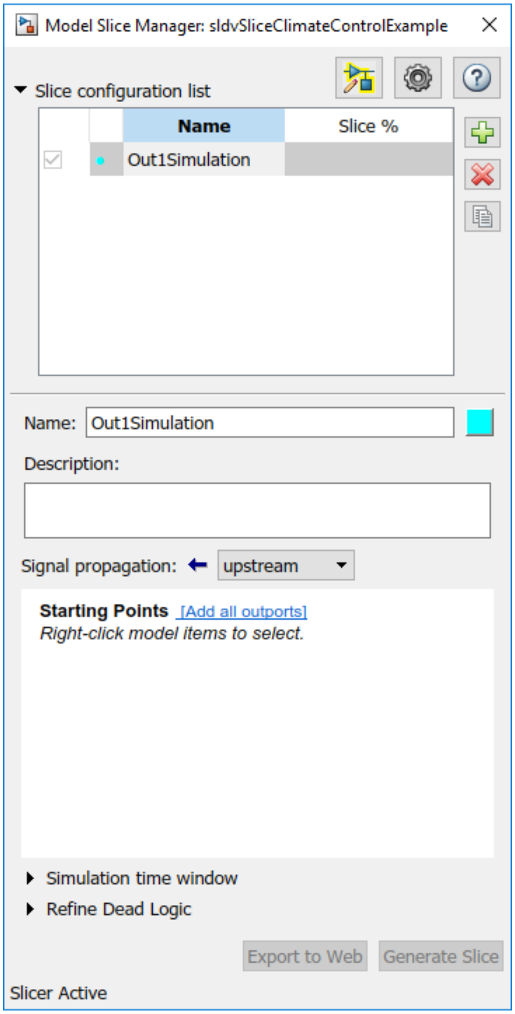
1. Add the example folder to the search path.

   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   ```
2. Open the `sldvSliceClimateControlExample` model.

   ```
   sldvSliceClimateControlExample
   ```
3. Select **Analysis > Model Slicer** to open the Model Slice Manager.

   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`


Model Slice Manager: sldvSliceClimateControlExample

Slice configuration list

		Name	Slice %
☑	●	Out1Simulation	

Name: Out1Simulation

Description:

Signal propagation: ← upstream

Starting Points [Add all outports]
Right-click model items to select.

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**.

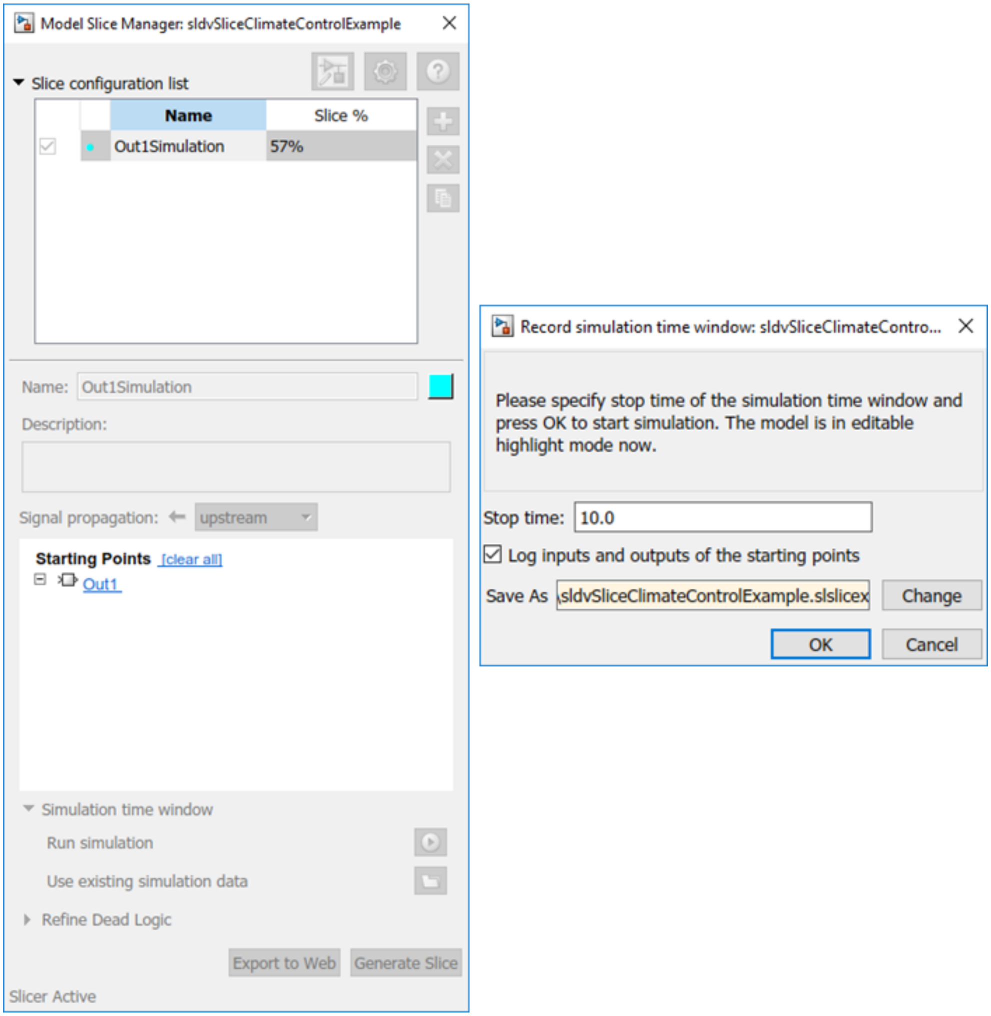
The model is highlighted.

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.



Model Slice Manager: sldvSliceClimateControlExample

Slice configuration list

		Name	Slice %
☑	●	Out1Simulation	57%

Name: Out1Simulation

Description:

Signal propagation: ← upstream

Starting Points [clear all]

⊟ Out1

Simulation time window

Run simulation

Use existing simulation data

Refine Dead Logic

Slicer Active

Record simulation time window: sldvSliceClimateContro...

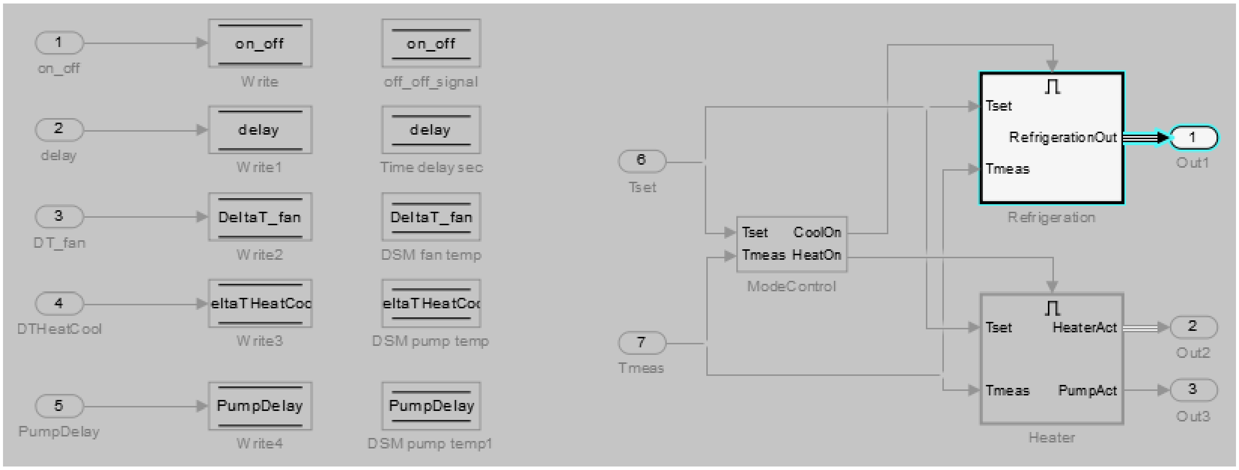
Please specify stop time of the simulation time window and press OK to start simulation. The model is in editable highlight mode now.

Stop time: 10.0

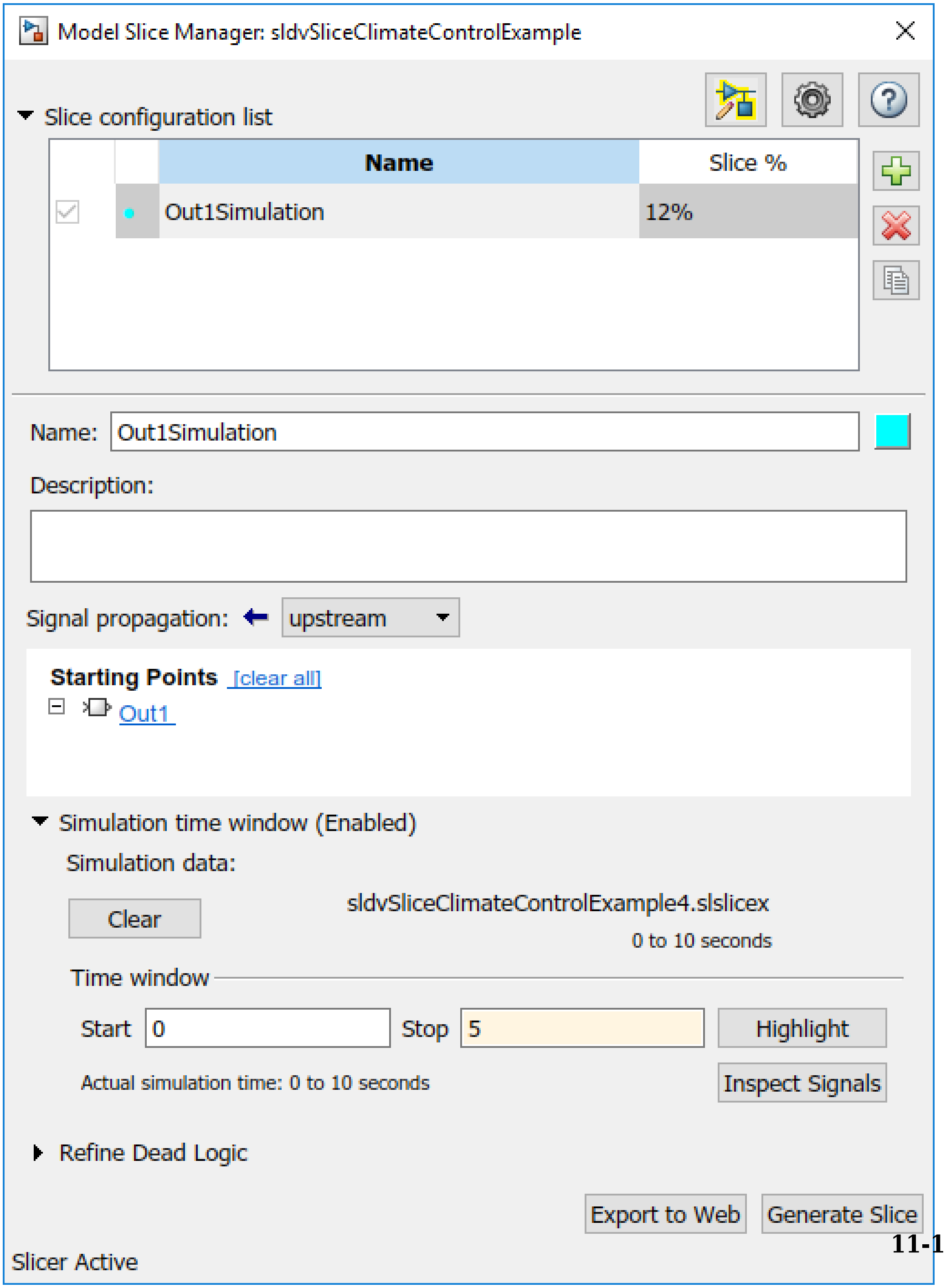
☑ Log inputs and outputs of the starting points

Save As \sldvSliceClimateControlExample.slslicex

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**.



Model Slice Manager: sldvSliceClimateControlExample

Slice configuration list

		Name	Slice %
☑	●	Out1Simulation	12%

Name: Out1Simulation

Description:

Signal propagation: upstream

Starting Points [clear all]

Out1

Simulation time window (Enabled)

Simulation data:

sldvSliceClimateControlExample4.slslicex

0 to 10 seconds

Time window

Start 0

Stop 5

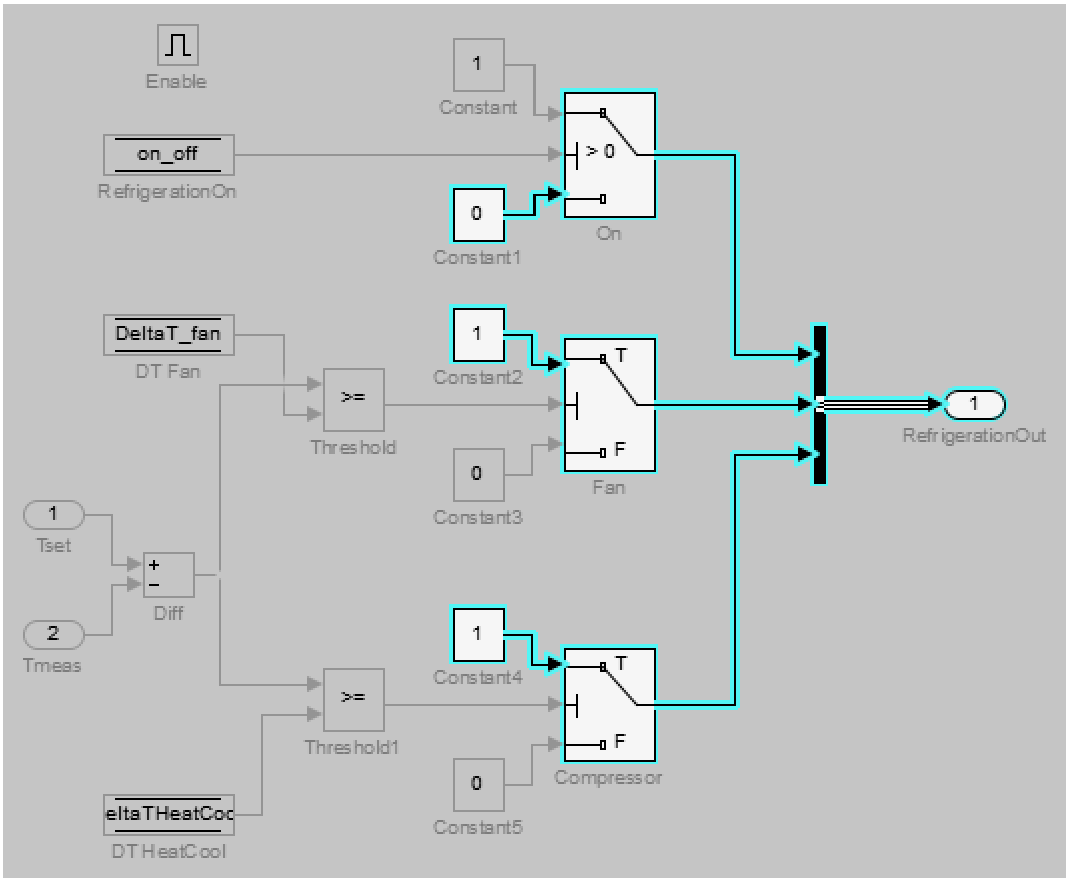
Actual simulation time: 0 to 10 seconds

Refine Dead Logic

Slicer Active

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 11-33 incorporating the highlighted portion of your model.

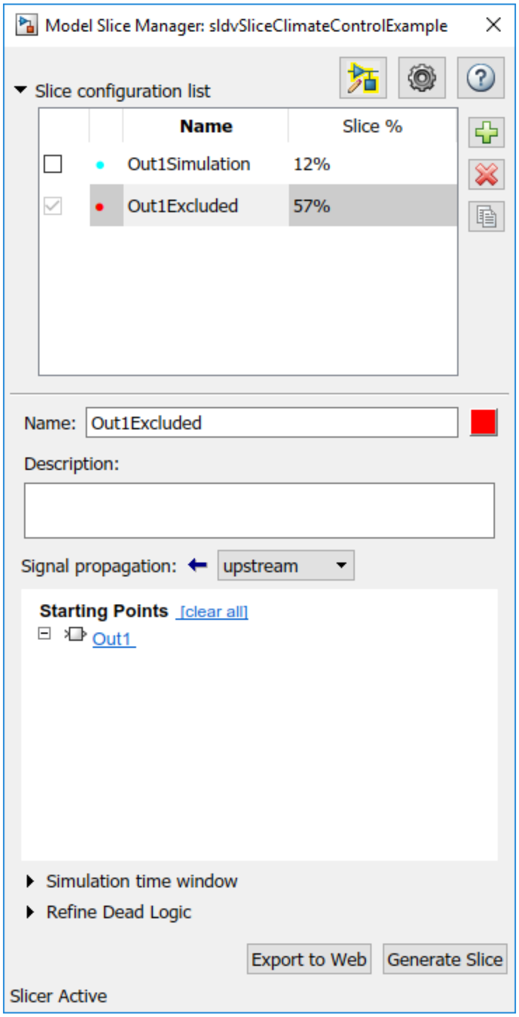
Exclude Blocks

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**.



		Name	Slice %
□	●	Out1Simulation	12%
☑	●	Out1Excluded	57%

Name: Out1Excluded 🟥

Description:

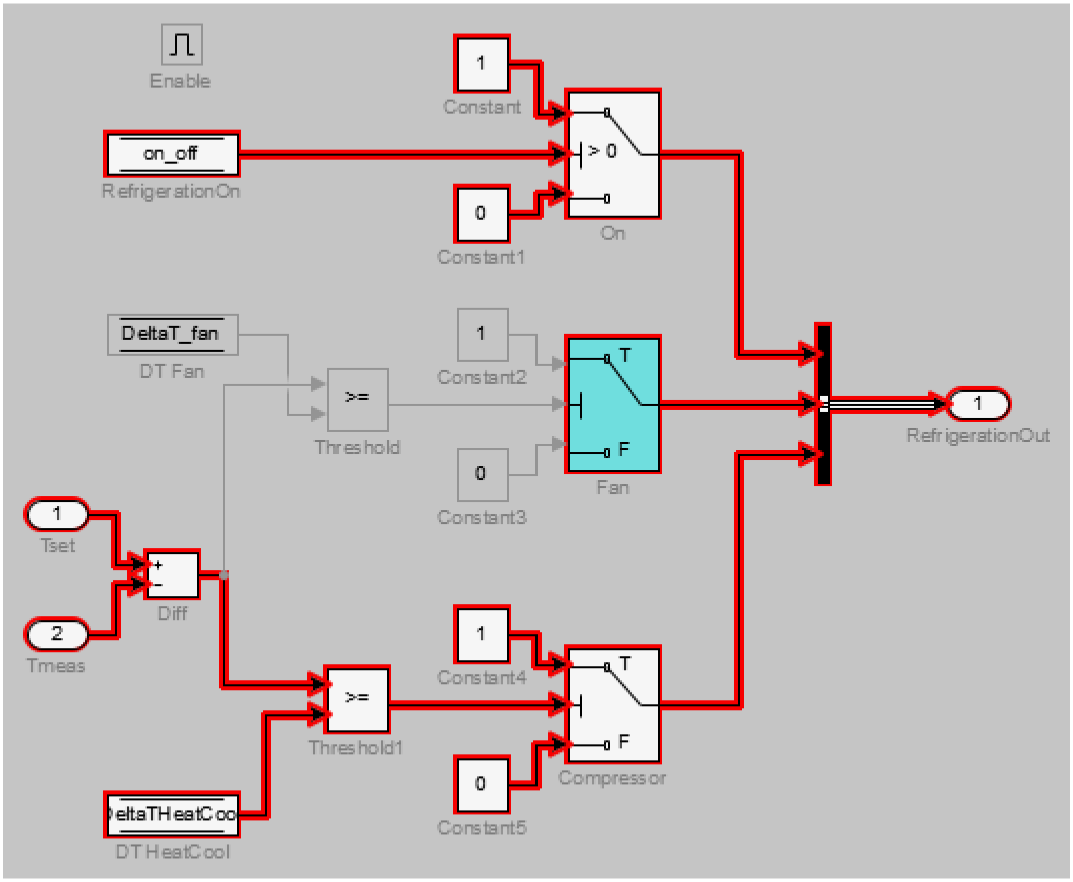
Signal propagation: ← upstream

The model is highlighted.

5 To open the subsystem, double-click `Refrigeration`.

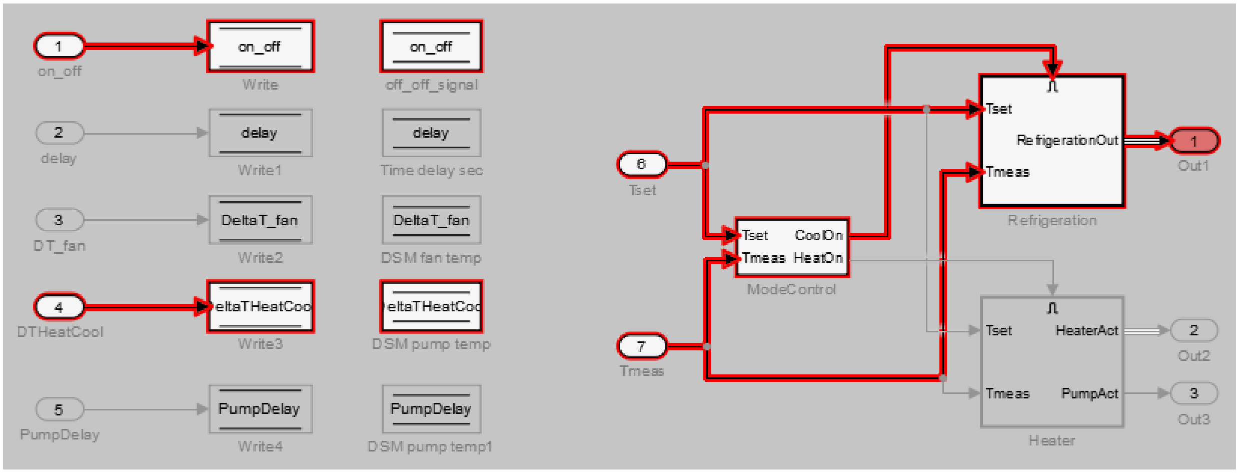
6 Right-click the `Fan` switch block, and then select **Model Slicer > Add as Exclusion Point**.

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.



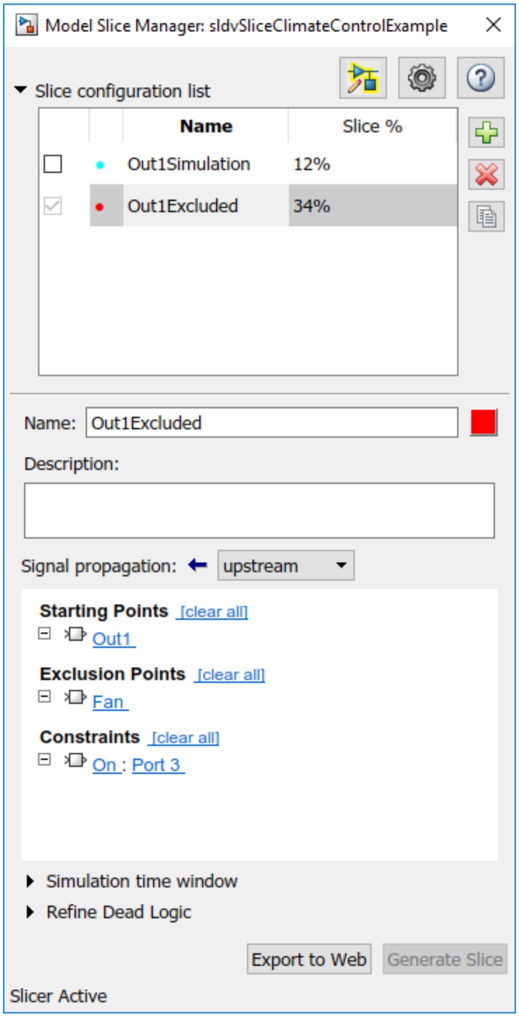
Exclude Inputs of a Switch Block

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**.



Model Slice Manager: sldvSliceClimateControlExample

Slice configuration list

		Name	Slice %
□	●	Out1Simulation	12%
☑	●	Out1Excluded	34%

Name: Out1Excluded 🟥

Description:

Signal propagation: upstream

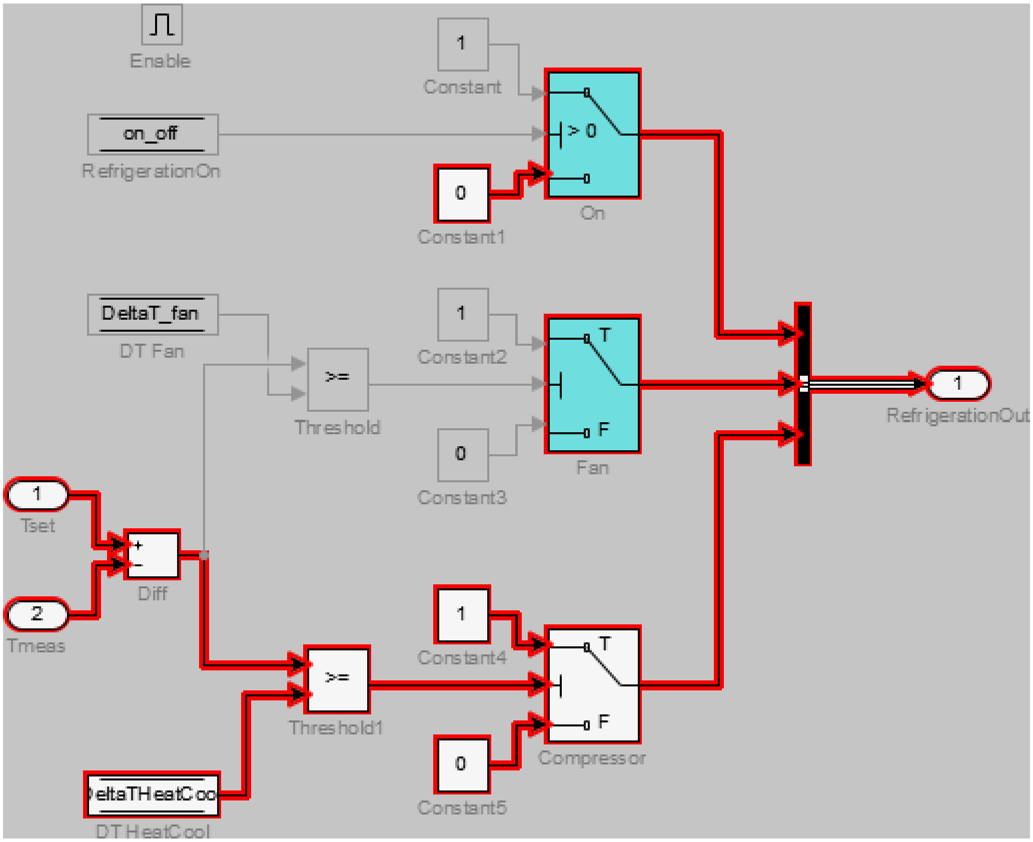
Starting Points [clear all]
Out1

Exclusion Points [clear all]
Fan

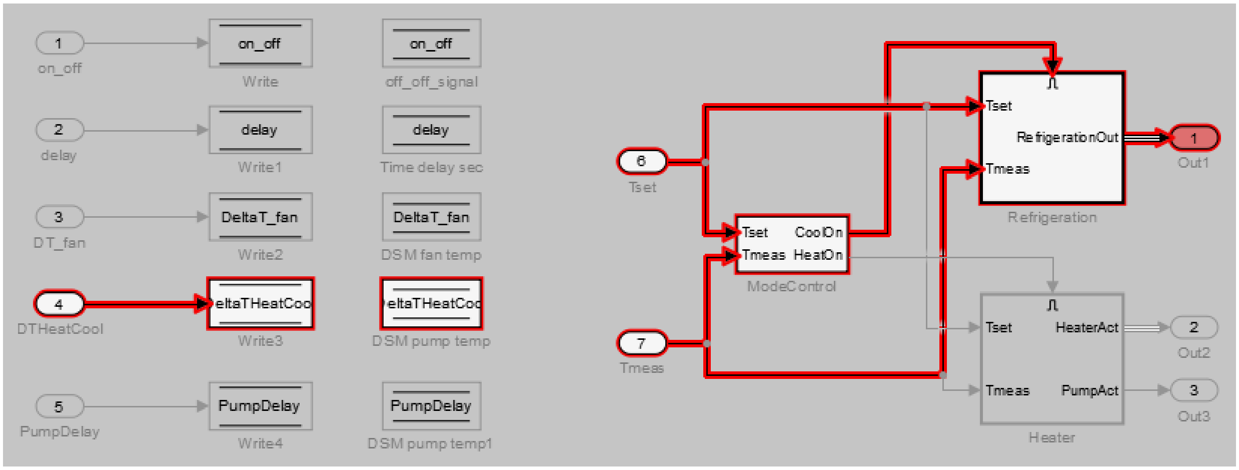
Constraints [clear all]
On : Port 3

Slicer Active

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 11-33
- “Model Slicer Considerations and Limitations” on page 11-53

Refine Dead Logic for Dependency Analysis

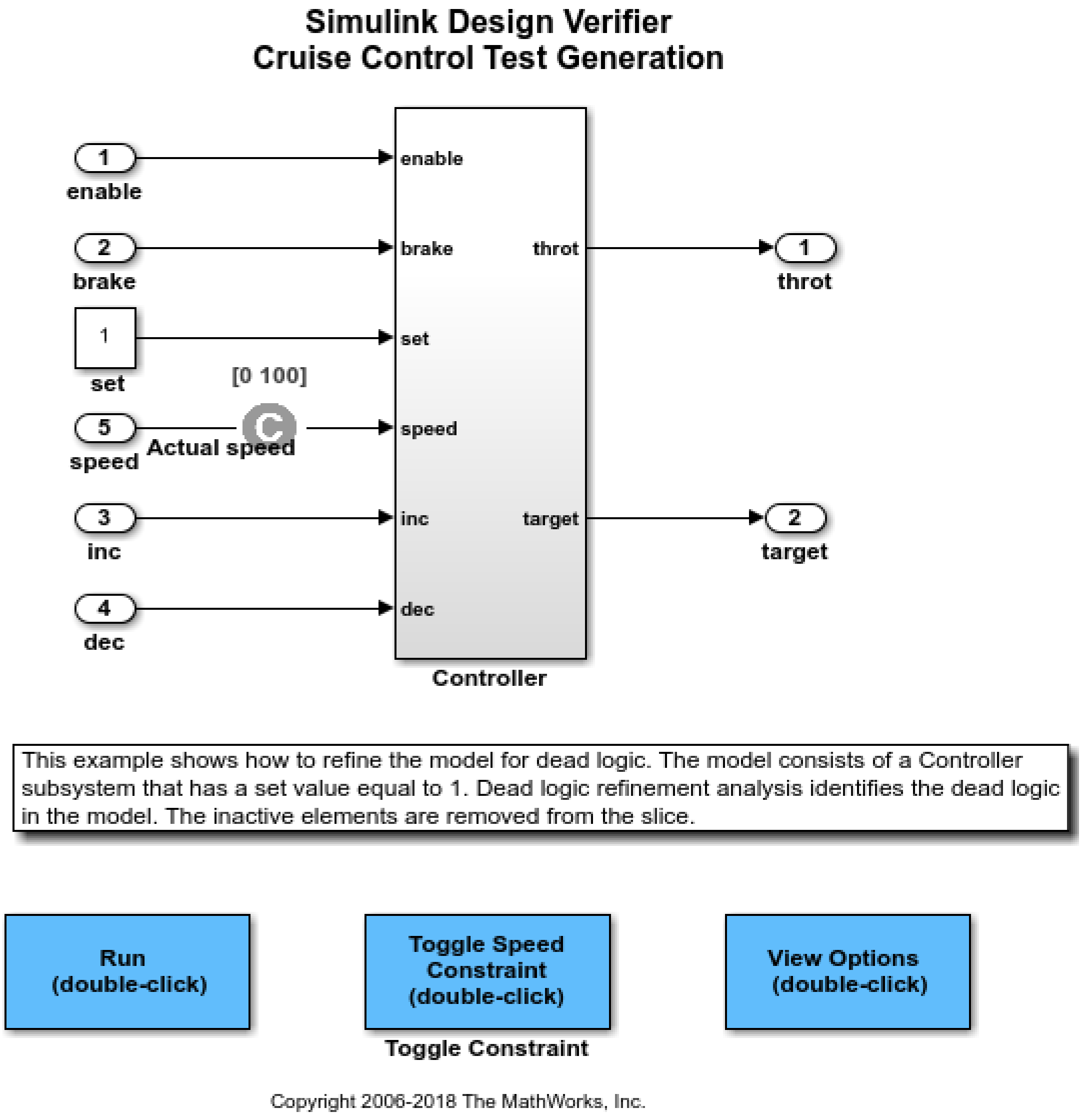
To refine the dead logic in your model for dependency analysis, use the Model Slicer. To provide an accurate slice, Model Slicer leverages Simulink Design Verifier dead logic analysis to remove the unreachable paths in the model. Model Slicer identifies the dead logic and refines the model slice for 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.

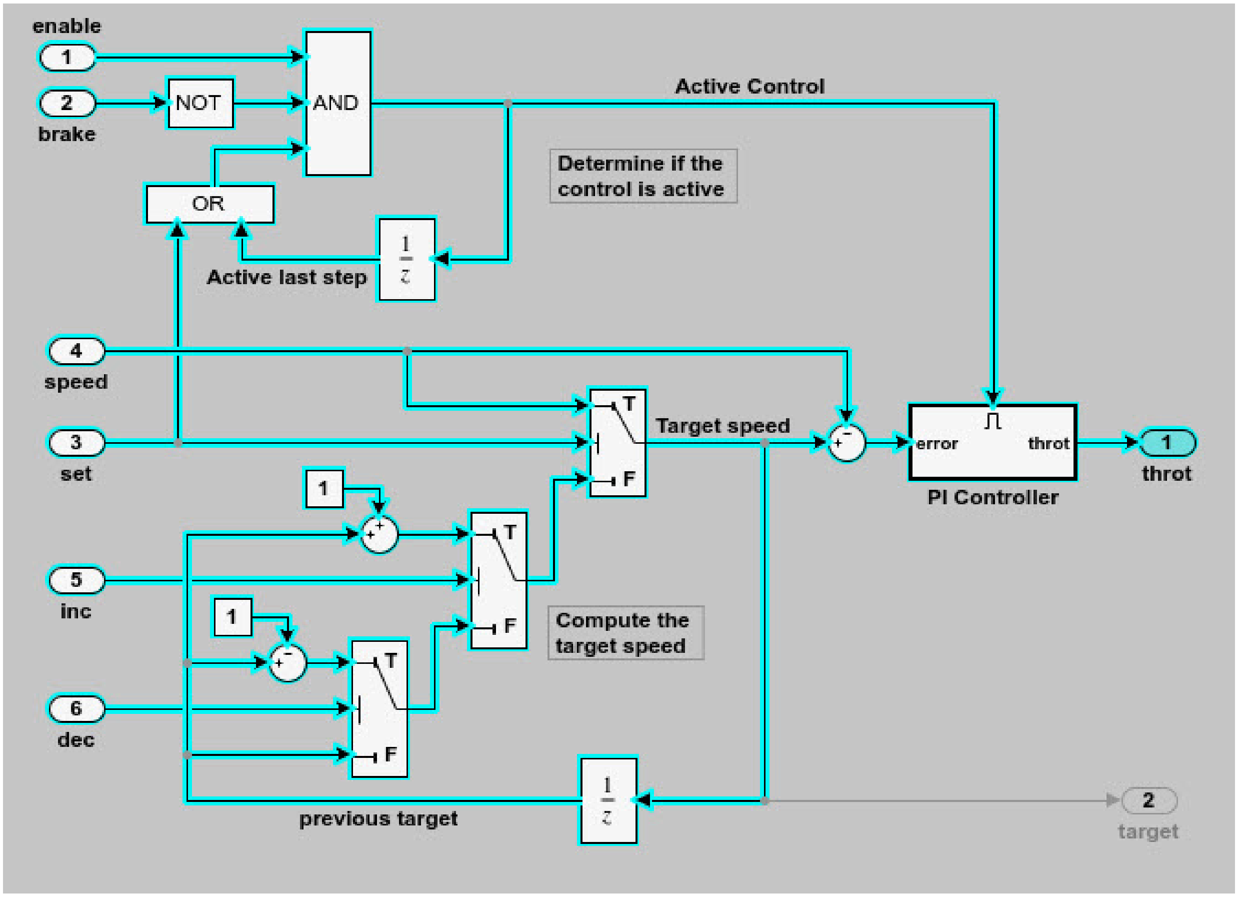
1. Open the `sldvSlicerdemo_dead_logic` model, and then select **Analysis** > **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```



This example shows how to refine the model for dead logic. The model consists of a Controller subsystem that has a set value equal to 1. Dead logic refinement analysis identifies the dead logic in the model. The inactive elements are removed from the slice.

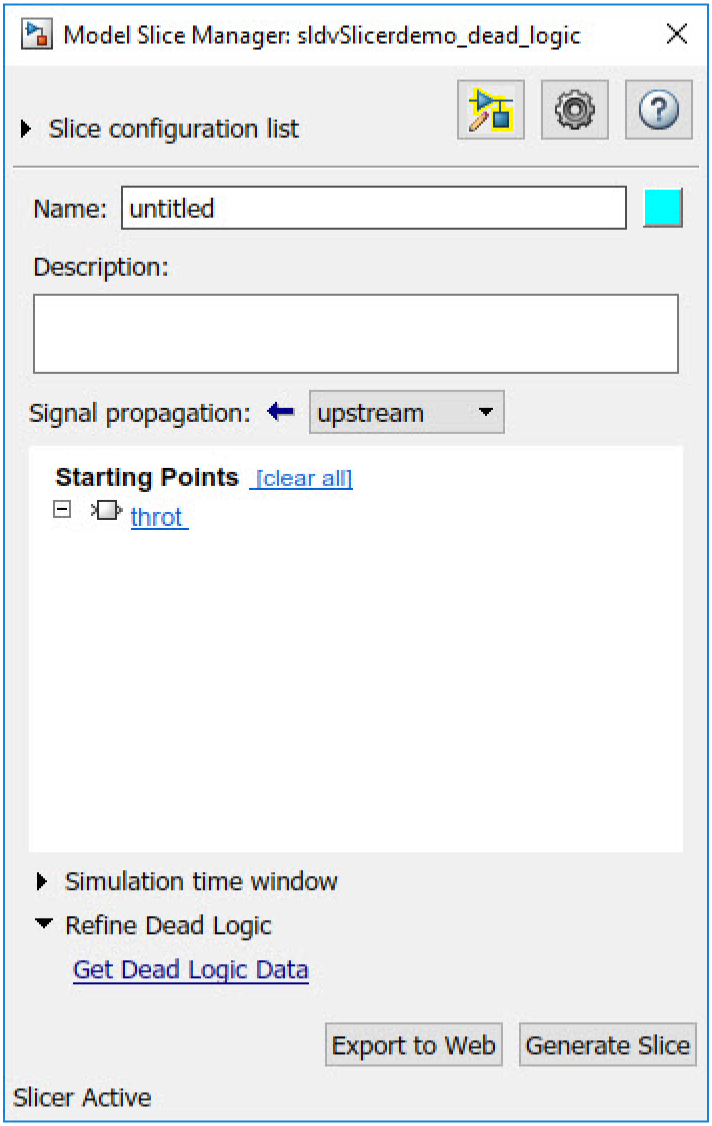
Open the `Controller` subsystem and add the outport `throt` as the starting point.



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**.



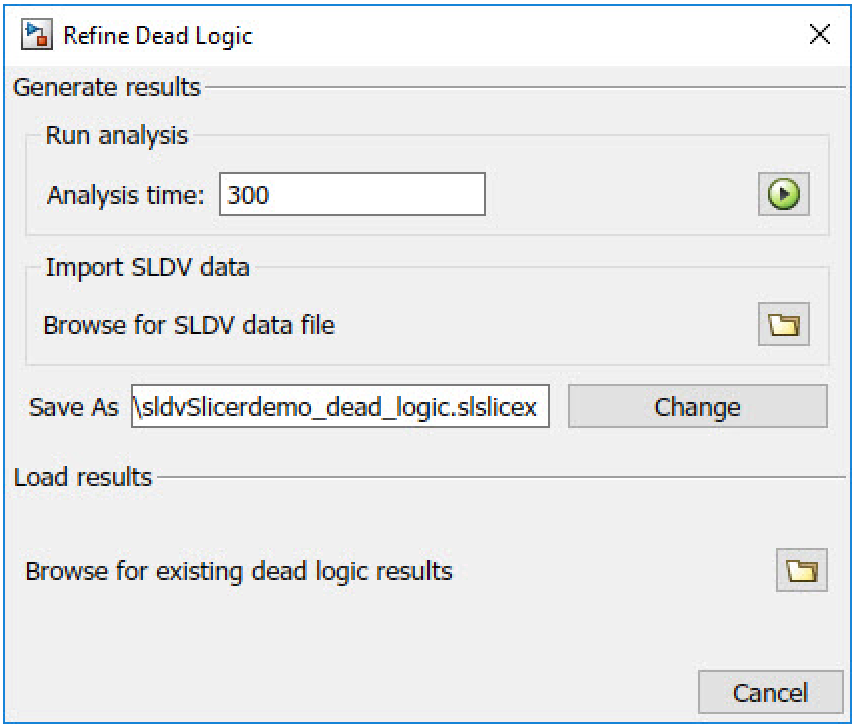
Model Slice Manager: sldvSlicerdemo_dead_logic

Name: untitled

Description:

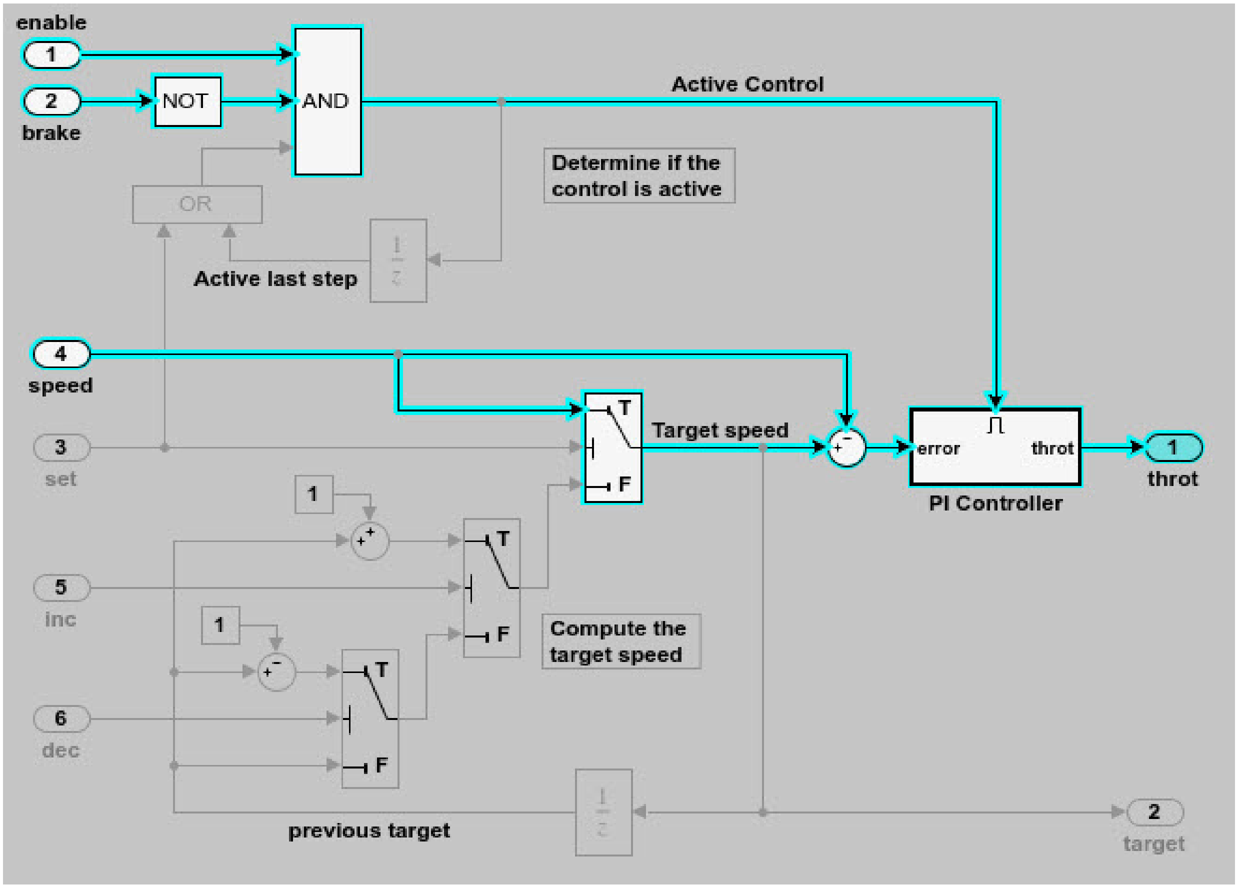
Signal propagation: upstream

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 11-78.



Analysis time: 300

Save As \sldvSlicerdemo_dead_logic.slslicex



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 11-13

- “Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results” on page 11-78

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 11-2 and “Refine Highlighted Model” on page 11-13.
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 11-53 for compilation considerations.

See Also

More About

- “Configure Model Highlight and Sliced Models” on page 11-49

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

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 11-2 by using Model Slicer.

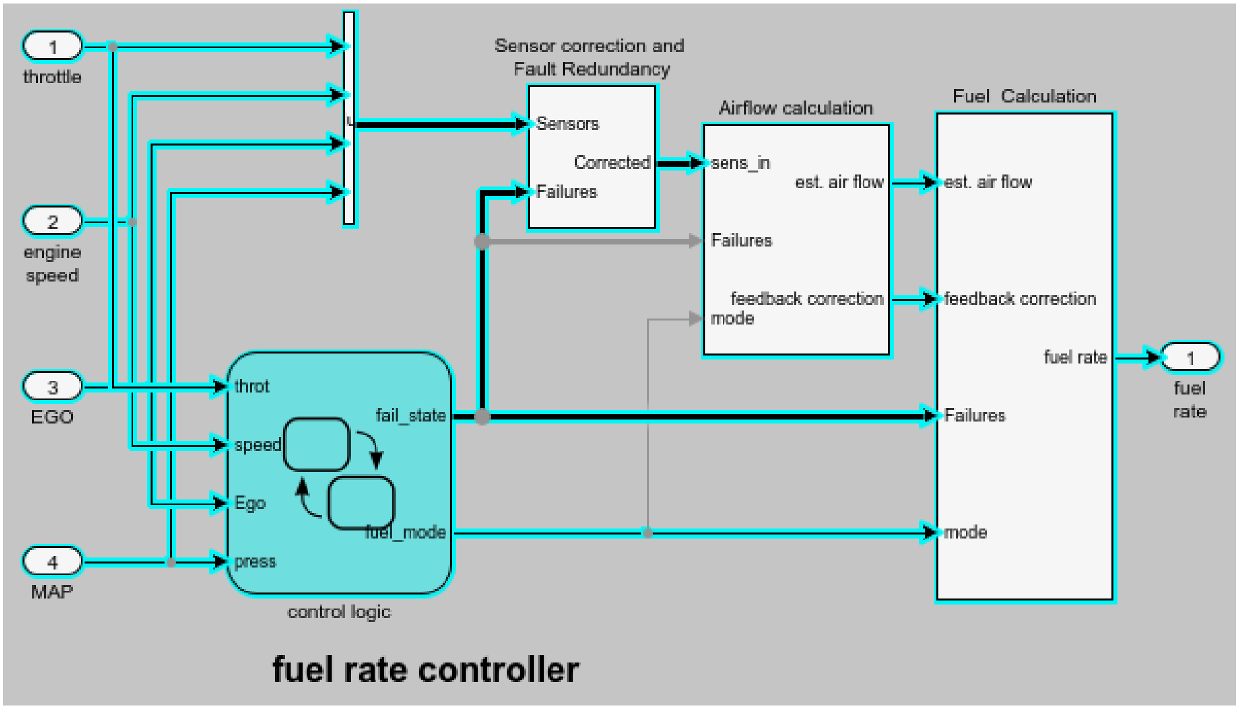
Create a Dynamic Slice Highlight for an Area of Interest

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.



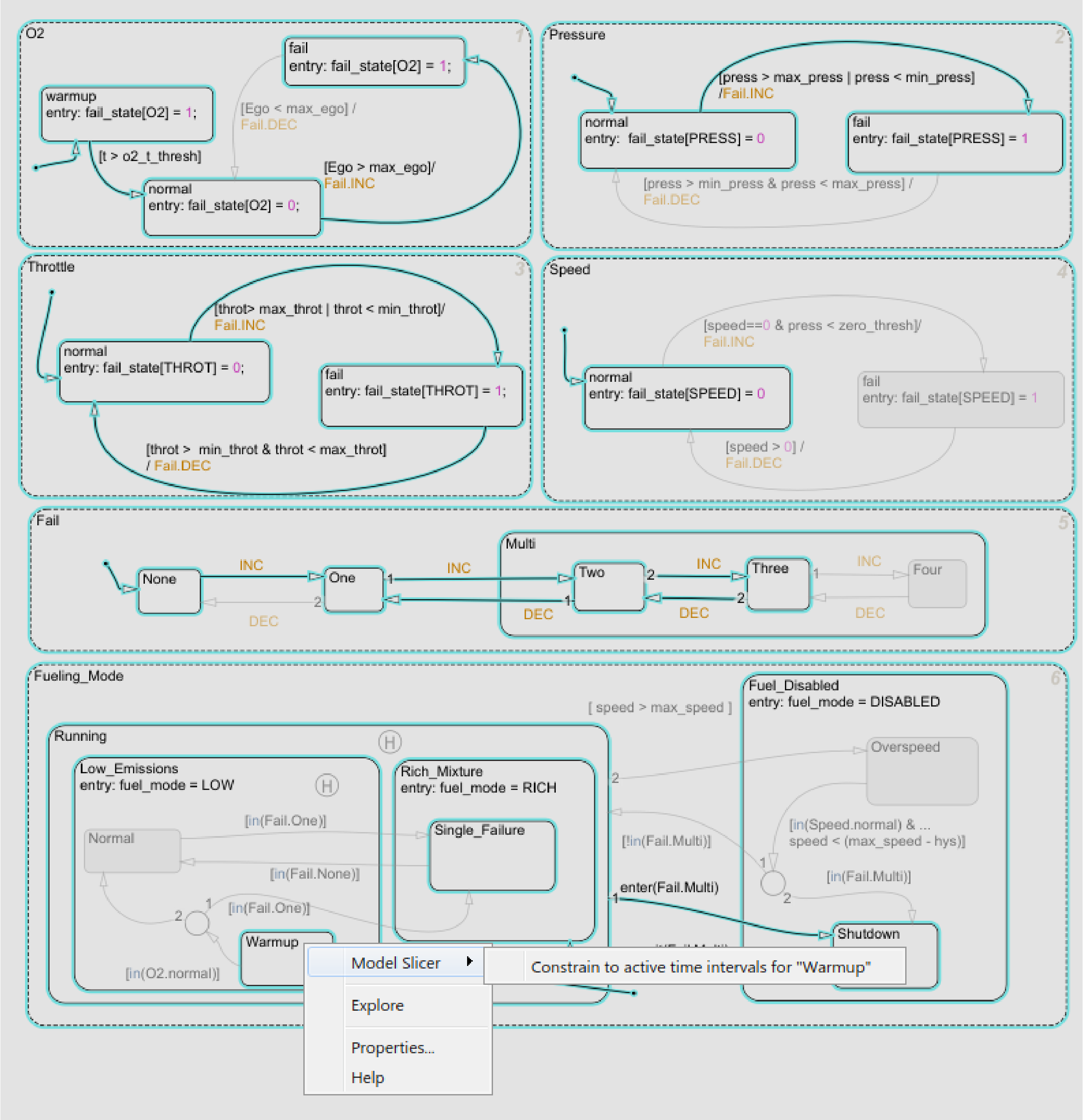
Constrain the Model Highlight to the Active Time Interval of a Stateflow State

1 To open the Model Slice Manager, from the Simulink menu, select **Analysis > Model Slicer** .

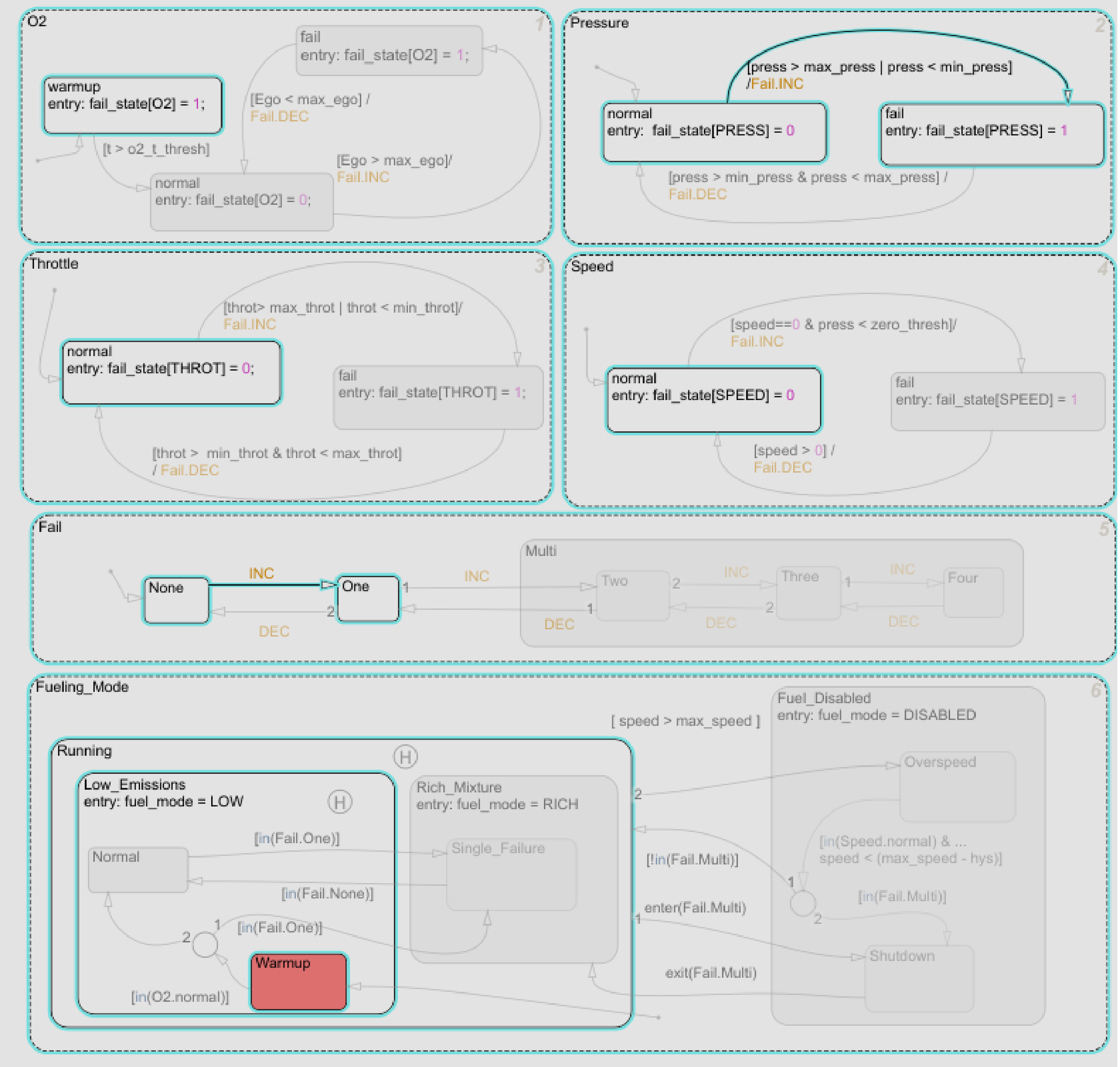
2 Navigate to the `control logic` Stateflow chart in the `fuel rate controller` subsystem.

```
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”**.


O2
1
fail
entry: fail_state[O2] = 1;
warmup
entry: fail_state[O2] = 1;
[Ego < max_ego] /
Fail.DEC
[t > o2_t_thresh]
[Ego > max_ego]/
Fail.INC
normal
entry: fail_state[O2] = 0;
Pressure
2
[press > max_press | press < min_press]
/Fail.INC
normal
entry: fail_state[PRESS] = 0
fail
entry: fail_state[PRESS] = 1
[press > min_press & press < max_press] /
Fail.DEC
Throttle
3
[throt> max_throt | throt < min_throt]/
Fail.INC
normal
entry: fail_state[THROT] = 0;
fail
entry: fail_state[THROT] = 1;
[throt > min_throt & throt < max_throt]
/ Fail.DEC
Speed
4
[speed==0 & press < zero_thresh]/
Fail.INC
normal
entry: fail_state[SPEED] = 0
fail
entry: fail_state[SPEED] = 1
[speed > 0] /
Fail.DEC
Fail
5
Multi
None
INC
One
1
2
INC
Two
2
1
INC
Three
2
1
INC
Four
DEC
DEC
DEC
DEC
Fueling_Mode
6
[speed > max_speed]
Fuel_Disabled
entry: fuel_mode = DISABLED
Running
H
Low_Emissions
entry: fuel_mode = LOW
H
Rich_Mixture
entry: fuel_mode = RICH
2
Overspeed
Normal
[in(Fail.One)]
Single_Failure
[!in(Fail.Multi)]
[in(Speed.normal) & ...
speed < (max_speed - hys)]
[in(Fail.None)]
1
[in(Fail.Multi)]
2
1
[in(Fail.One)]
enter(Fail.Multi)
1
2
Warmup
Shutdown
[in(O2.normal)]
Model Slicer
Constrain to active time intervals for "Warmup"
Explore
Properties...
Help

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.



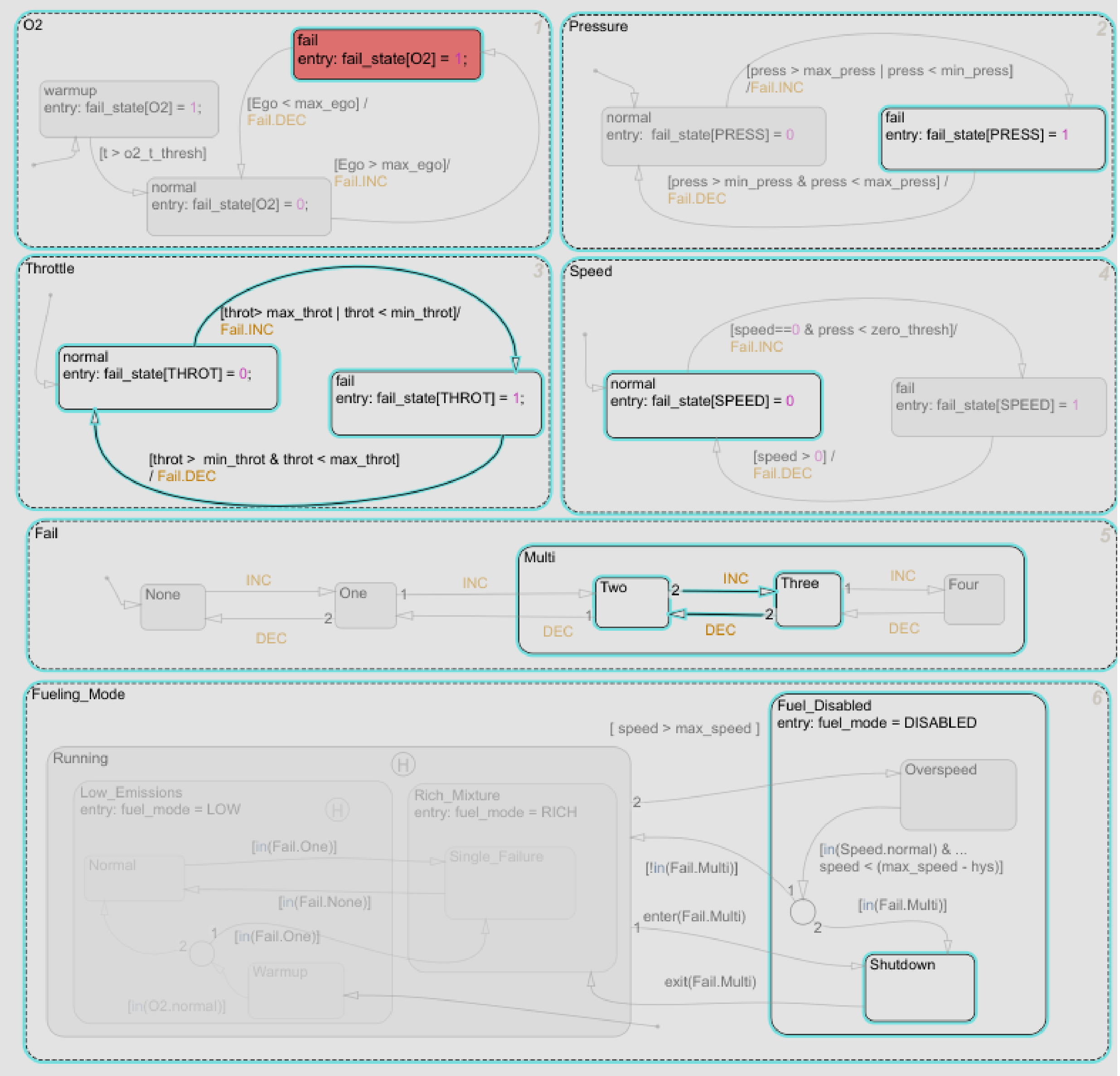
The Model Slice Manager is also updated to show the time interval in which the `warmup` state is active:

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.

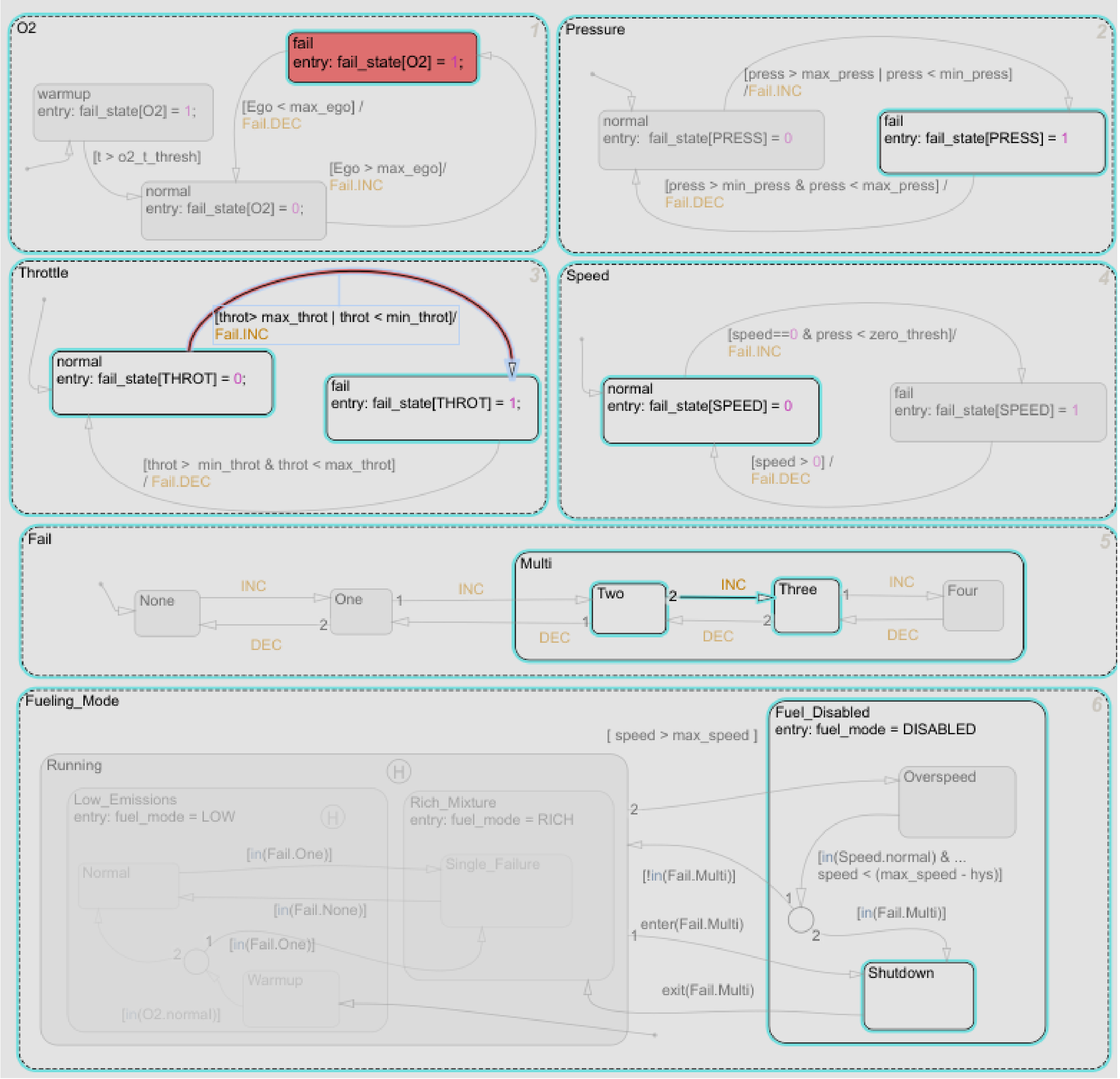


Model Slicer is updated to highlight only those portions of the model that are active during the time intervals in which the **O2 > fail** state is active. The Model Slice

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 11-53.

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 11-61
- “States” (Stateflow)
- “Transitions” (Stateflow)

Simplify a Standalone Model by Inlining Content

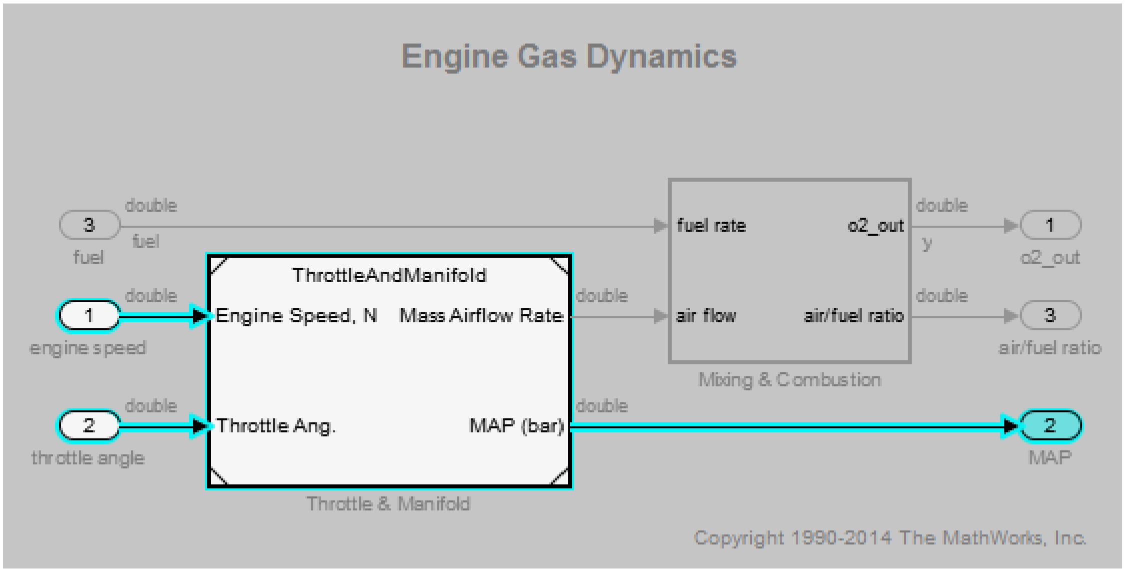
You can reduce file dependencies by inlining model content when you generate the sliced model. Inlining brings functional content into the sliced model and can eliminate model references, library links, and variant structures that are often not needed for model refinement or debugging.

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 11-51.

This example demonstrates inlining content of a model referenced by a Model block.

1. Add the path to the example and open the model

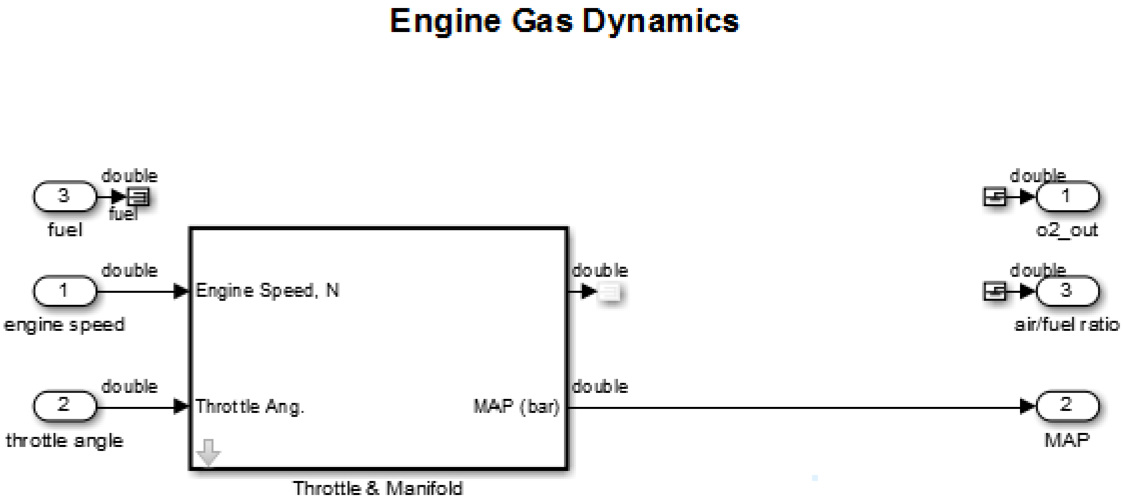
   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'))
   open_system('sldvSliceEngineDynamicsExample')
   ```
2. From the menu, select **Analysis > Model Slicer** to open the Model Slice Manager.
3. In the model, right-click the MAP outport and select **Model Slicer > Add as Starting Point**. The path is highlighted through the Model block.



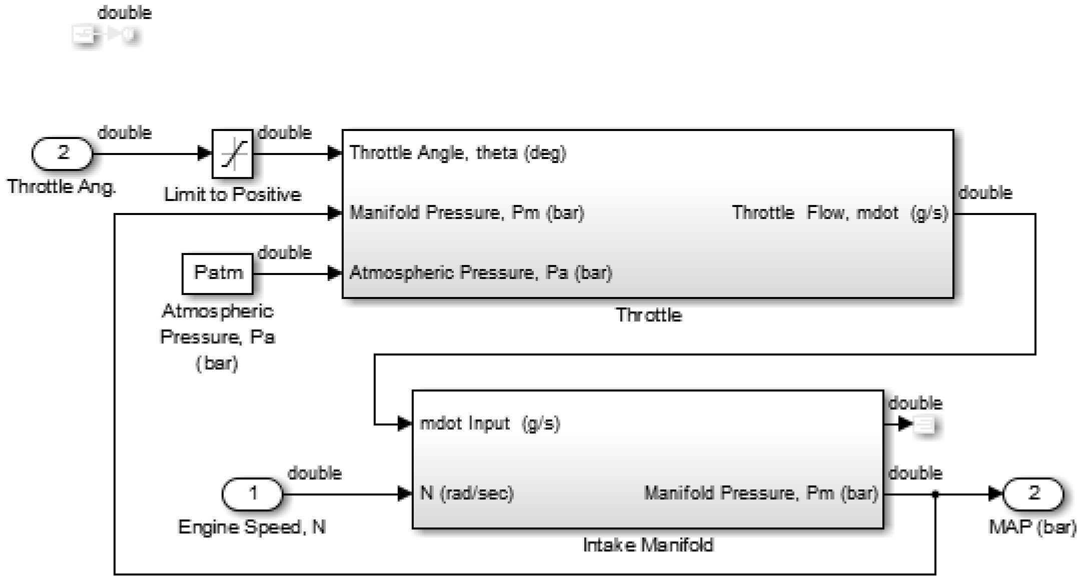
4 Create a sliced model from the highlight. In the Model Slice Manager, click the **Generate slice** button.

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.


double
2
Throttle Ang.
double
Limit to Positive
double
Throttle Angle, theta (deg)
Manifold Pressure, Pm (bar)
Throttle Flow, mdot (g/s)
double
Patm
double
Atmospheric Pressure, Pa (bar)
Atmospheric Pressure, Pa (bar)
Throttle
mdot Input (g/s)
double
1
double
N (rad/sec)
Manifold Pressure, Pm (bar)
double
2
Engine Speed, N
Intake Manifold
MAP (bar)

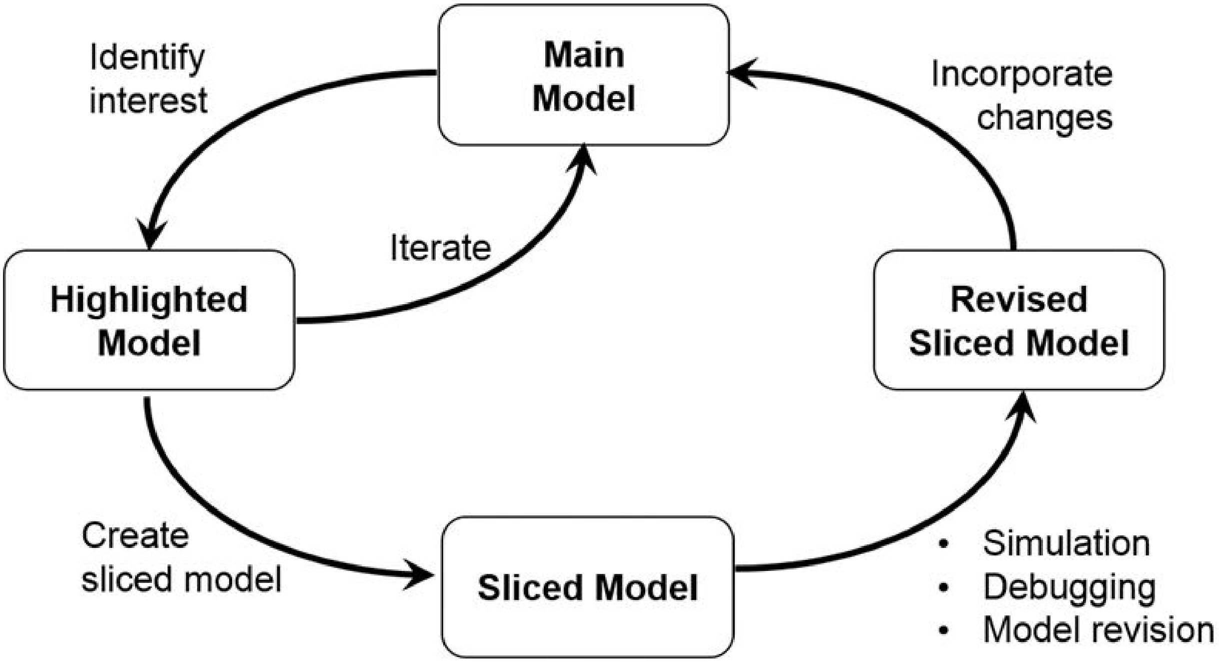
Workflow for Dependency Analysis

In this section...

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 11-2
- "Refine Highlighted Model" on page 11-13
- "Create a Simplified Standalone Model" on page 11-33

Configure Model Highlight and Sliced Models

In this section...

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 11-51.

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 11-2
- "Refine Highlighted Model" on page 11-13
- "Simplify a Standalone Model by Inlining Content" on page 11-43

Model Slicer Considerations and Limitations

When you work with the Model Slicer, consider these behaviors and limitations:

In this section...

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 11-51.

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 11-51.

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 “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 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 support multiple instances of the same Model block with its **Simulation mode** set to `Normal`. 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.

Block	Limitation
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 11-13 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 11-34, 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...

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 11-13, 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 11-13 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 11-61
"Using Model Slicer with Stateflow State Transition Tables" on page 11-62
"Support Limitations for Using Model Slicer with Stateflow" on page 11-62

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 11-59.

See Also

More About

- “Highlight Functional Dependencies” on page 11-2
- “Refine Highlighted Model” on page 11-13
- “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.

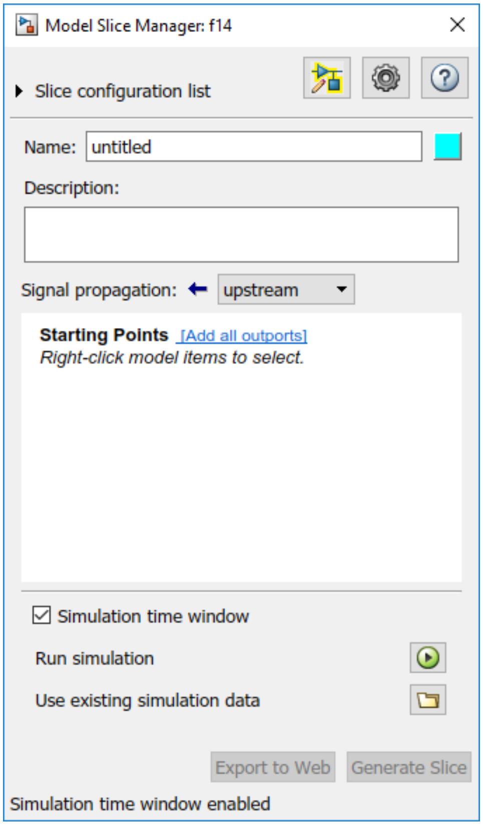
In this section...

Choose Starting Points and Direction

1 Open the `f14` example model.

```
f14
```

2 Select **Analysis > Model Slicer** to open the Model Slice Manager.



Model Slice Manager: f14

Name: untitled

Description:

Signal propagation: upstream

Starting Points [Add all outports]

Right-click model items to select.

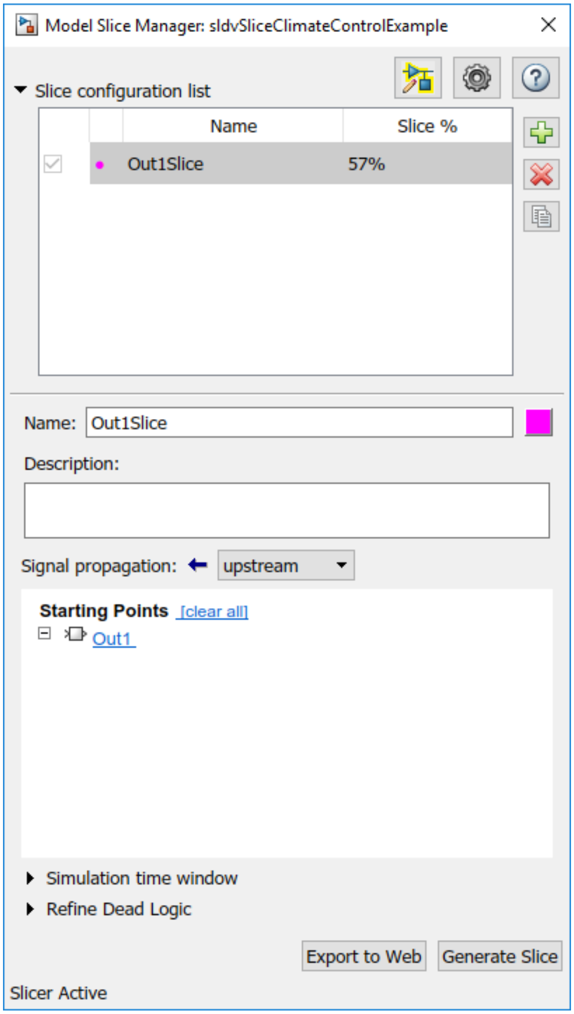
☑ Simulation time window

Simulation time window enabled

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**.



	Name	Slice %
☑	● Out1Slice	57%

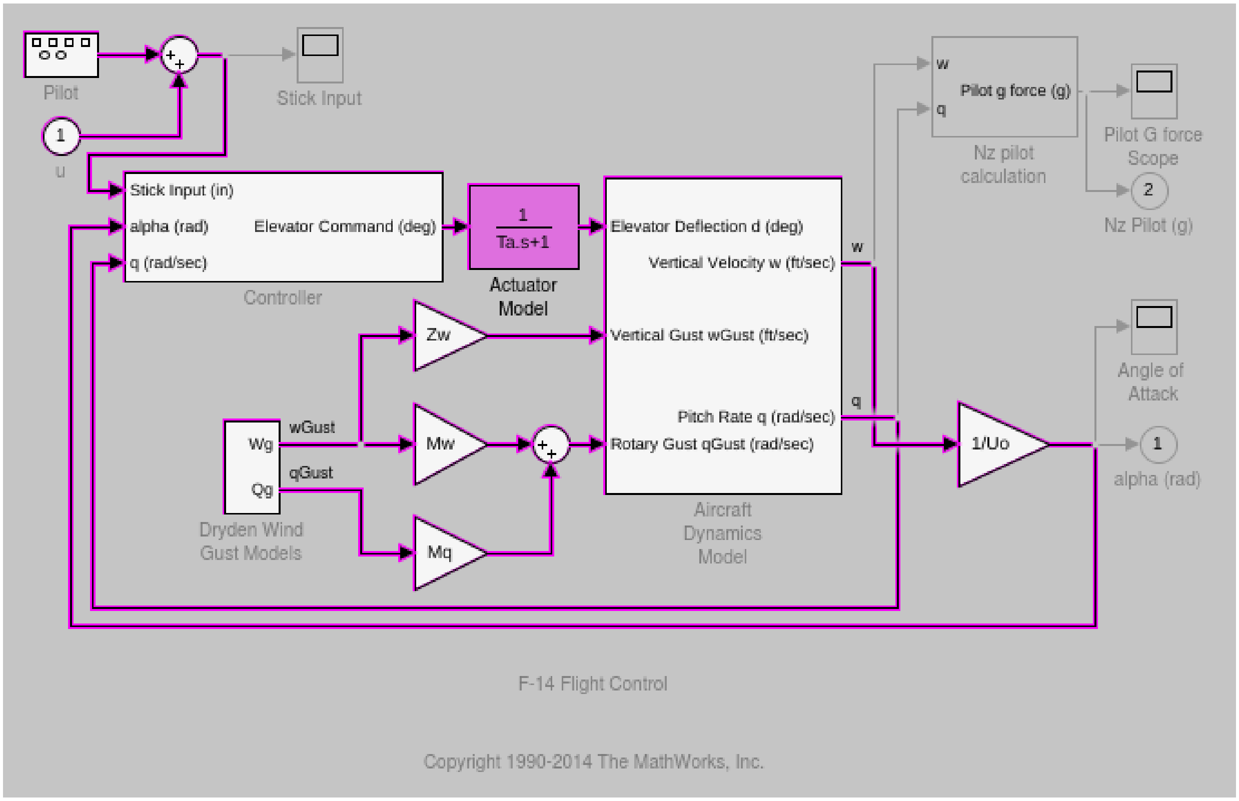
Name: Out1Slice

Description:

Signal propagation: upstream

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`.

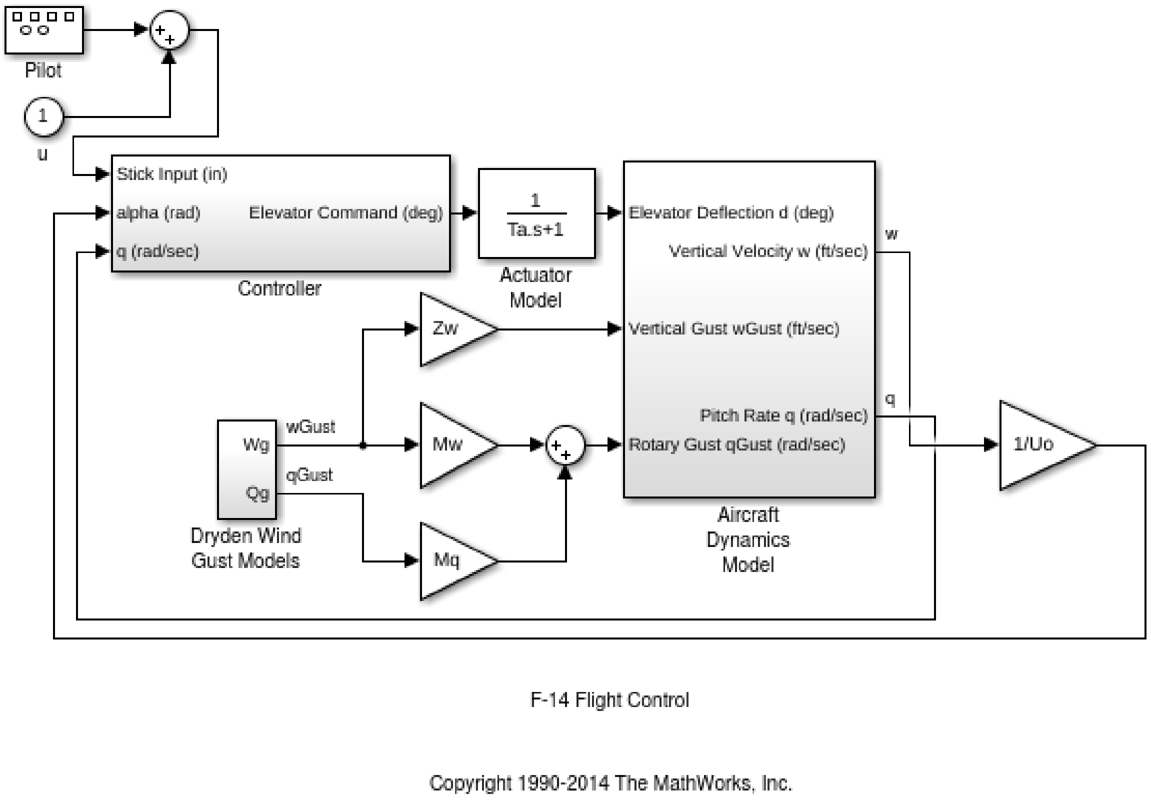
2 Generate a standalone model containing the highlighted model items:

   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.



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

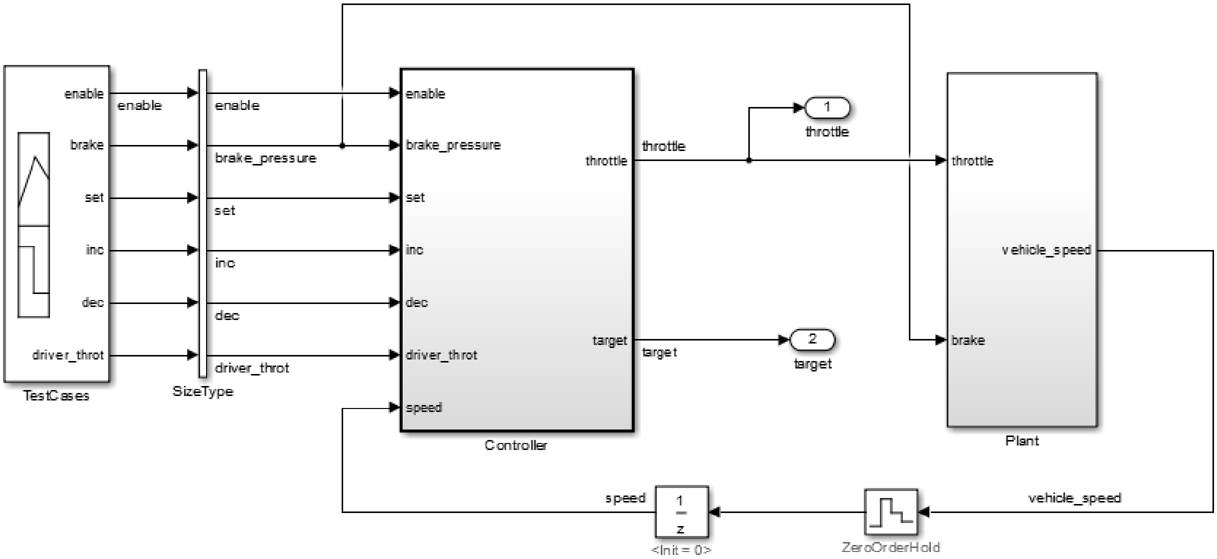
To debug and refine a subsystem of your model, create a standalone model. The standalone model isolates the subsystem and relevant signals. You can observe the subsystem behavior without simulating the entire source model.

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.

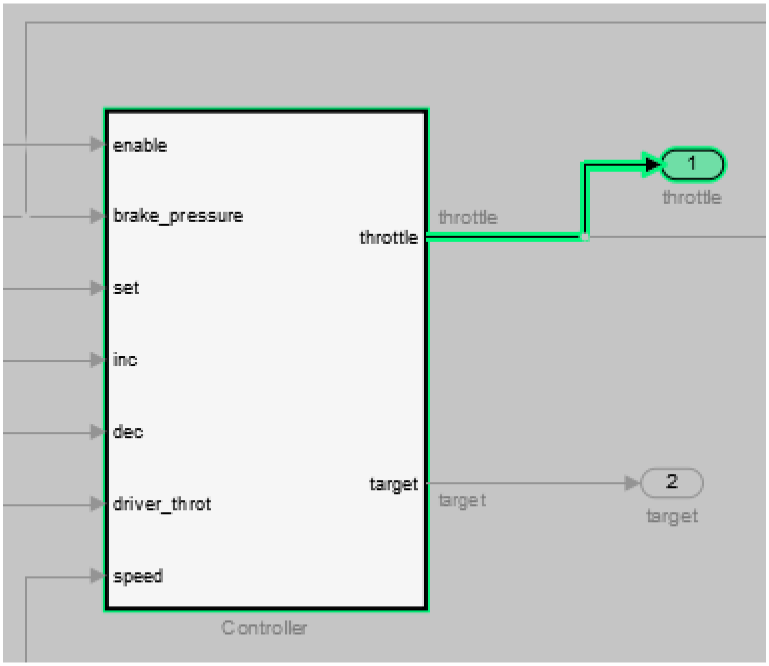


1 To open the Model Slice Manager, select **Analysis** > **Model Slicer**.

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.

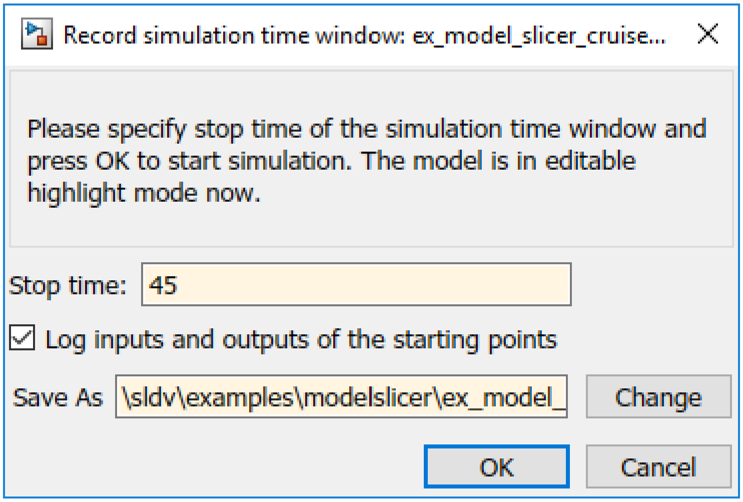


4 To specify the simulation time window:

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**.



Record simulation time window: ex_model_slicer_cruise...

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: 45

☑ Log inputs and outputs of the starting points

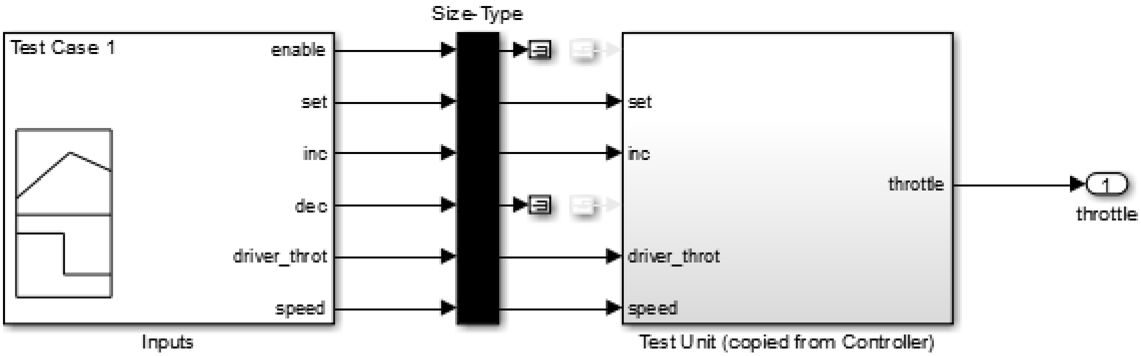
Save As \sldv\examples\modelslicer\ex_model_

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

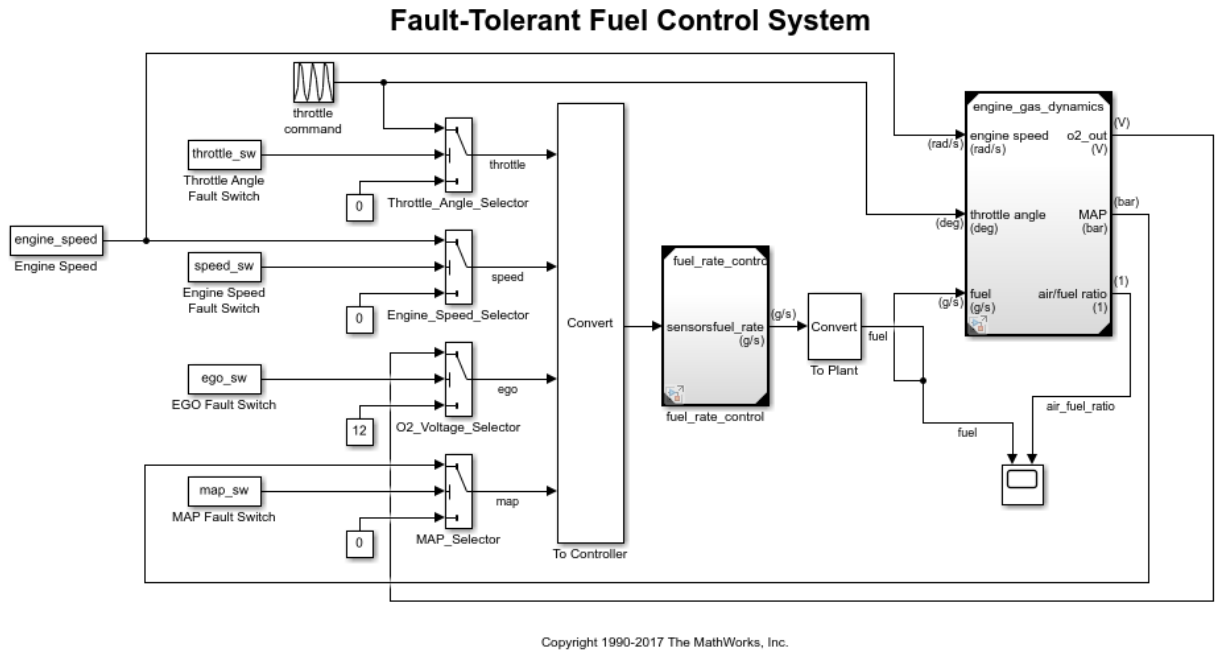
To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

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');
```



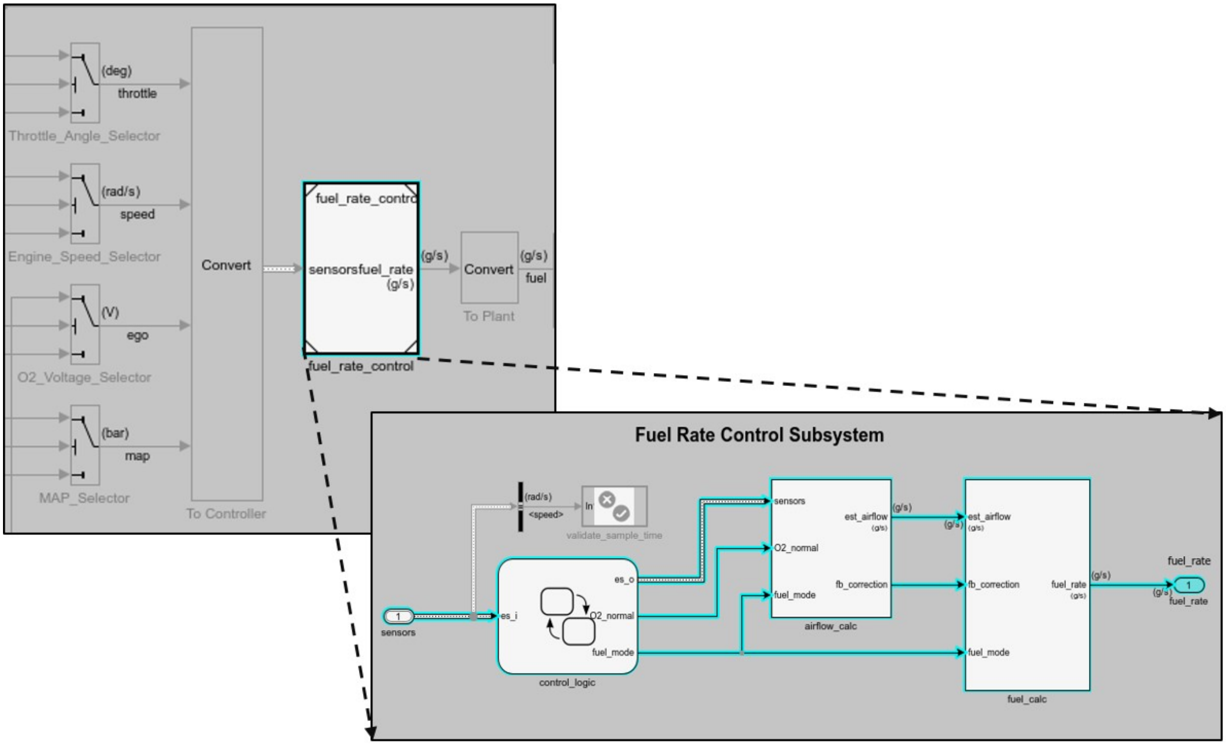
Step 2: Slice the Referenced Model

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 **Analysis** > **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`.



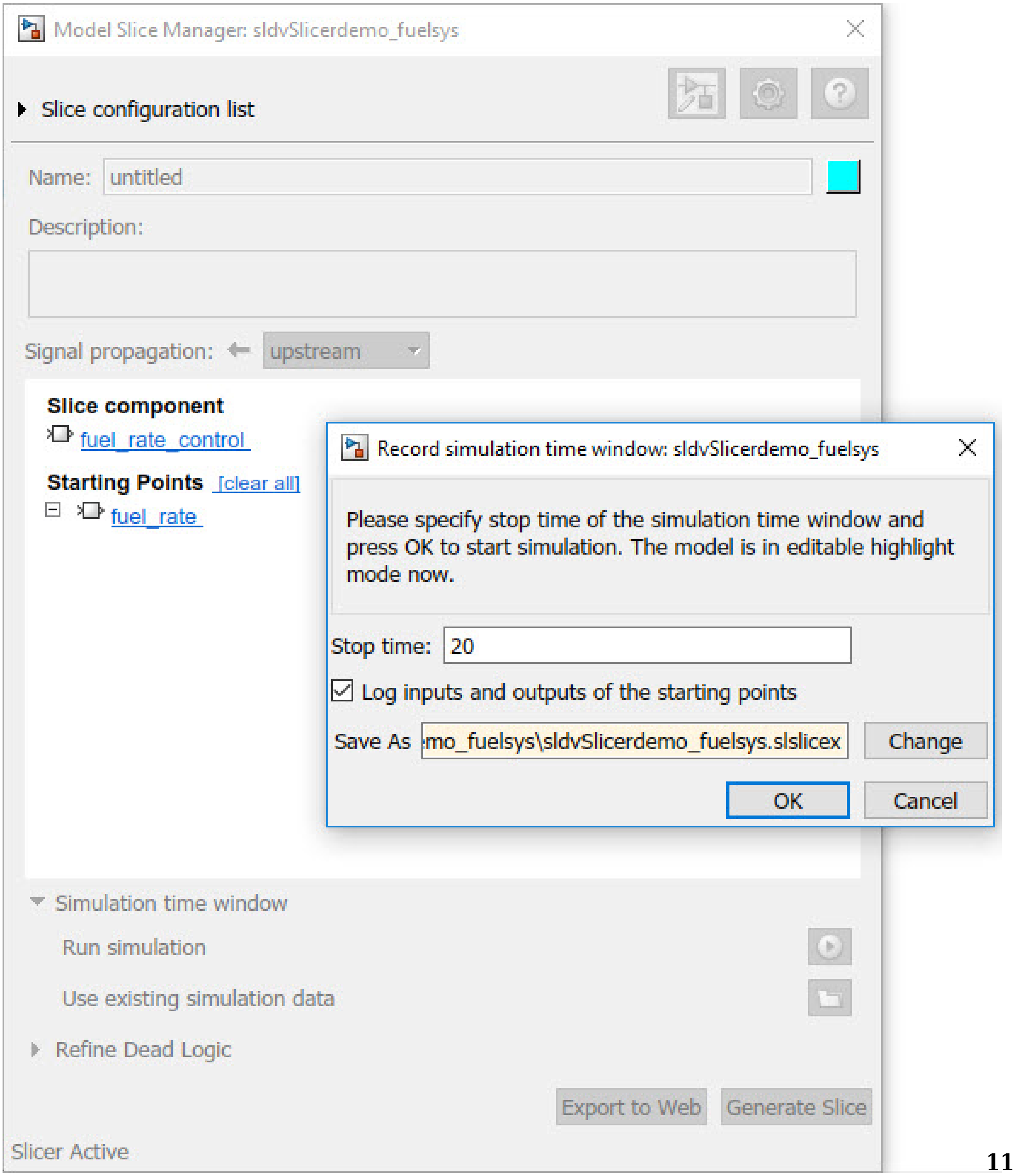
Step 4: Generate Slice

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.



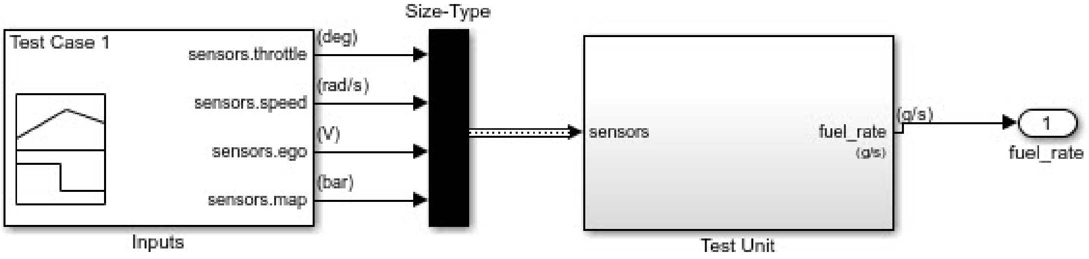
Please specify stop time of the simulation time window and press OK to start simulation. The model is in editable highlight mode now.

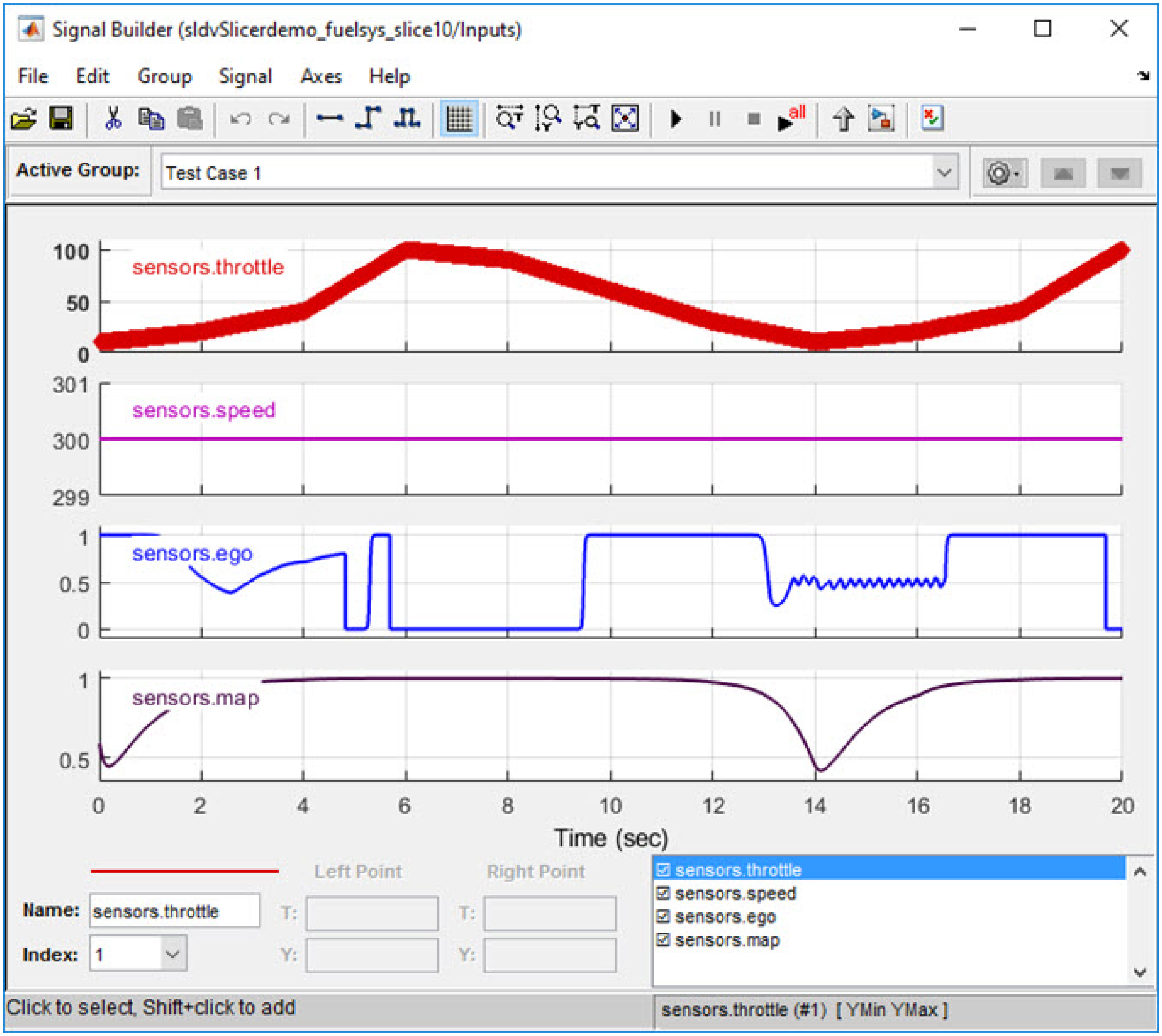
Stop time: 20

☑ Log inputs and outputs of the starting points

Save As mo_fuelsys\sldvSlicerdemo_fuelsys.slslicex

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.





Signal Builder (sldvSlicerdemo_fuelsys_slice10/Inputs)

Active Group: Test Case 1

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]

See Also

“Model Slicer Considerations and Limitations” on page 11-53 | “Highlight Functional Dependencies” on page 11-2

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.

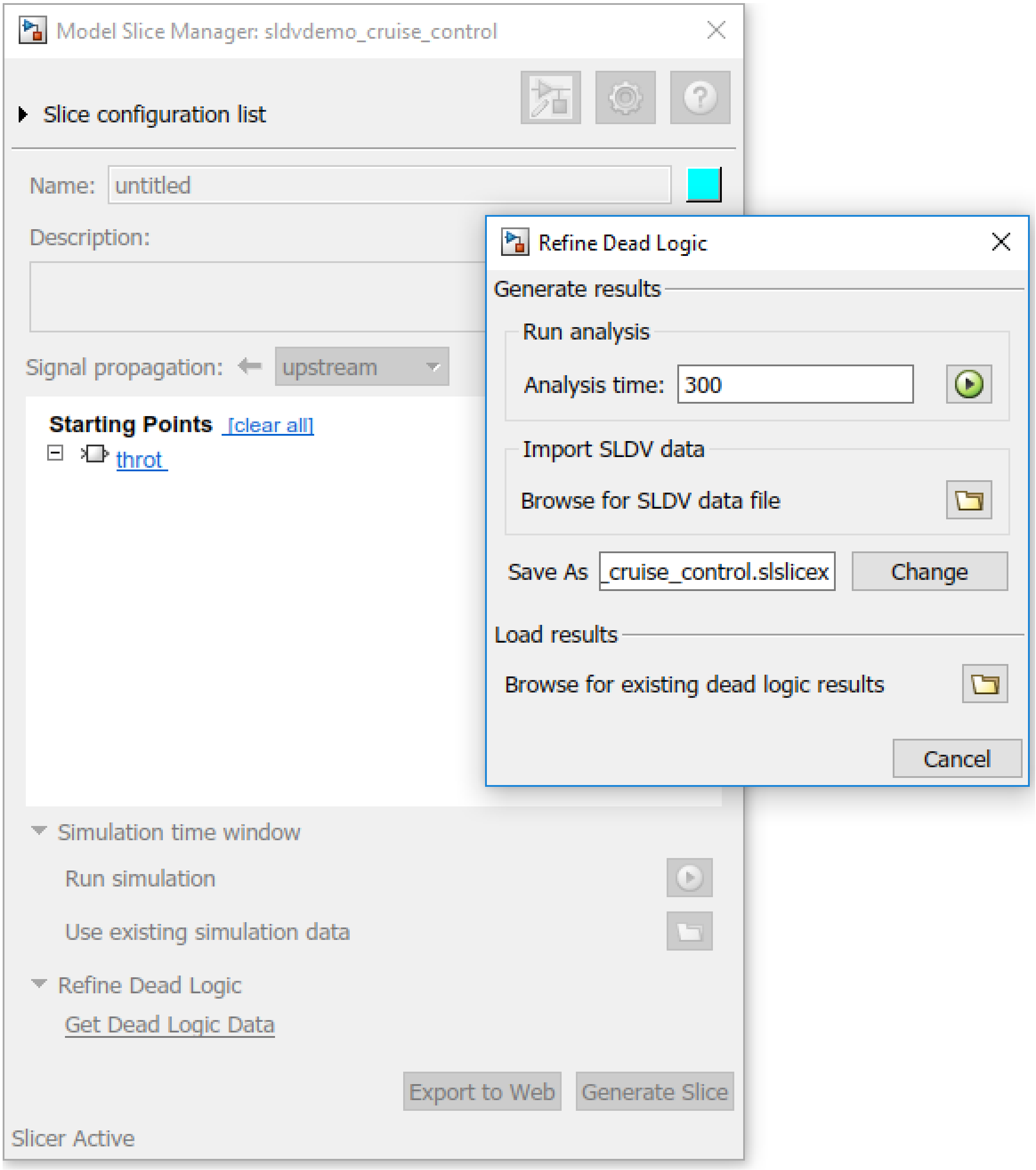
If you want to highlight functional dependencies in the model again at another time, you can use the existing`.slslicex` simulation time window data without needing to resimulate the model. Model Slicer then uses the existing simulation data to highlight the model.

1. Open the Simulink model.
2. To open the Model Slice Manager, select **Analysis** > **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**.



Model Slice Manager: sldvdemo_cruise_control

Slice configuration list

Name: untitled

Description:

Signal propagation: upstream

Starting Points [clear all]

throt

Simulation time window

Run simulation

Use existing simulation data

Refine Dead Logic

Get Dead Logic Data

Slicer Active

Refine Dead Logic

Generate results

Run analysis

Analysis time: 300

Import SLDV data

Browse for SLDV data file

Save As _cruise_control.slslicex

Load results

Browse for existing dead logic results

See Also

More About

- “Highlight Functional Dependencies” on page 11-2
- “Configure Model Highlight and Sliced Models” on page 11-49
- “Refine Dead Logic for Dependency Analysis” on page 11-26

Simplification of Variant Systems

In this section...

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 11-2 in a model containing variant systems, only active variant choices are highlighted. When you “Create a Simplified Standalone Model” on page 11-33 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

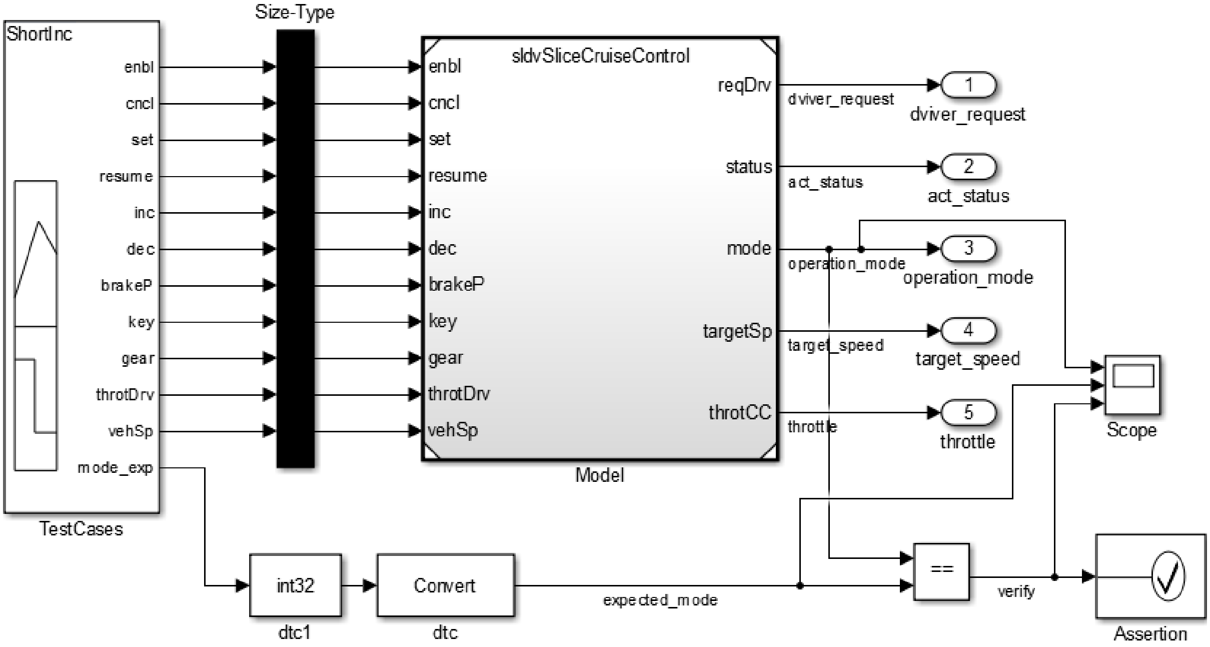
Be familiar with the behavior and purpose of Model Slicer and the functionality of the Model Slicer API. “Highlight Functional Dependencies” on page 11-2 outlines how to use Model Slicer user interface to explore models. The `slslicer`, `slsliceroptions`, and `slslicertrace` function reference pages 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.

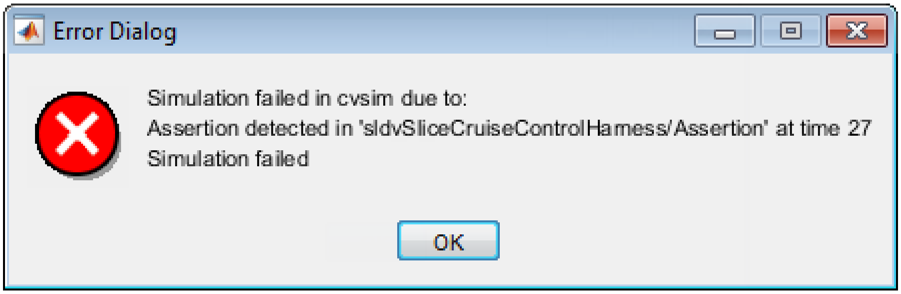
   ```
   open_system('sldvSliceCruiseControlHarness')
   ```



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 ▶all to run all of the included test cases. You receive an error during the ResumeWO test case.



Simulation failed in cvsim due to:
Assertion detected in 'sldvSliceCruiseControlHarness/Assertion' at time 27
Simulation failed

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**.

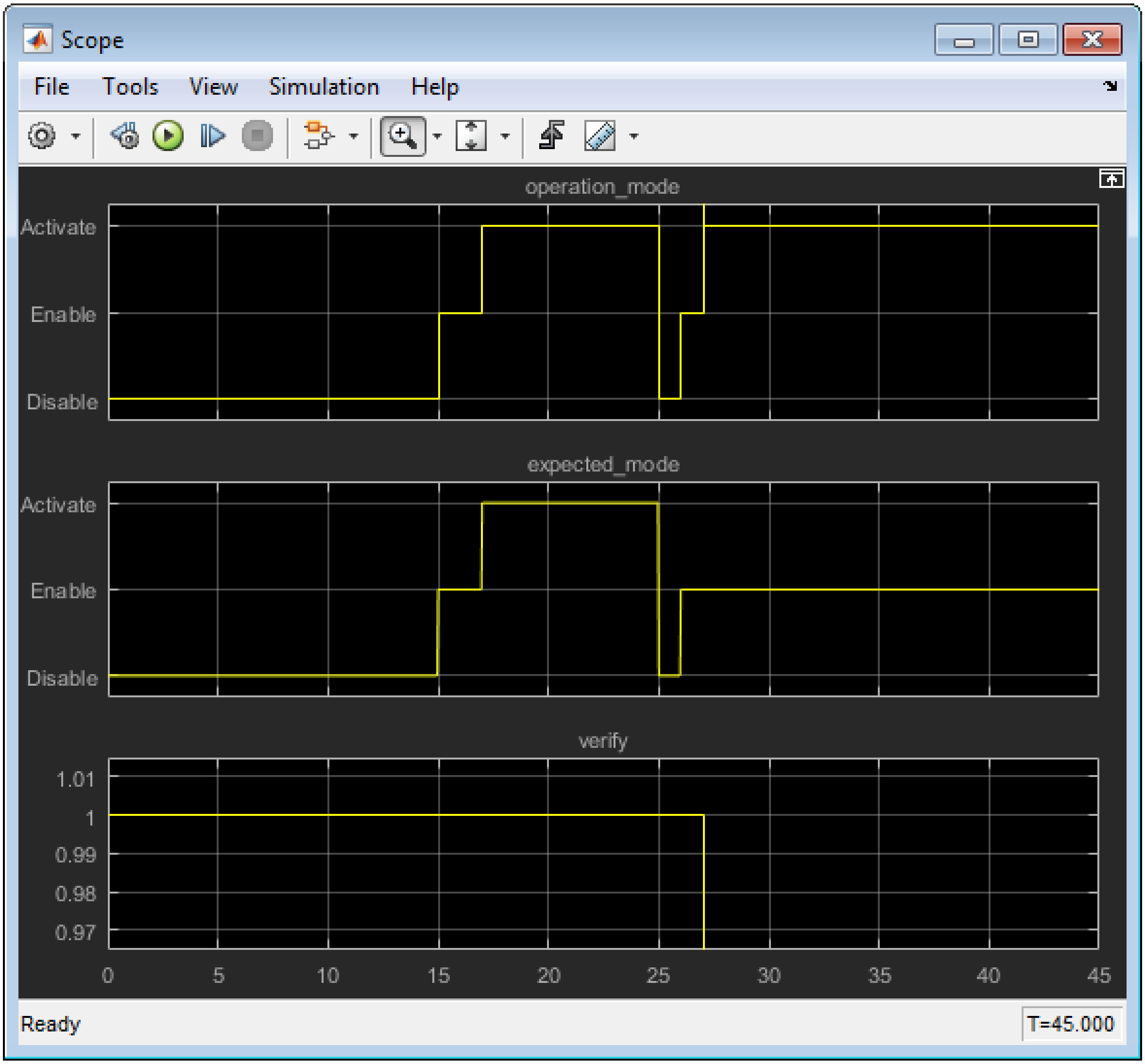
```
set_param('sldvSliceCruiseControlHarness/Assertion','Enabled','off')
```

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')
```

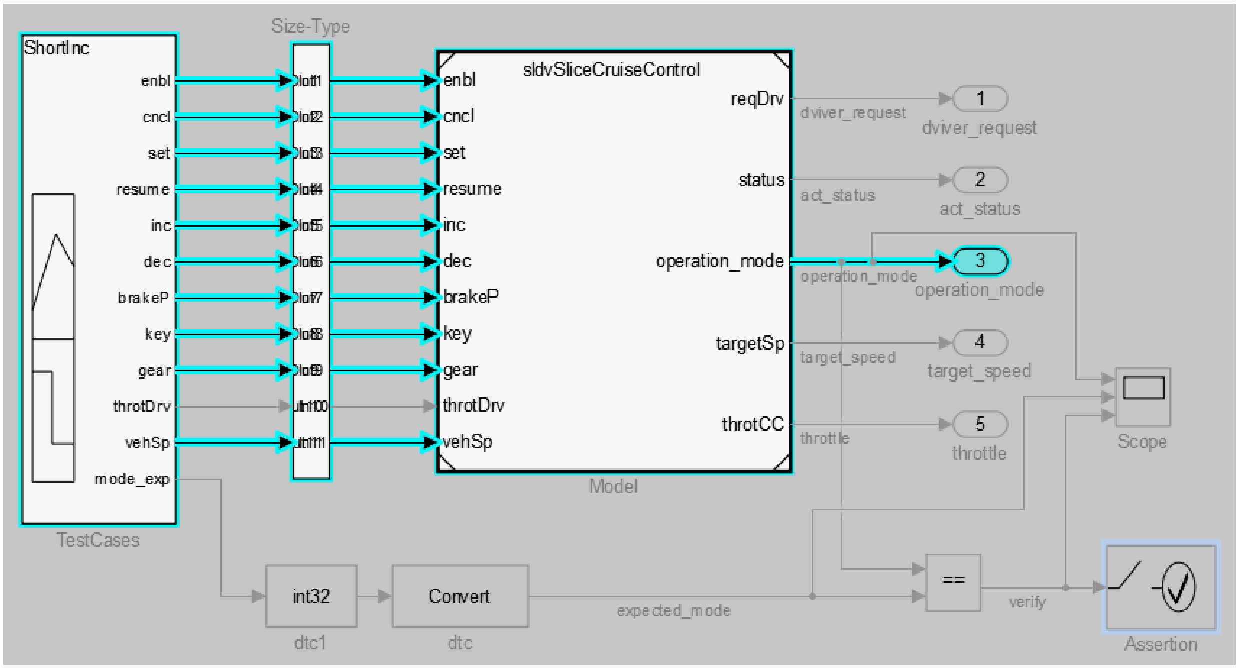
```
obj = 

  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)
```

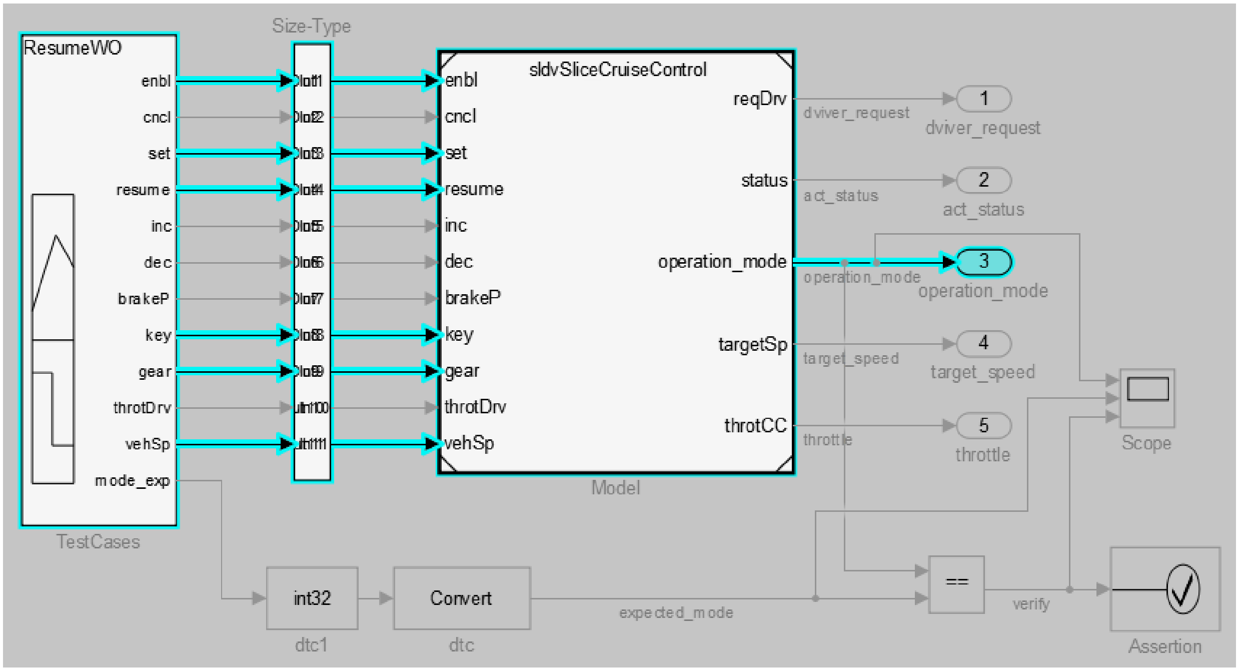


The area of the model upstream of the starting point and active during simulation is highlighted.

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.

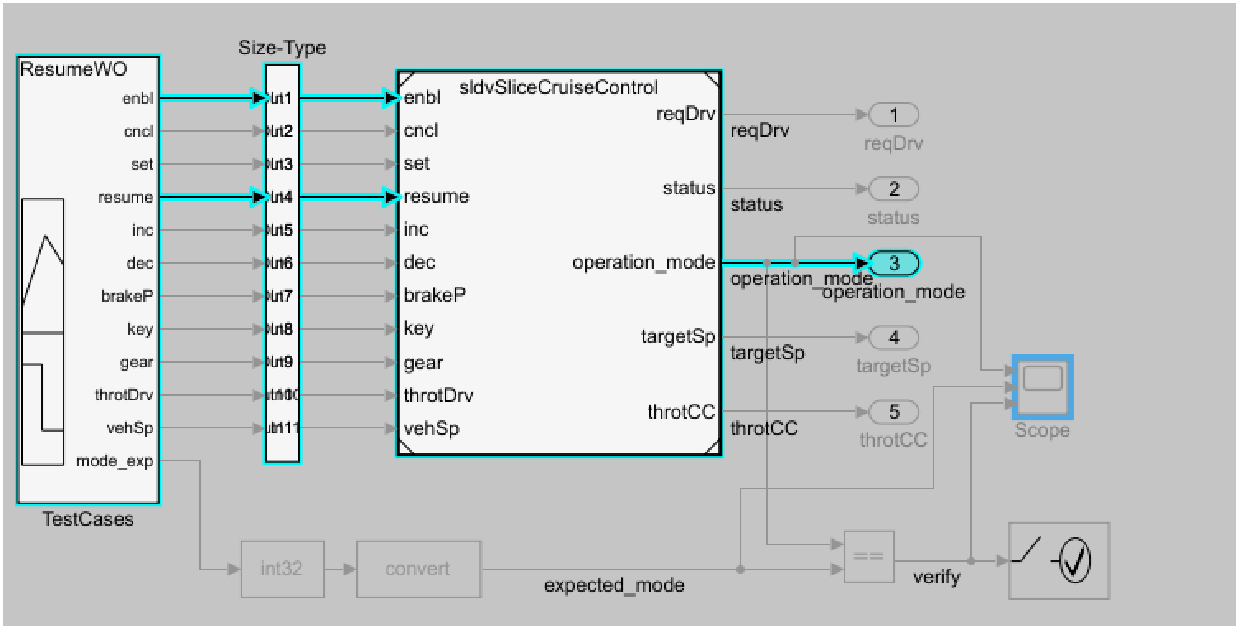
```
simulate(obj,0,30)
```

Only the portion of the model upstream of the starting point and active during the simulation time window is highlighted.



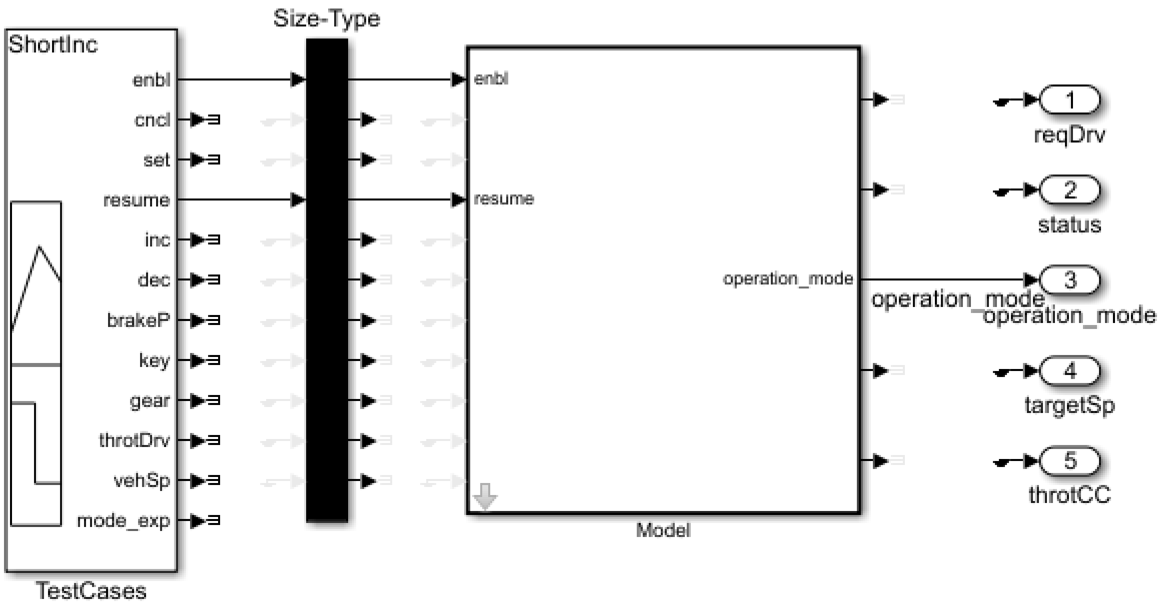
5 You can further narrow the simulation time window by changing the start time to 20 seconds.

```
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

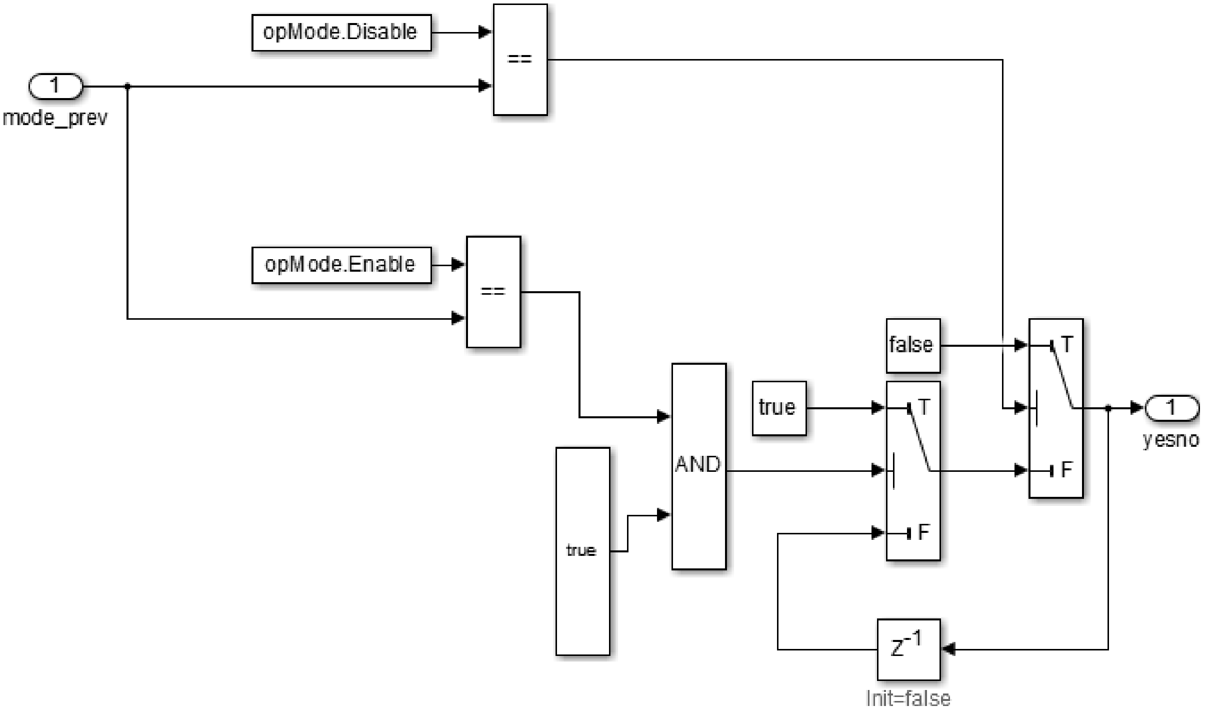
You can now debug the unexpected behavior in the simplified standalone model and then apply changes to the source model.

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.

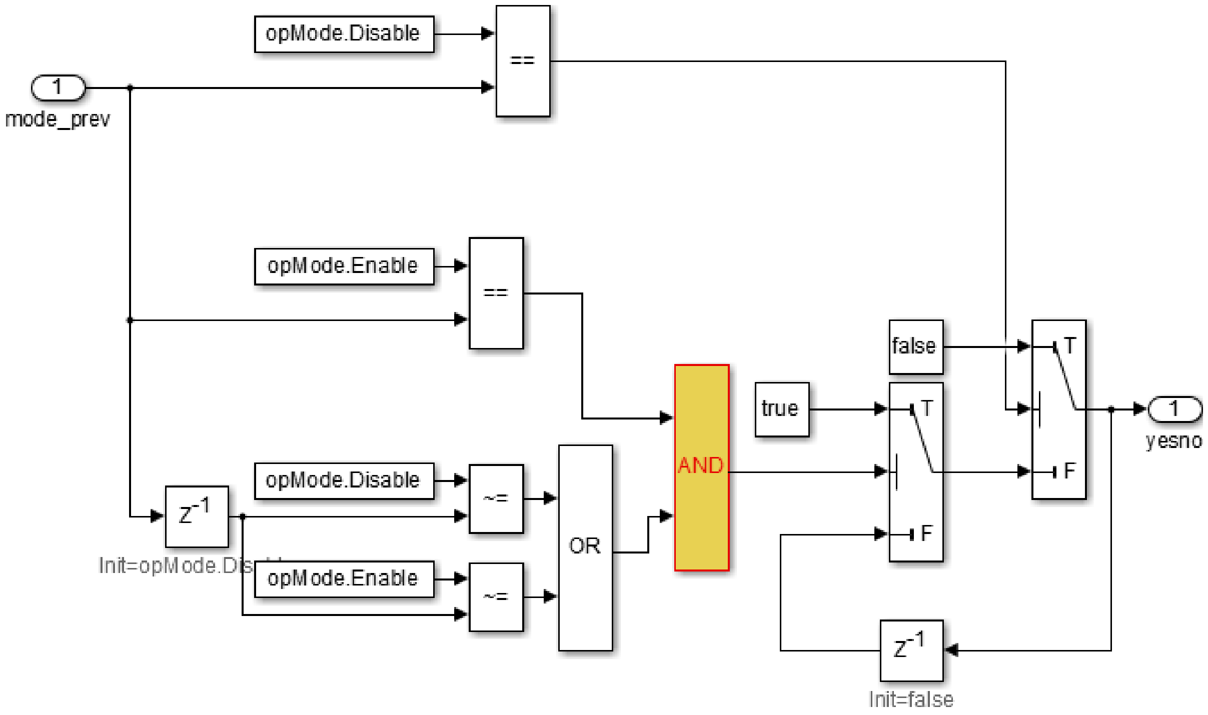
```
open_system('sldvSliceCruiseControlHarness_sliced/Model/CruiseControlMode/opMode/resumeCondition/hasCanc
```



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/LogicC
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.

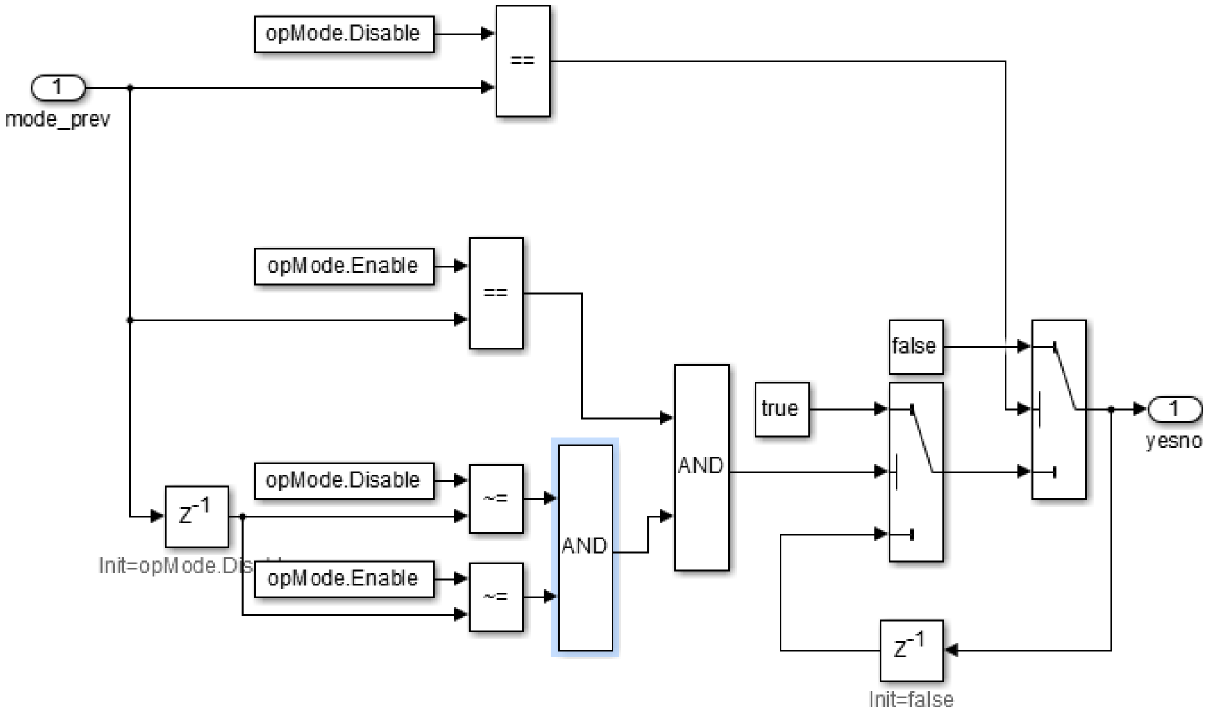
```
save_system('sldvSliceCruiseControl','sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness','sldvSliceCruiseControlHarness_fixed')
```

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.

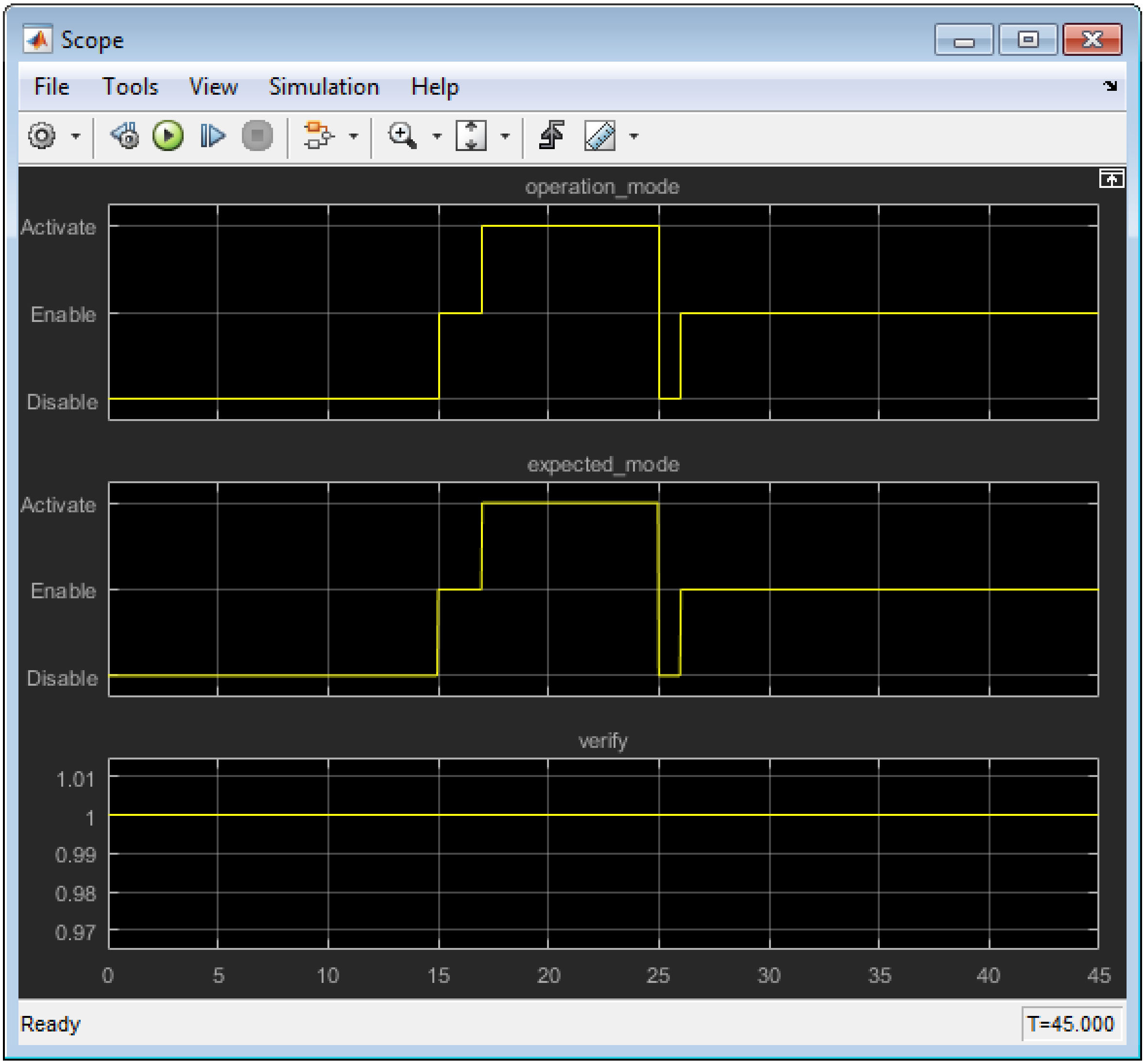
```
set_param('sldvSliceCruiseControl_fixed/CruiseControlMode/opMode/resumeCondition/hasCanceled/LogicOp2',
'LogicOp','AND')
```



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

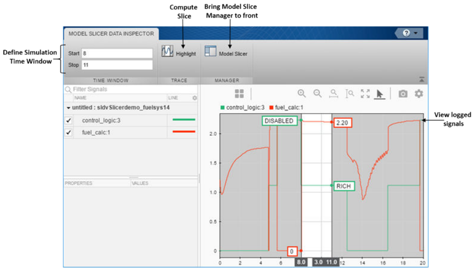
- “Highlight Functional Dependencies” on page 11-2

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 11-2.



Start 8

Stop 11

Investigate Highlighted Model Slice by Using Model Slicer Data Inspector

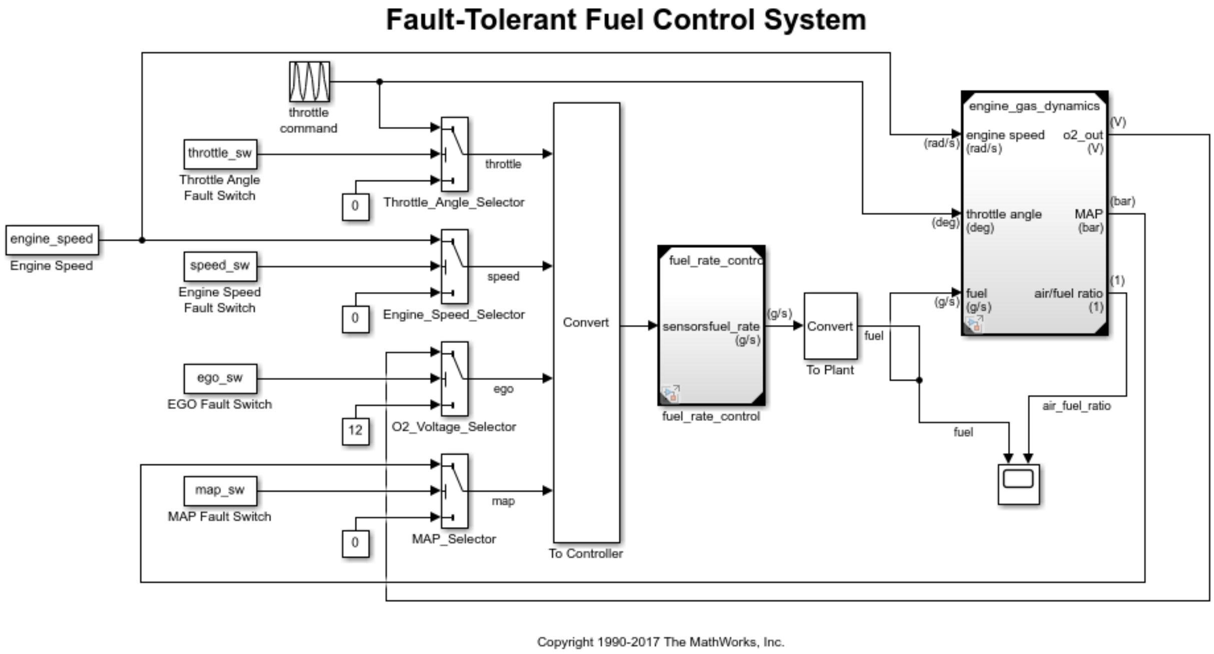
This example shows how to investigate and refine the highlighted model slice by using the Model Slicer Data Inspector.

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" (Stateflow).

Step 1: Start the Model Slice Manager

To start the Model Slice Manager, open the `fuel_rate_control` model, and select **Analysis > 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**.

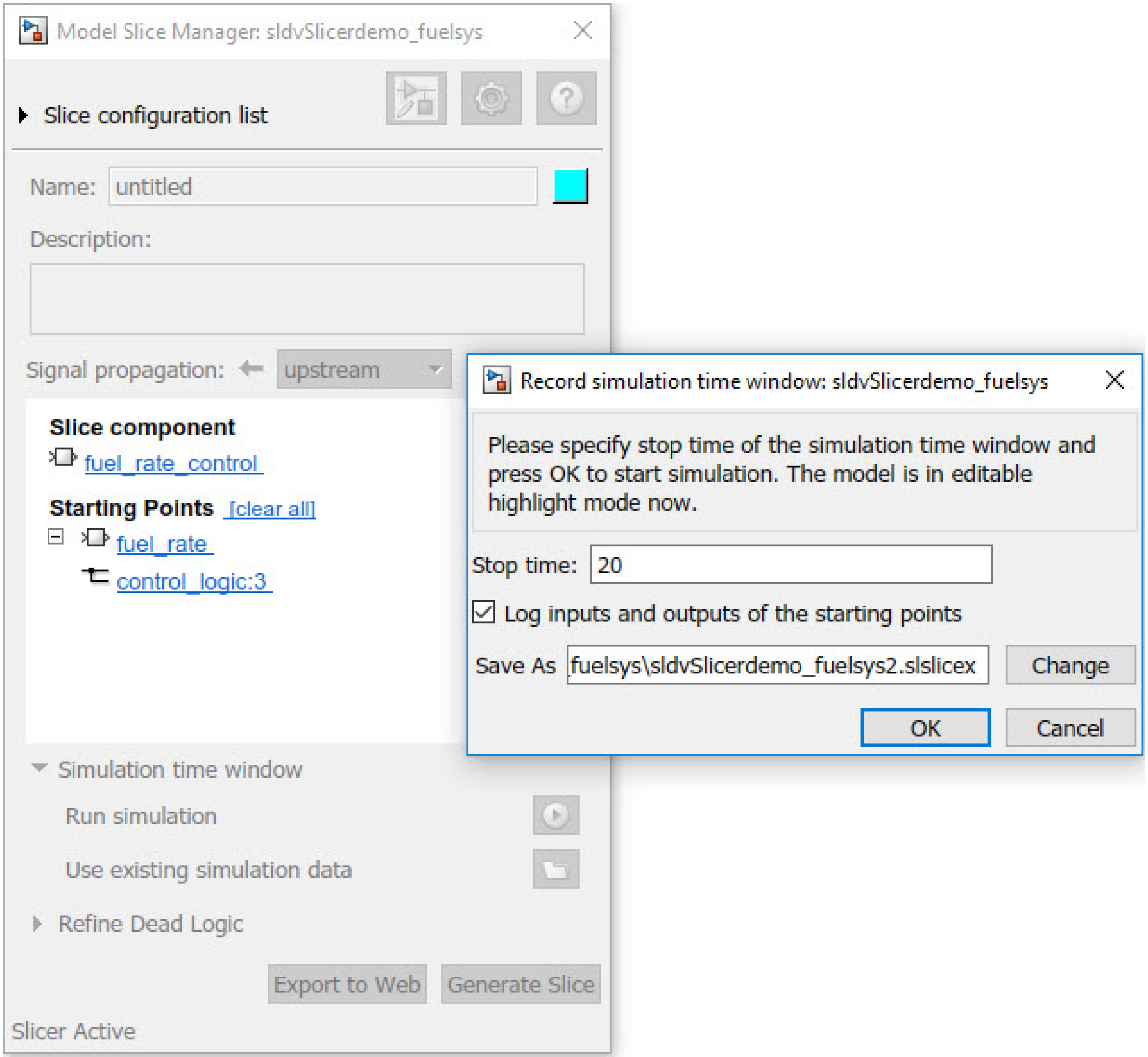
Step 2: Log input and output signals

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**.



Name: untitled

Description:

Signal propagation: upstream

Slice component

fuel_rate_control

Starting Points [clear all]

fuel_rate

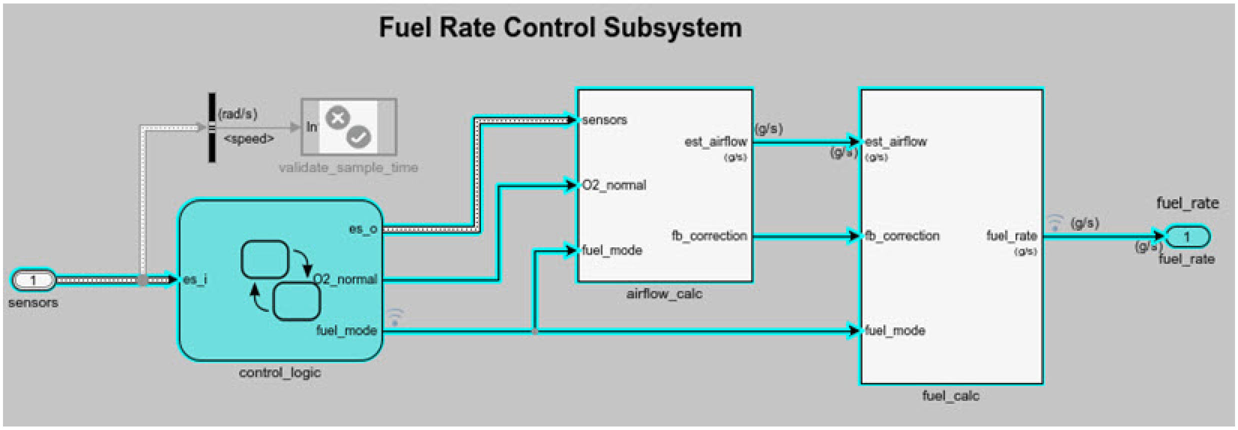
control_logic:3

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

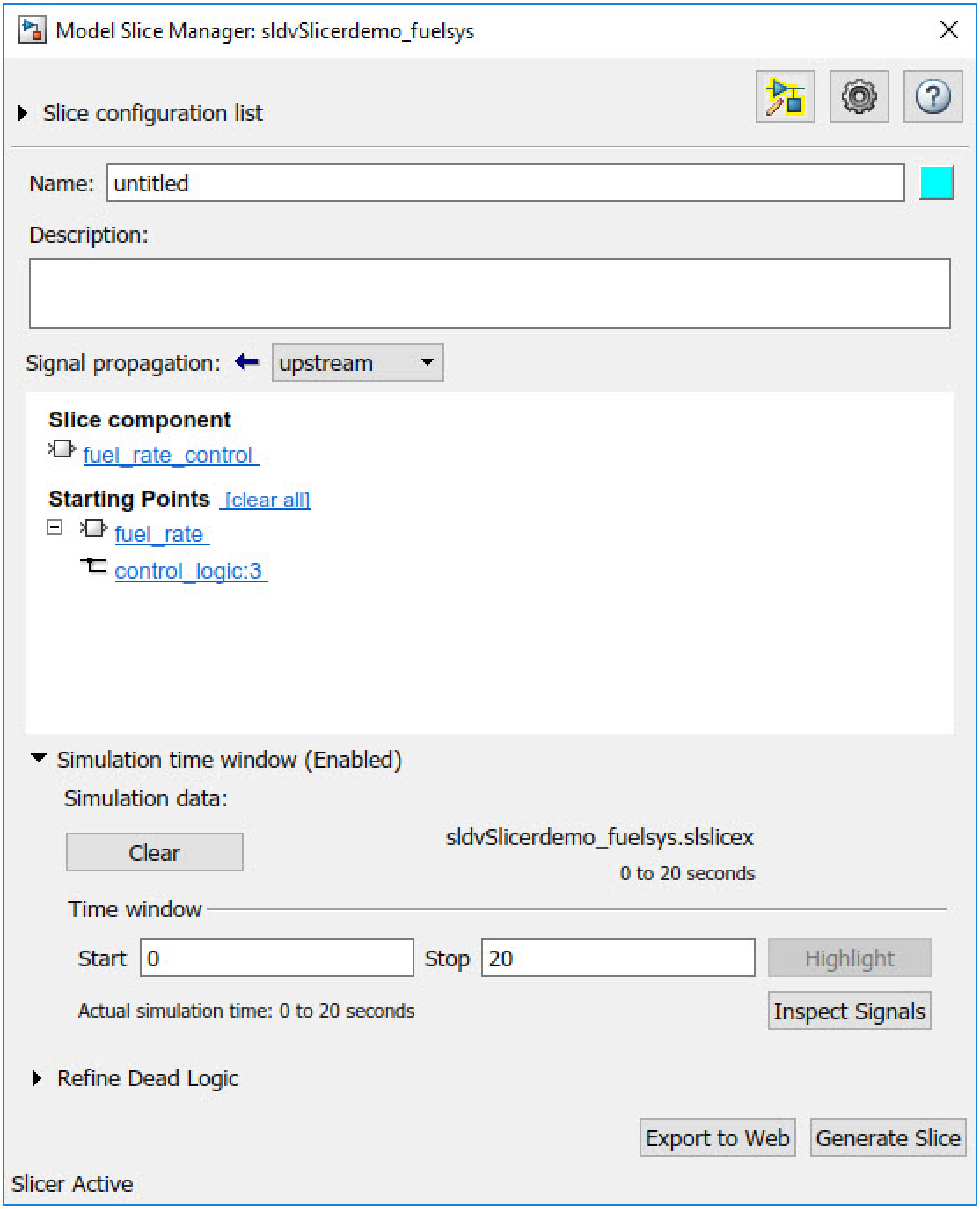
☑ Log inputs and outputs of the starting points

Save As fuelsys\sldvSlicerdemo_fuelsys2.slslicex



Step 3: Inspect signals

To open the Model Slicer Data Inspector, click **Inspect Signals**.



Model Slice Manager: sldvSlicerdemo_fuelsys

Name: untitled

Description:

Signal propagation: ← upstream

Slice component

fuel_rate_control

Starting Points [clear all]

fuel_rate

control_logic:3

Simulation time window (Enabled)

Simulation data: sldvSlicerdemo_fuelsys.slslicex

0 to 20 seconds

Time window

Start 0 Stop 20

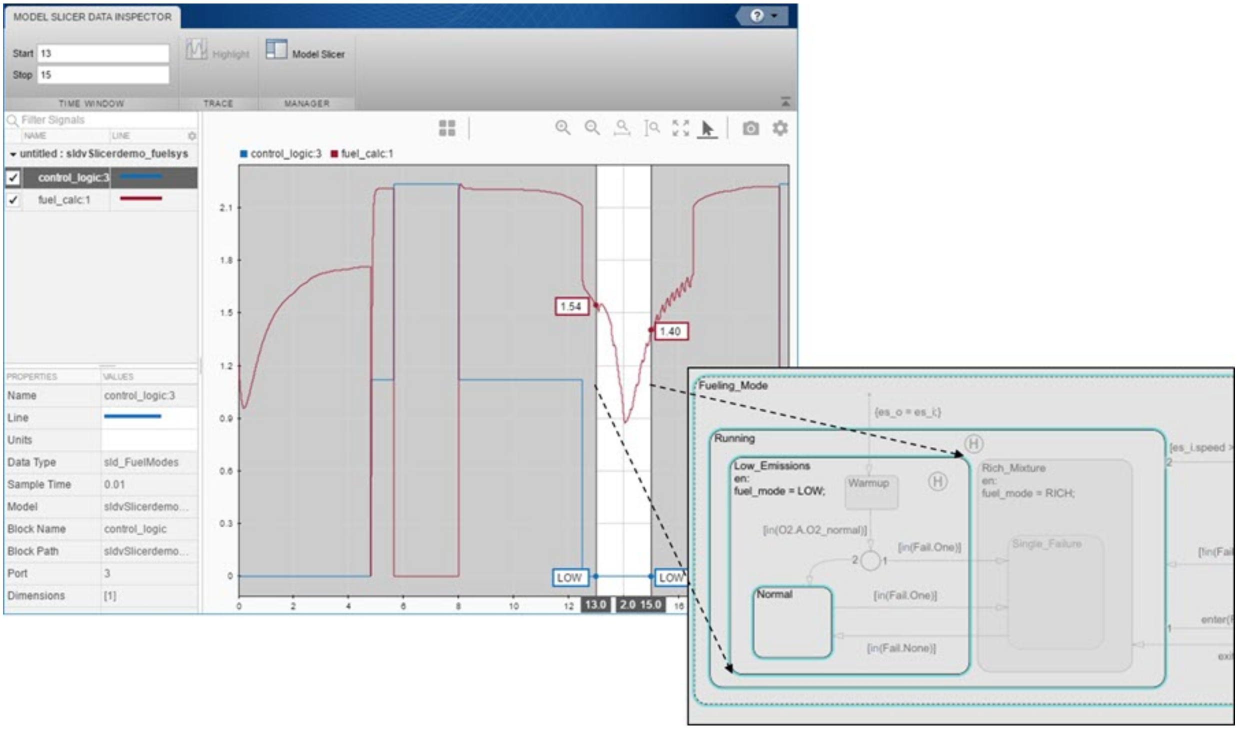
Actual simulation time: 0 to 20 seconds

Slicer Active

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.

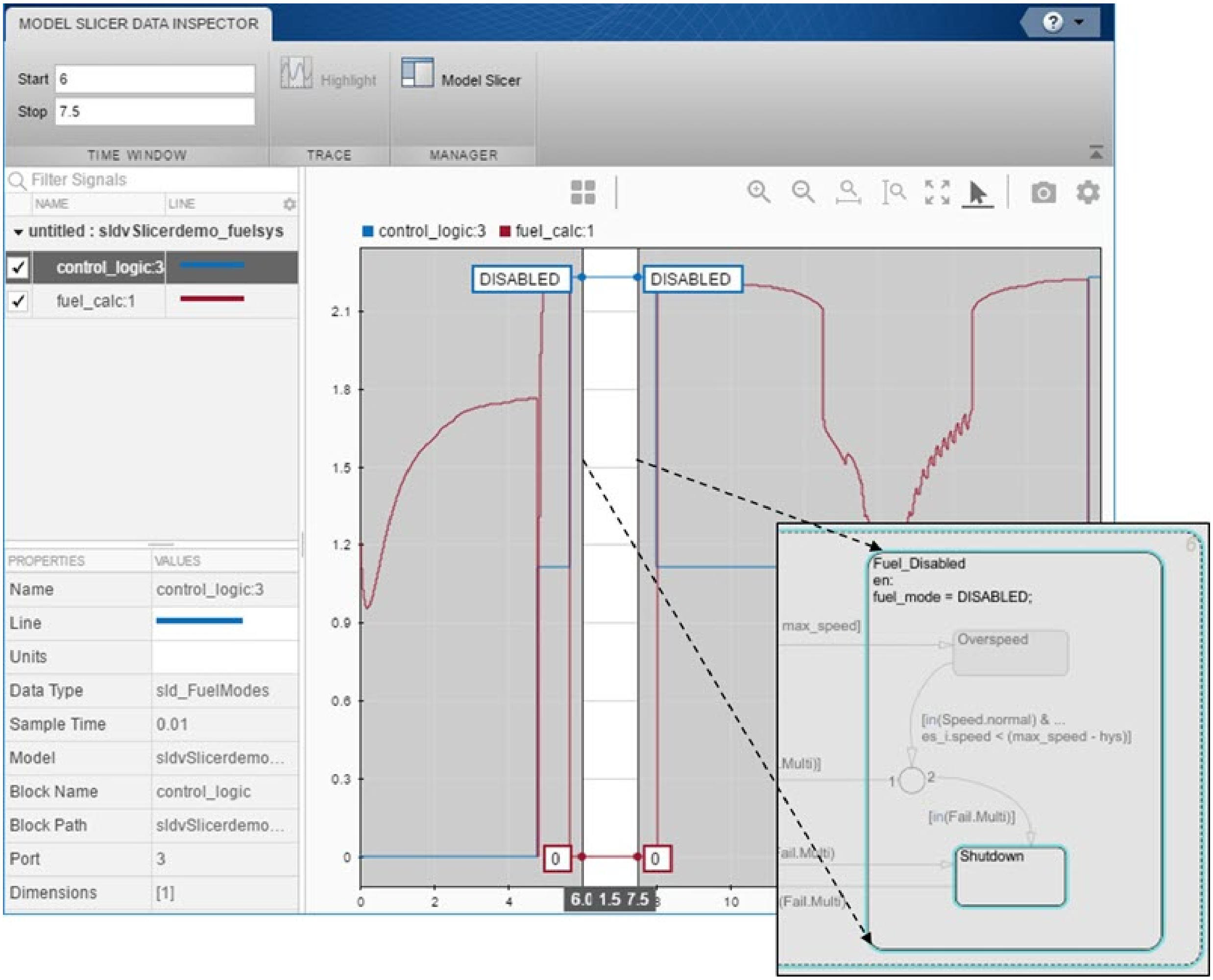


Start 13

Stop 15

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]

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.



Start 6

Stop 7.5

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]

See Also

“Highlight Functional Dependencies” on page 11-2 | “Refine Highlighted Model” on page 11-13

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.

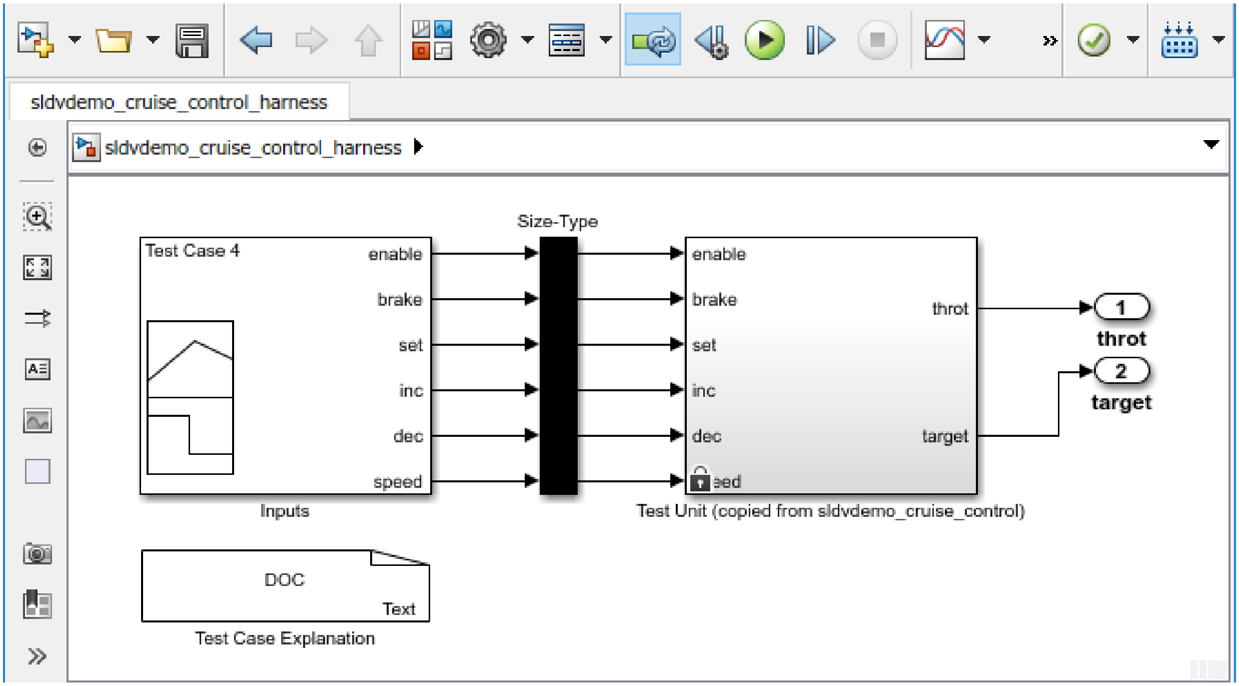
1. Open the `sldvdemo_cruise_control` model.

   ```
   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);
   ```

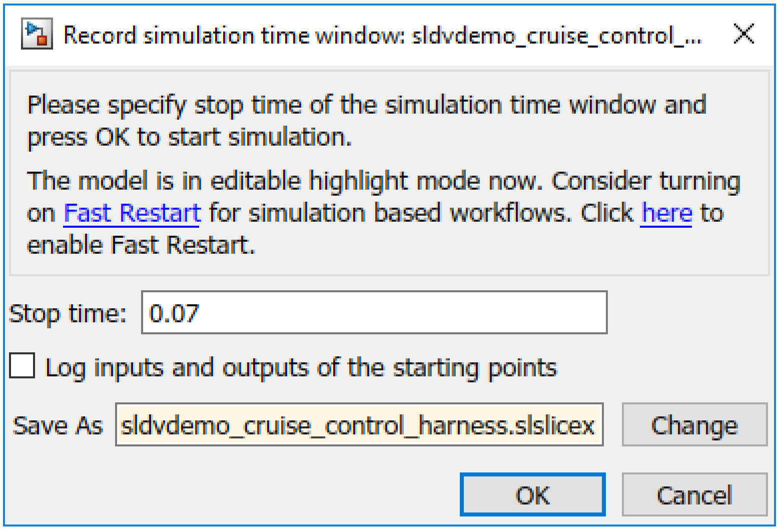
   After the analysis, the software opens a harness model `sldvdemo_cruise_control_harness` and saves it in the default location `<current_folder>\sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_harness.slx`. For more information, see “Simulink Design Verifier Harness Models” (Simulink Design Verifier).



3 To enable the fast restart mode, click **Enable Fast Restart** button .

4 To open the Model Slice Manager, click **Analysis > 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.



Record simulation time window: sldvdemo_cruise_control_...

Please specify stop time of the simulation time window and press OK to start simulation.

The model is in editable highlight mode now. Consider turning on Fast Restart for simulation based workflows. Click here to enable Fast Restart.

Stop time: 0.07

□ Log inputs and outputs of the starting points

Save As sldvdemo_cruise_control_harness.slslicex

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**.

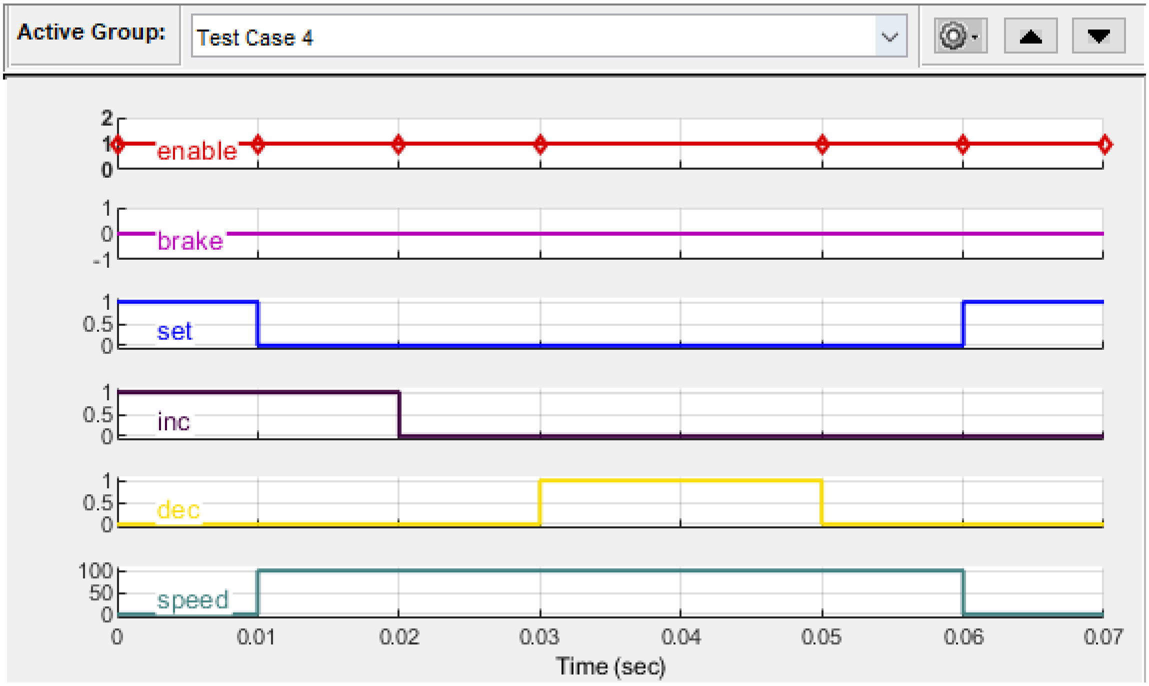
6 You can simulate a test case and analyze the highlighted dependencies in the slice.

a In the Signal Builder block, select `Test Case 4`.

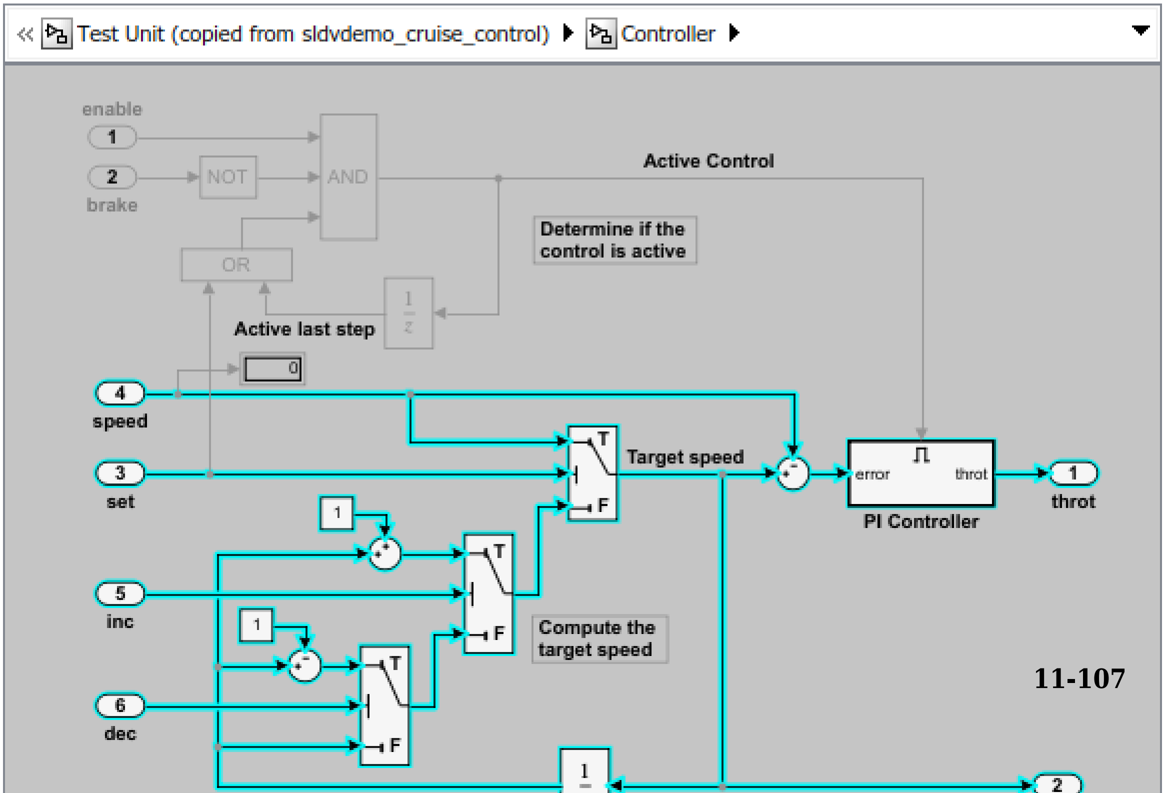
b To simulate the test case, click **Start simulation** button, ▶.

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.

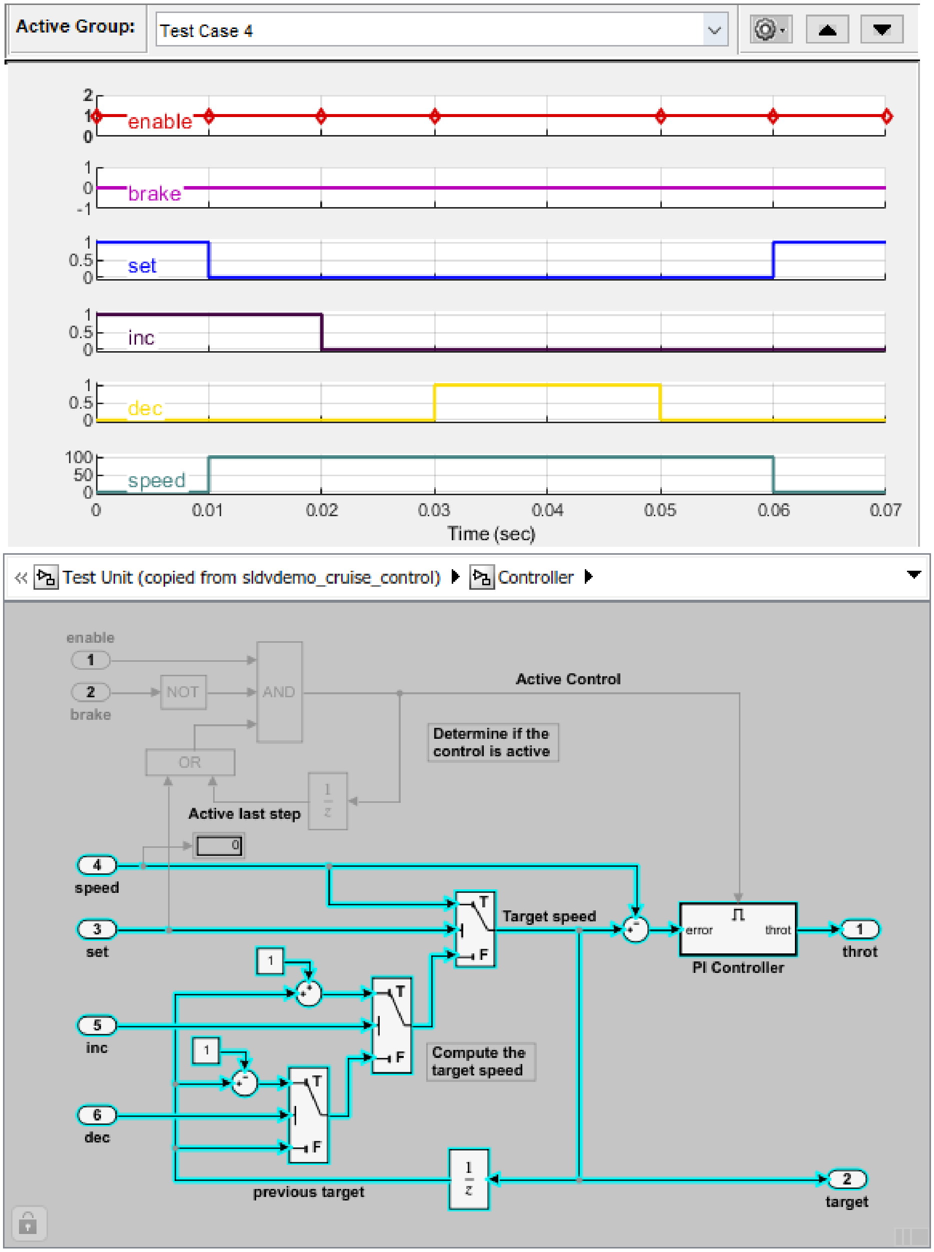
The slice shows the highlighted dependencies for the `Test Case 4` inputs.



Active Group: Test Case 4

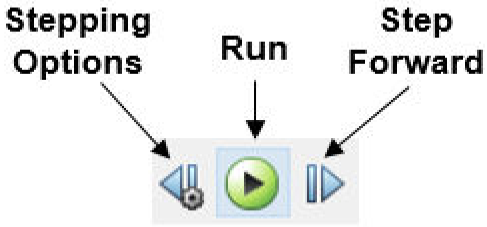


Test Unit (copied from sldvdemo_cruise_control) ▸ Controller ▸

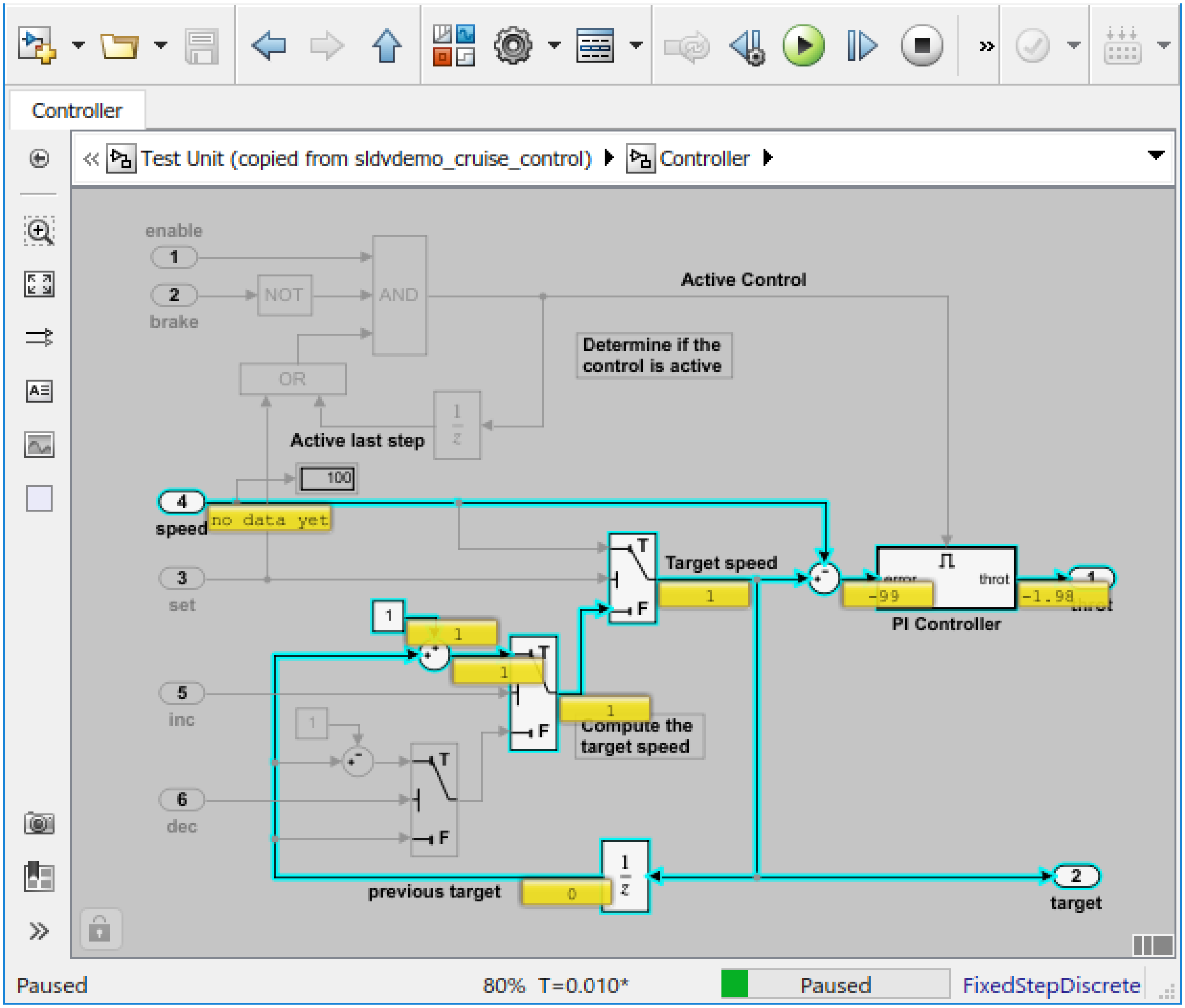

Active Group:
Test Case 4
enable
brake
set
inc
dec
speed
2
1
0
-1
0.5
100
50
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
Time (sec)
Test Unit (copied from sldvdemo_cruise_control)
Controller
enable
1
2
brake
NOT
AND
OR
Active Control
Determine if the control is active
Active last step
1/z
0
4
speed
3
set
5
inc
6
dec
1
T
F
Target speed
error
throt
PI Controller
1
throt
Compute the target speed
previous target
2
target

You can simulate a slice for different test case inputs and analyze the dependency analysis.

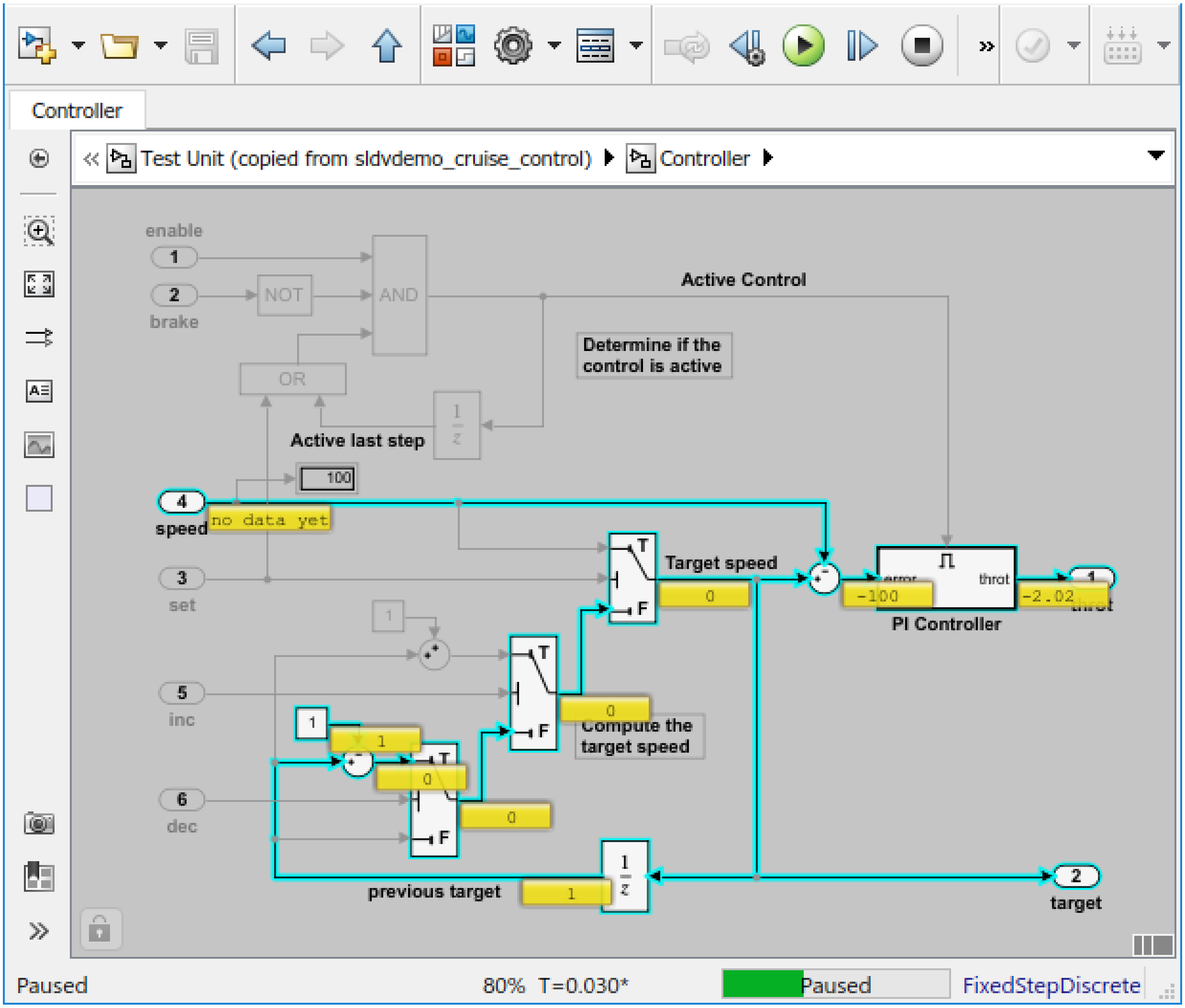
7 Debug a slicer simulation by using a simulation stepper. For more information see, “Simulation Stepper Access” (Simulink).



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 11-2
- “Simulation Stepper” (Simulink)
- “Get Started with Fast Restart” (Simulink)

Isolate Referenced Model for Functional Testing

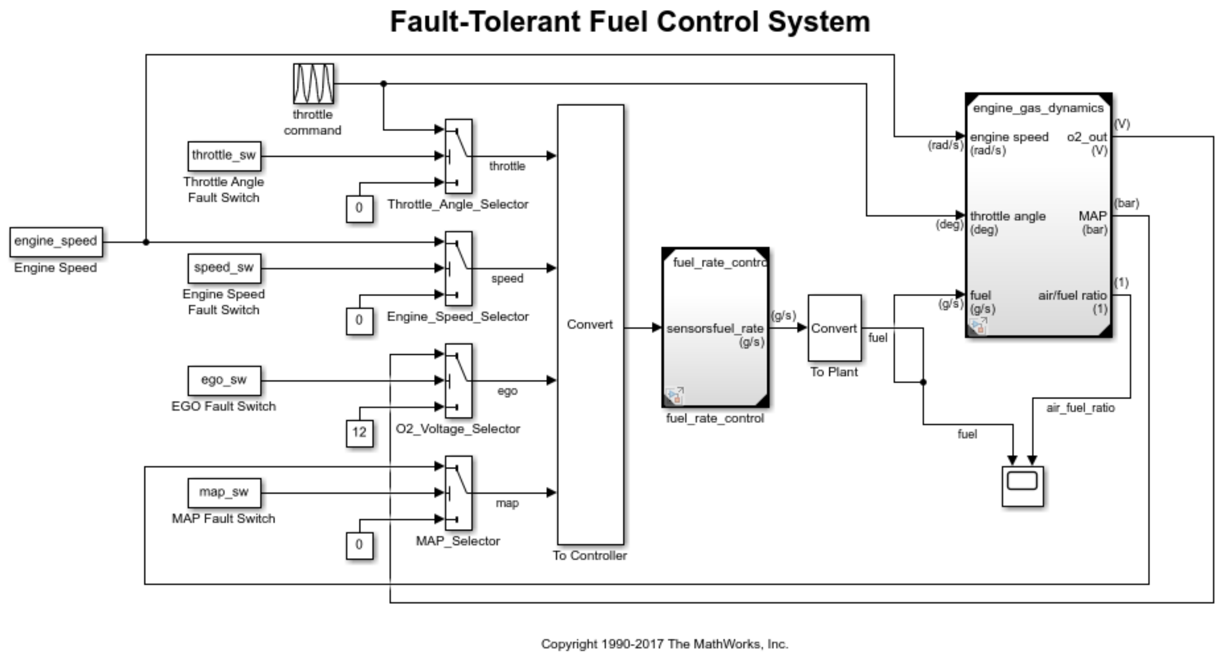
To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

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');
```



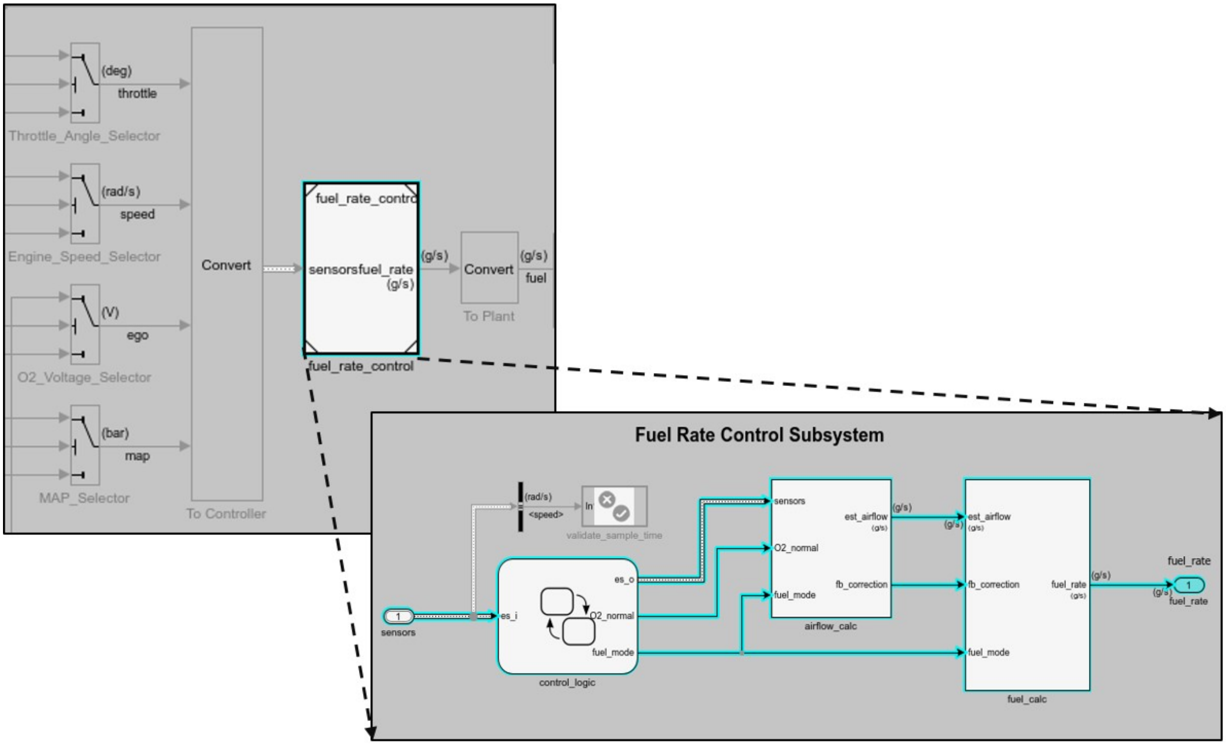
Step 2: Slice the Referenced Model

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 **Analysis** > **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`.



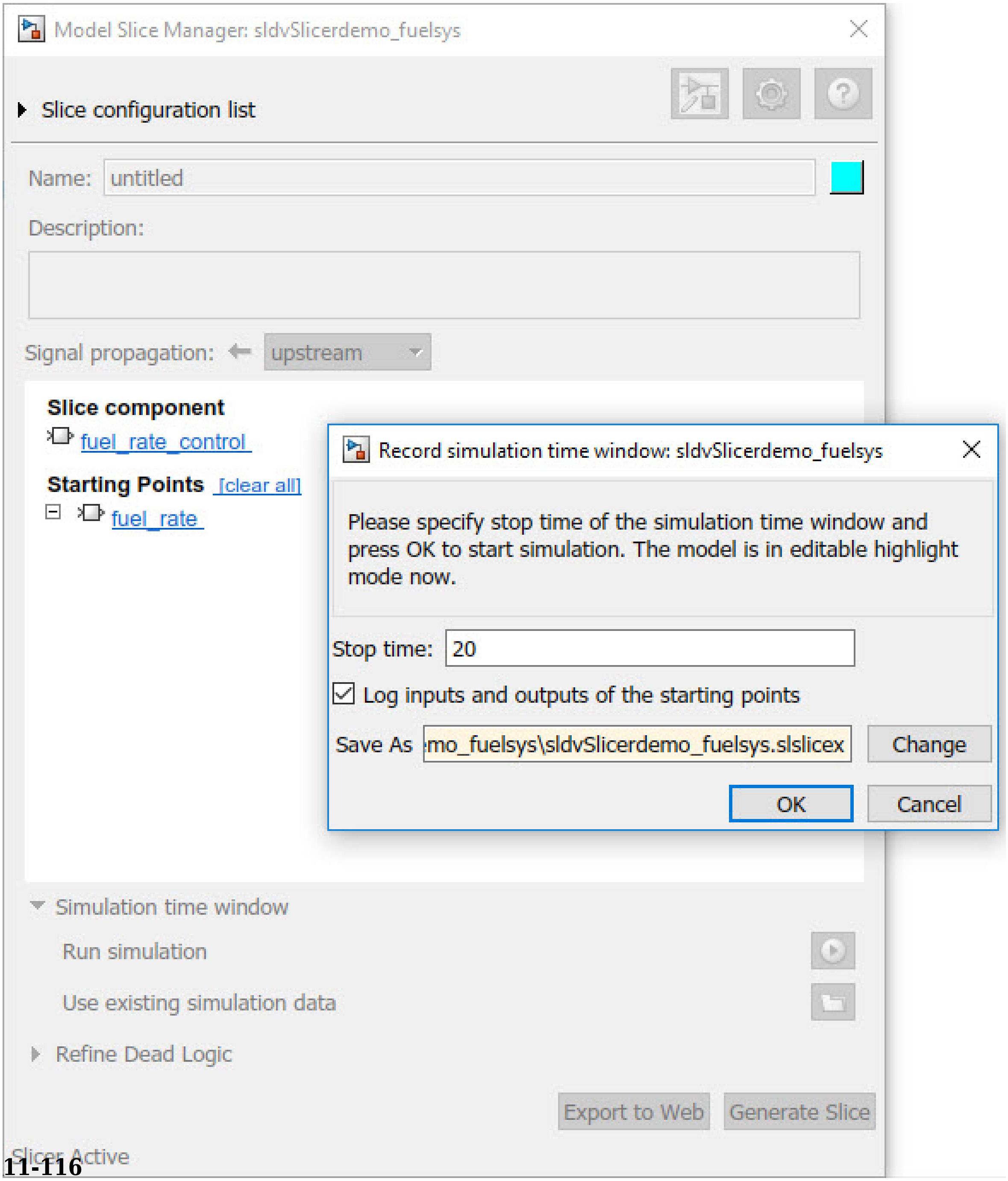
Step 4: Generate Slice

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.



Model Slice Manager: sldvSlicerdemo_fuelsys

Name: untitled

Description:

Signal propagation: upstream

Slice component

fuel_rate_control

Starting Points [clear all]

fuel_rate

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.

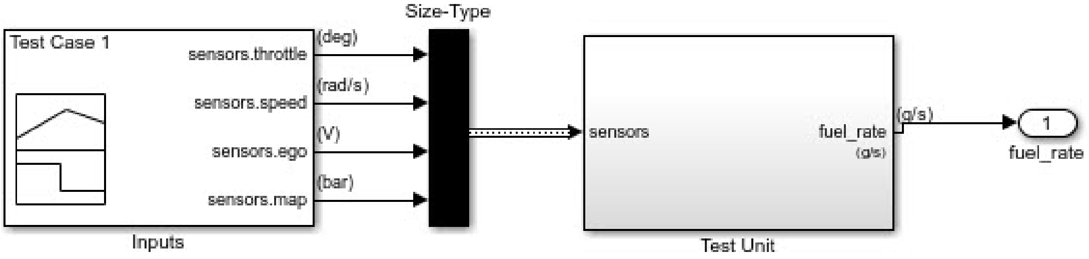
Stop time: 20

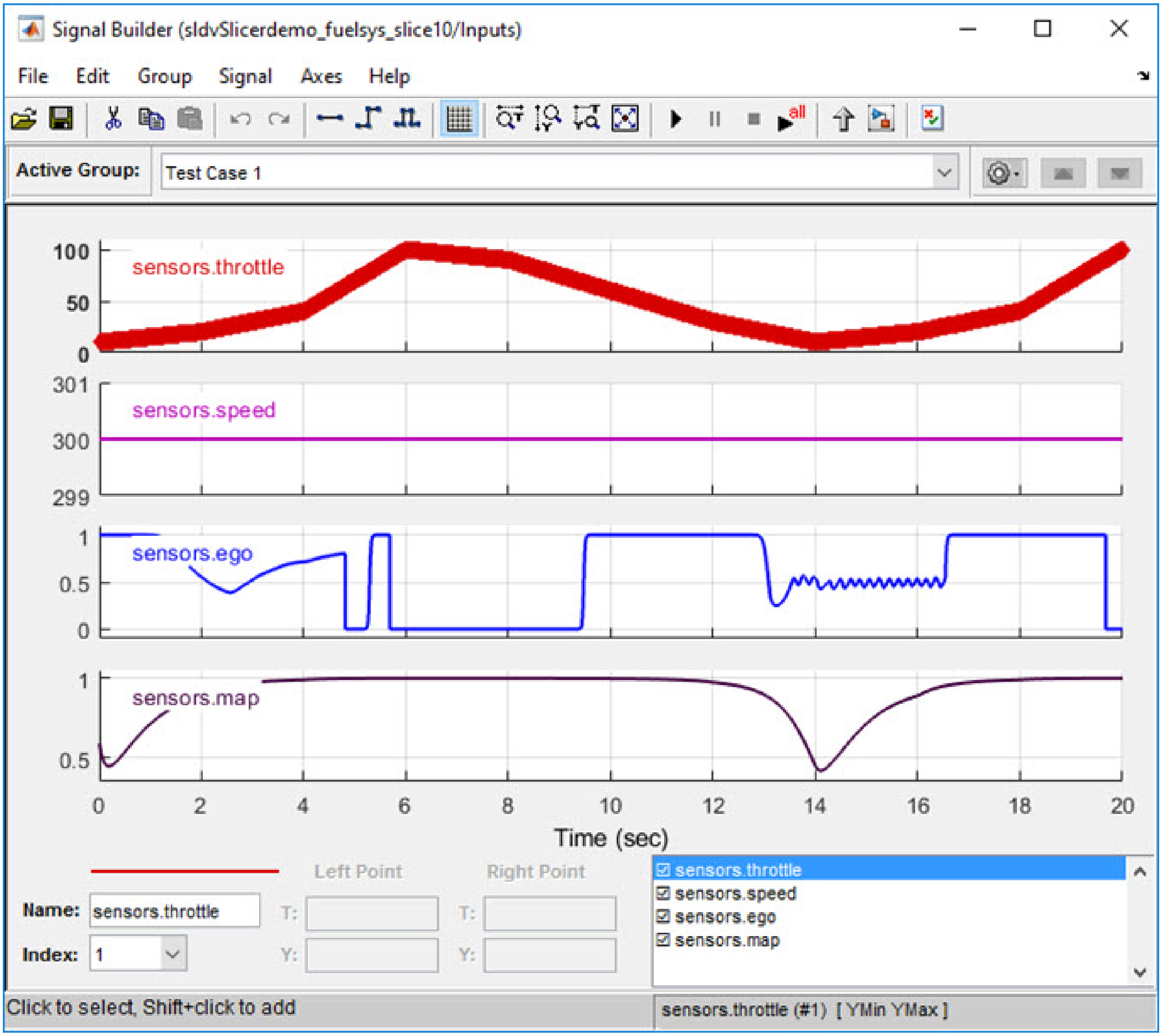
☑ Log inputs and outputs of the starting points

Save As mo_fuelsys\sldvSlicerdemo_fuelsys.slslicex

Slicer Active

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.





Signal Builder (sldvSlicerdemo_fuelsys_slice10/Inputs)

Active Group: Test Case 1

Name: sensors.throttle

Index: 1

☑ sensors.throttle
☑ sensors.speed
☑ sensors.ego
☑ sensors.map

Click to select, Shift+click to add

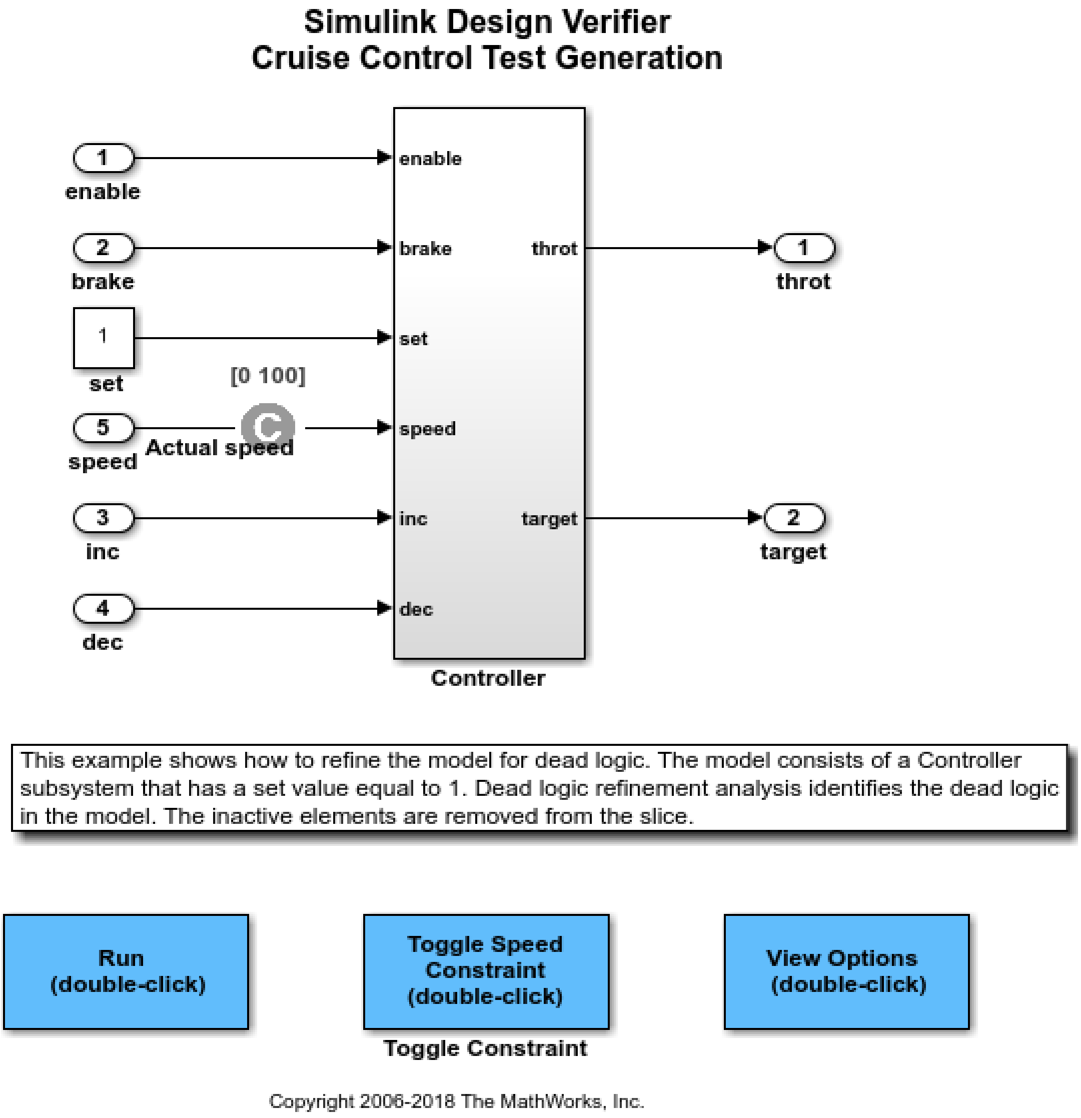
sensors.throttle (#1) [YMin YMax]

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.

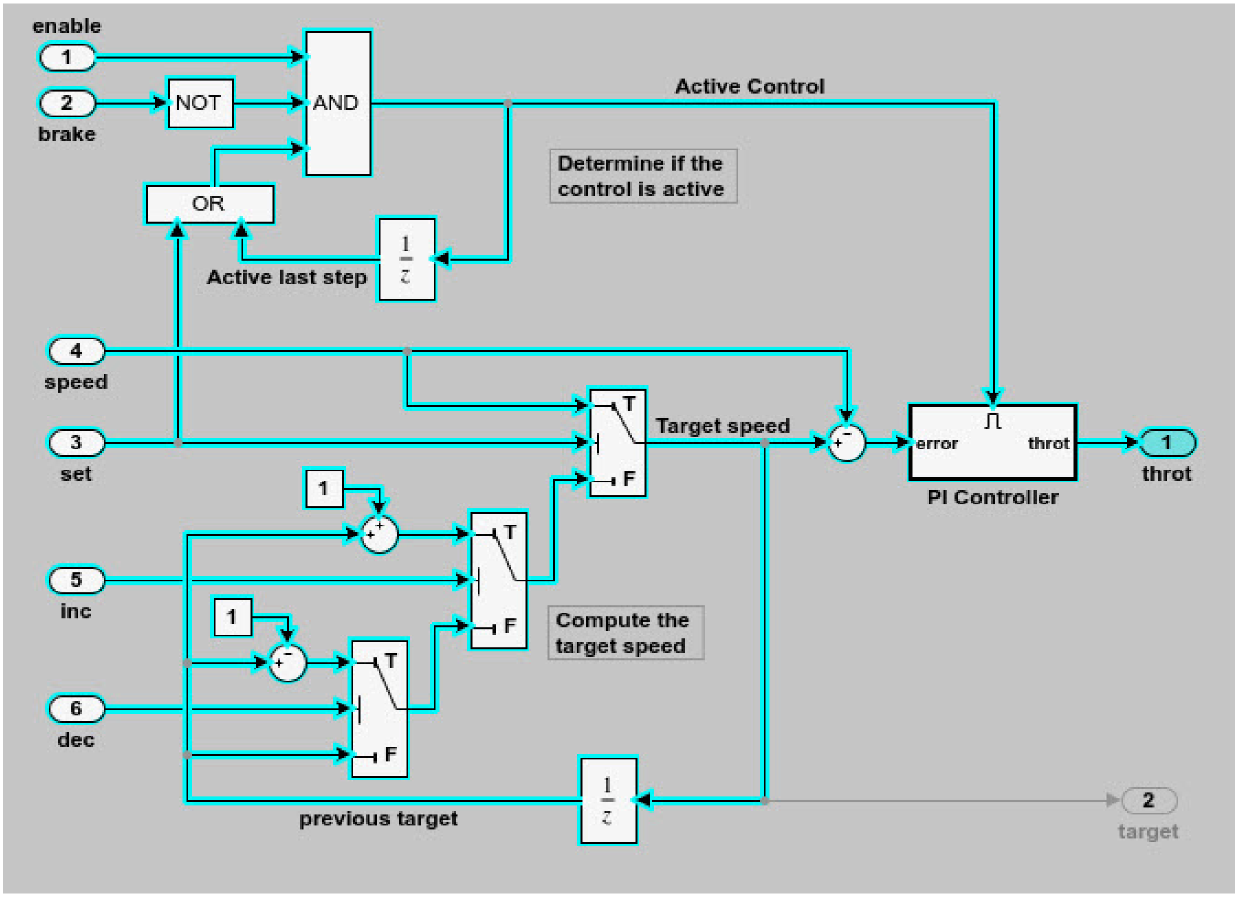
1. Open the `sldvSlicerdemo_dead_logic` model, and then select **Analysis** > **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```



This example shows how to refine the model for dead logic. The model consists of a Controller subsystem that has a set value equal to 1. Dead logic refinement analysis identifies the dead logic in the model. The inactive elements are removed from the slice.

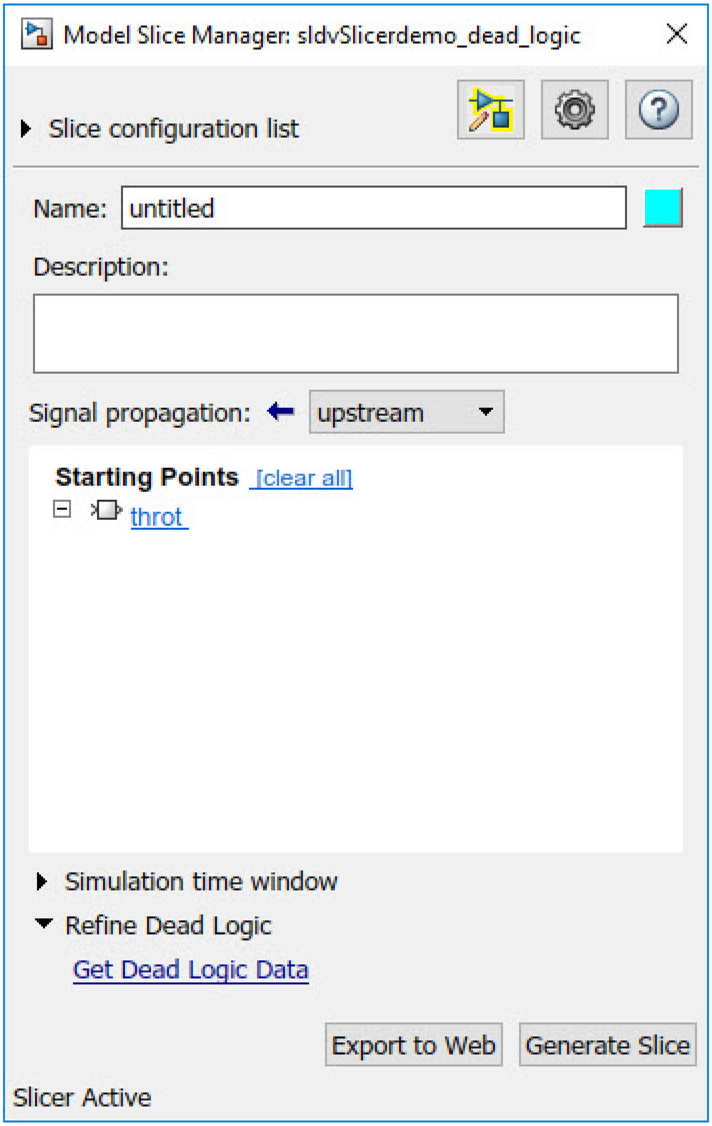
Open the `Controller` subsystem and add the outport `throt` as the starting point.



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**.



Model Slice Manager: sldvSlicerdemo_dead_logic

Slice configuration list

Name: untitled

Description:

Signal propagation: upstream

Starting Points [clear all]

throt

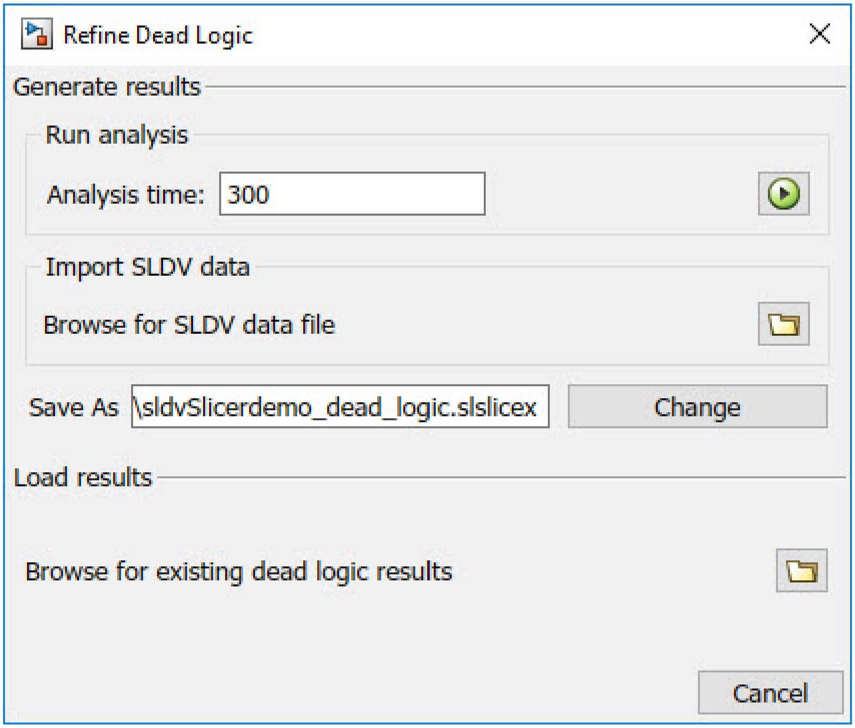
Simulation time window

Refine Dead Logic

Get Dead Logic Data

Slicer Active

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 11-78.



Refine Dead Logic

Generate results

Run analysis

Analysis time: 300

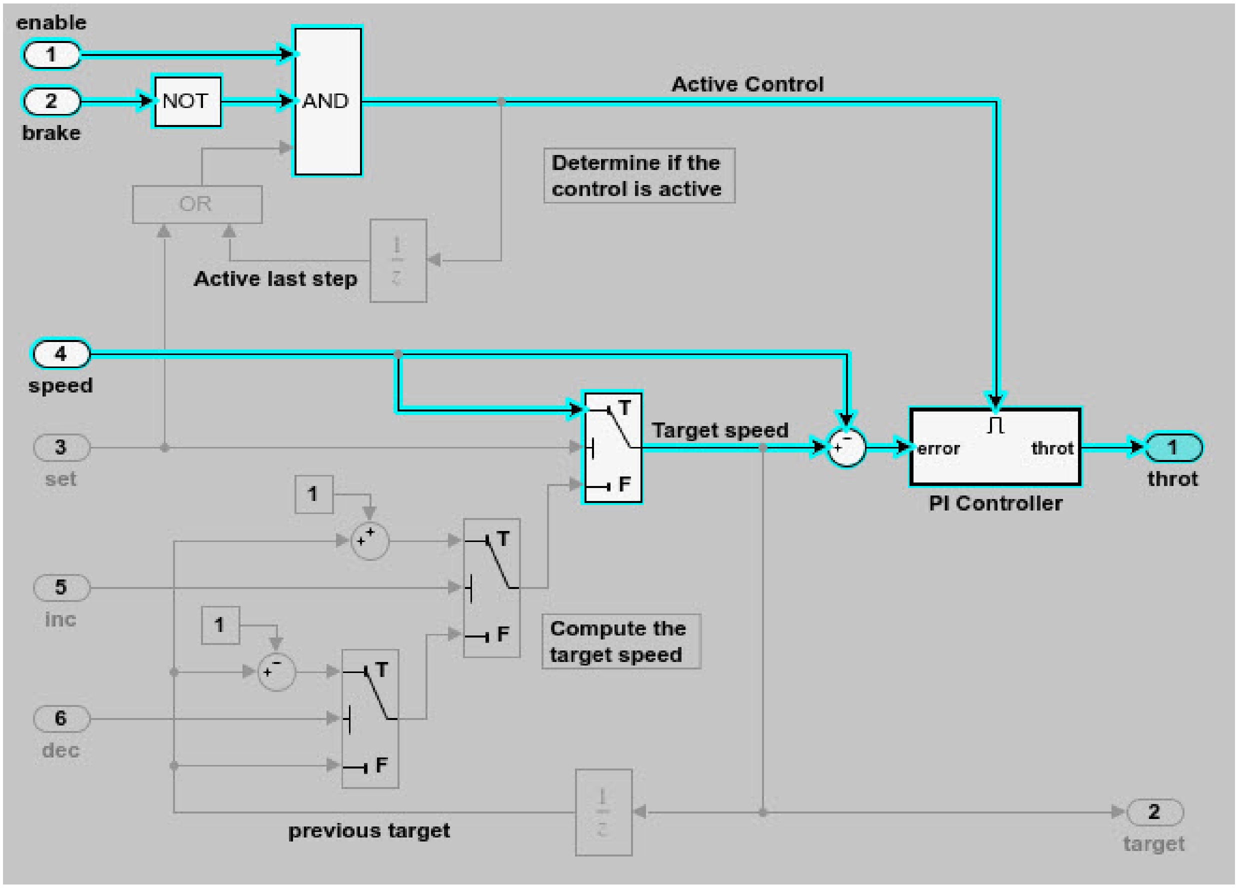
Import SLDV data

Browse for SLDV data file

Save As \sldvSlicerdemo_dead_logic.slslicex

Load results

Browse for existing dead logic results



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

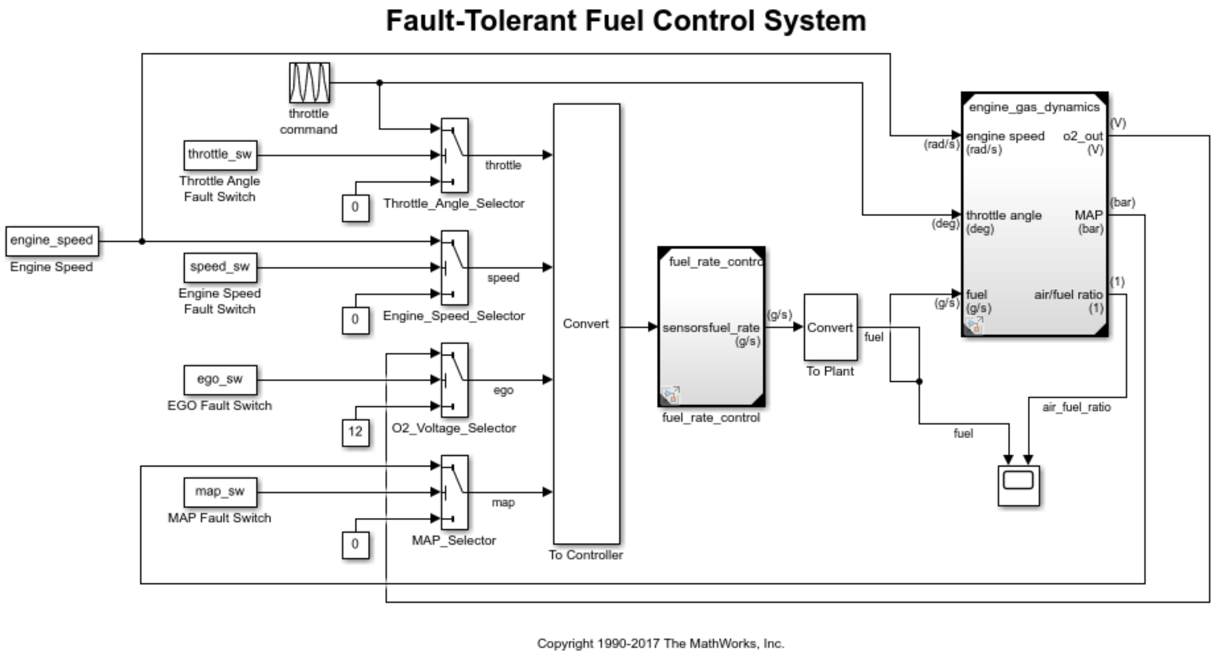
This example shows how to investigate and refine the highlighted model slice by using the Model Slicer Data Inspector.

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” (Stateflow).

Step 1: Start the Model Slice Manager

To start the Model Slice Manager, open the `fuel_rate_control` model, and select **Analysis** > **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**.

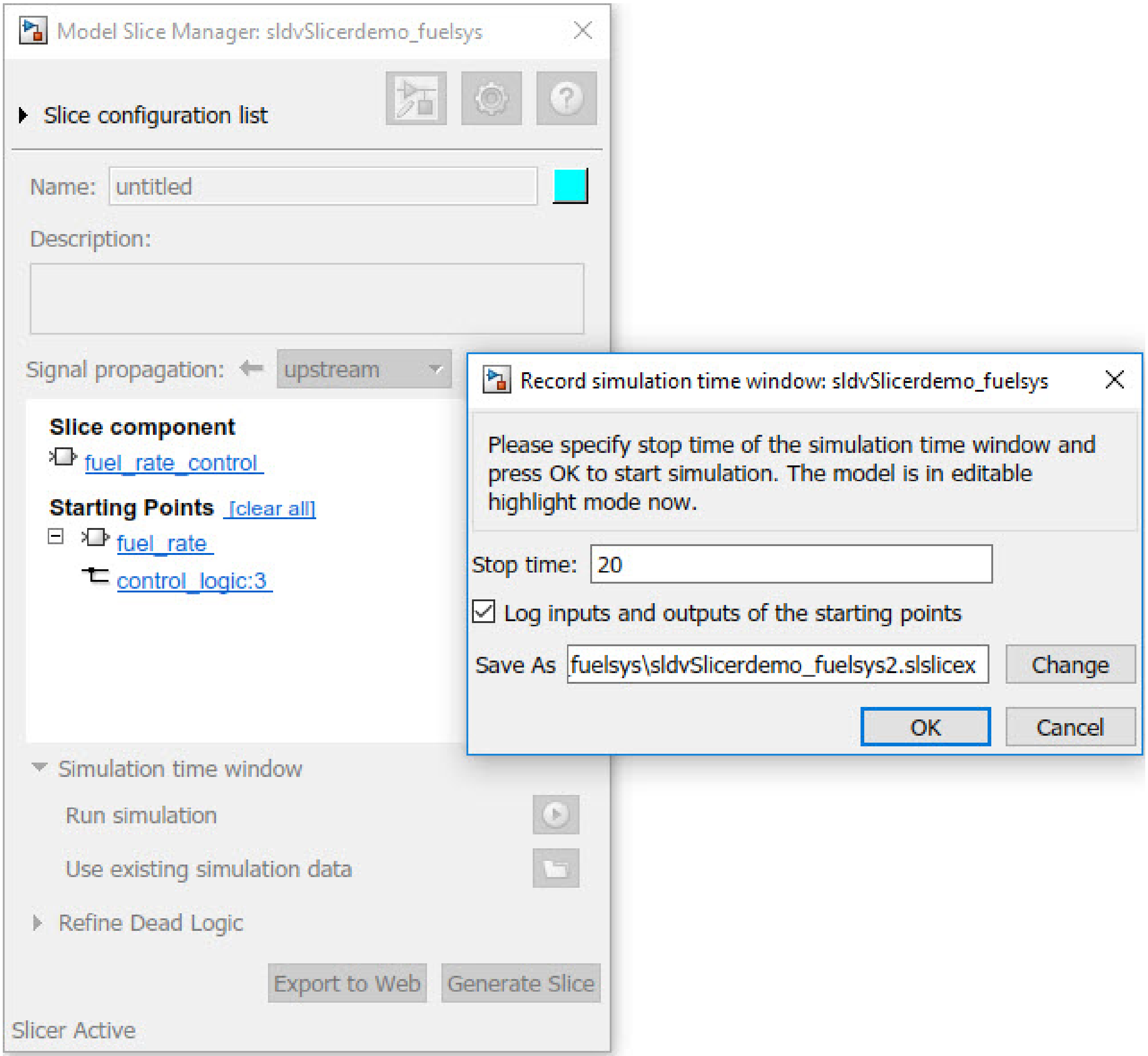
Step 2: Log input and output signals

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**.



Name: untitled

Description:

Signal propagation: upstream

Slice component: fuel_rate_control

Starting Points: fuel_rate, control_logic:3

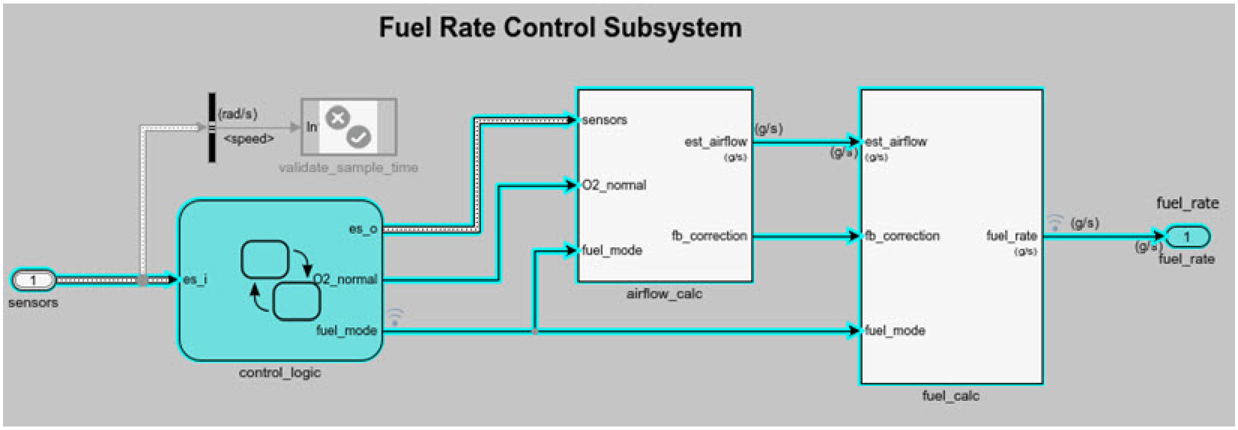
Slicer Active

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

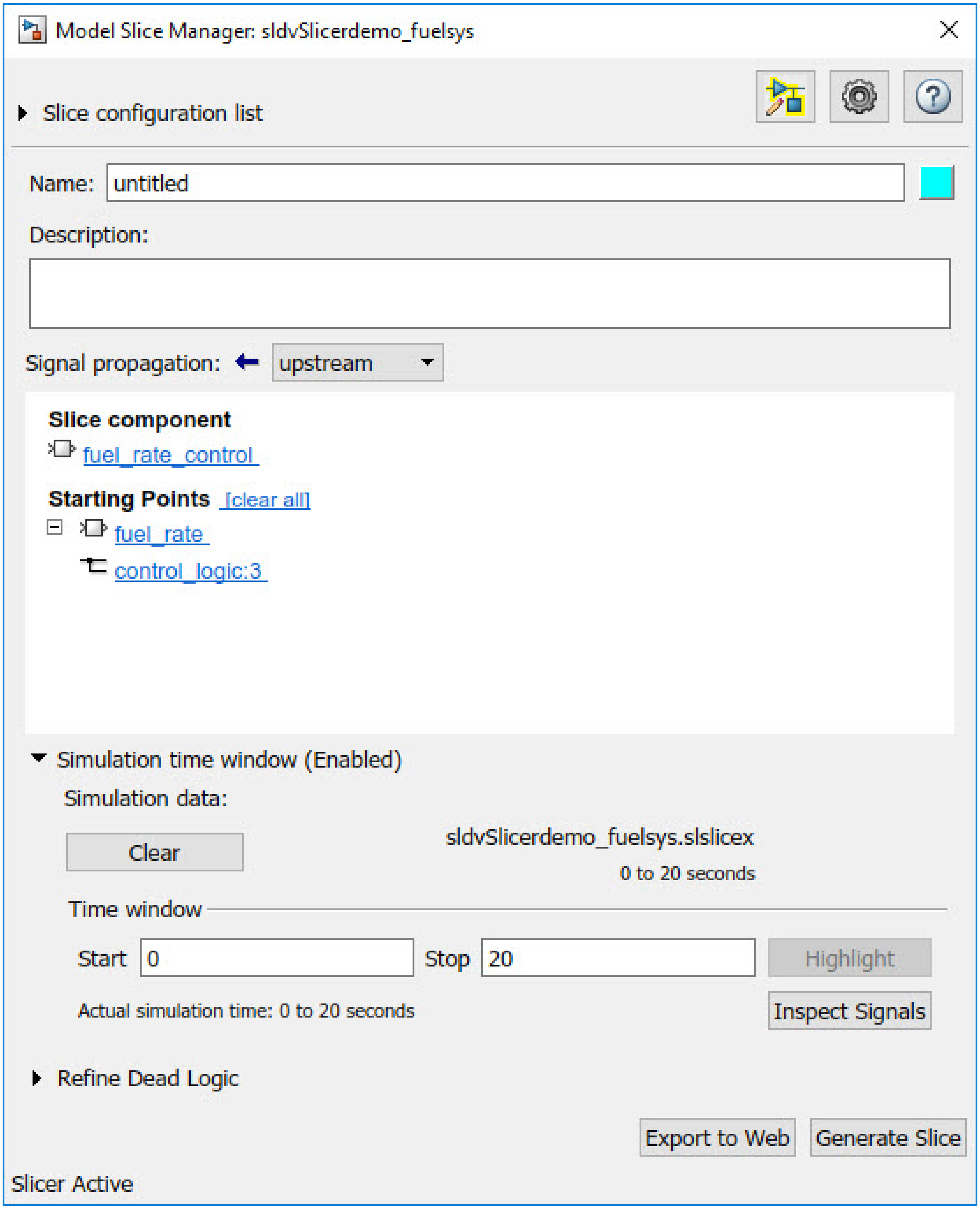
☑ Log inputs and outputs of the starting points

Save As fuelsys\sldvSlicerdemo_fuelsys2.slslicex



Step 3: Inspect signals

To open the Model Slicer Data Inspector, click **Inspect Signals**.



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 (Enabled)

Simulation data:

sldvSlicerdemo_fuelsys.slslicex

0 to 20 seconds

Time window

Start 0 Stop 20

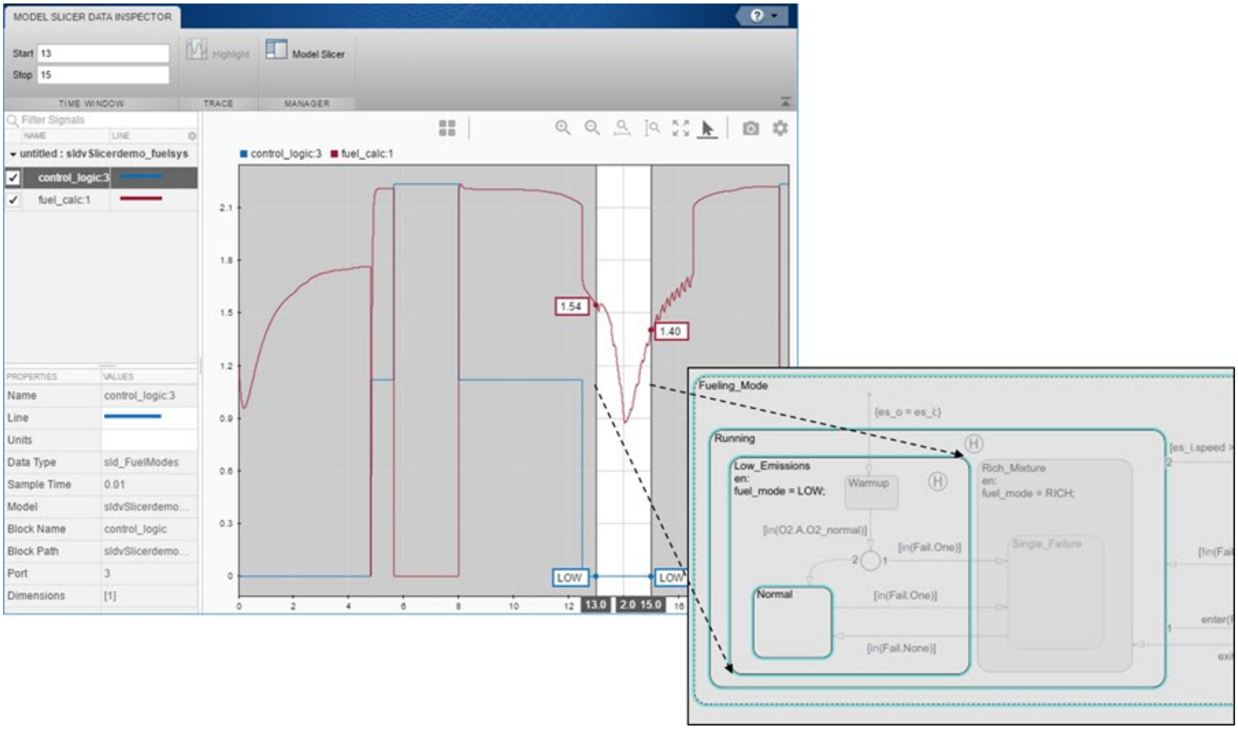
Actual simulation time: 0 to 20 seconds

Slicer Active

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.

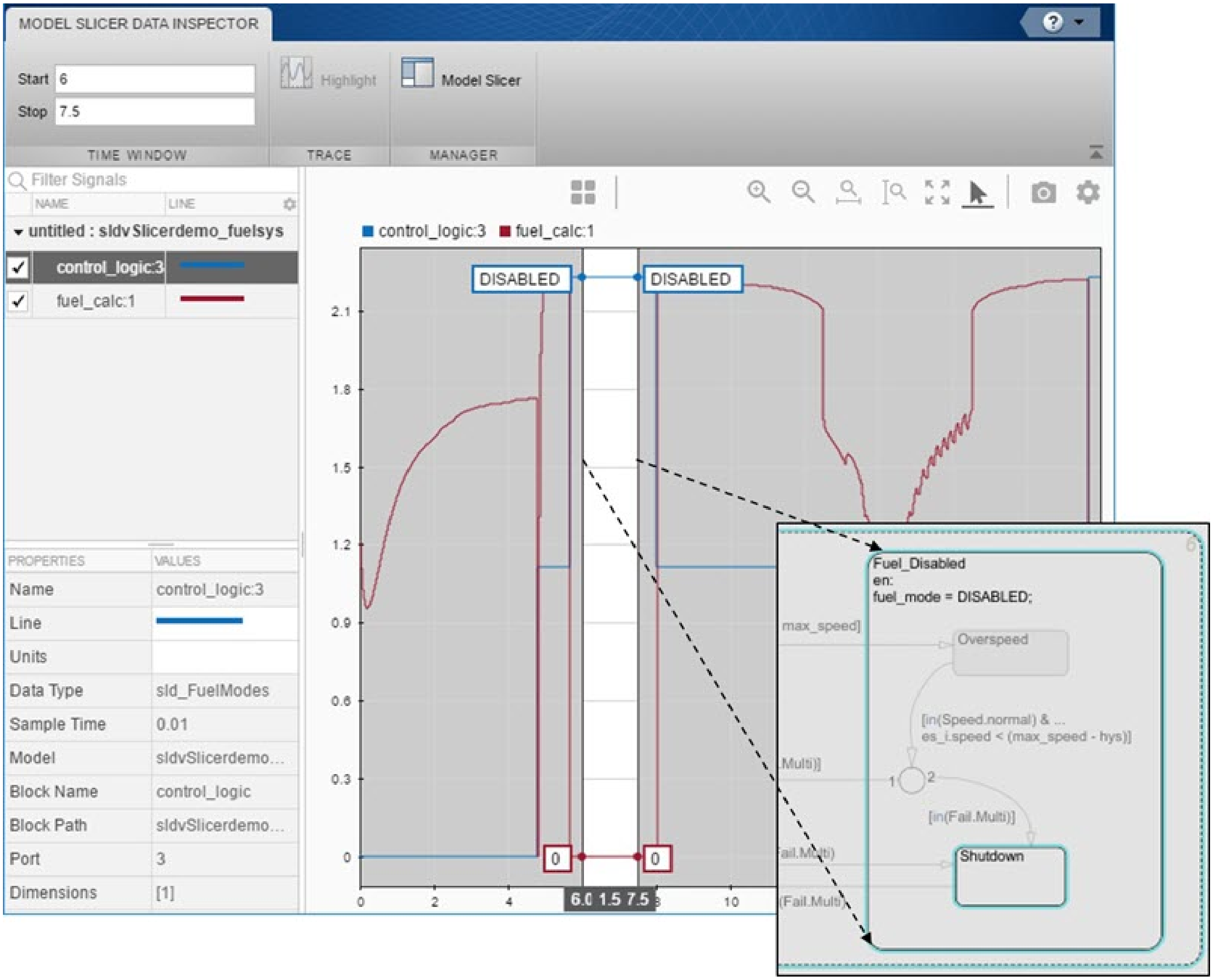


Start 13
Stop 15

☑ 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]

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.



Start 6

Stop 7.5

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]