Simulink® Check™

User's Guide

MATLAB®&SIMULINK®

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

Contents

Checking Systems Interactively

3

Check Systems Programmatically

4

Model Metrics

5

Create Model Advisor Checks

6

Model Advisor Customization

7

Model Slicer

8

1

Get Started

Simulink Check Product Description

Measure design quality, track verification activities, and verify compliance with standards

Simulink Check analyzes your models, requirements, and tests to assess design quality and compliance with standards. It provides industry-recognized checks and metrics that identify modeling standard and guideline violations as you design. Supported high-integrity software development standards include ISO 26262, DO-178C, DO-254, IEC 61508, ISO 25119, IEC 62304, and MathWorks Advisory Board (MAB) style guidelines. Simulink Check also supports secure coding standards such as CERT C, CWE, and ISO/IEC TS 17961. You can create custom checks to comply with your own standards or guidelines that can identify compliance issues right in the editor.

Simulink Check provides metrics such as size and complexity for assessing the status and quality of your design. The model testing dashboard consolidates data from your requirements-based testing activities to track testing status. Automatic model refactoring lets you replace modeling clones, reduce design complexity, and identify reusable content. The Model Slicer tool isolates problematic behavior in models and generates simplified models 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).

Assess and Verify Model Quality

With the Simulink Check product, you can use industry-recognized checks and metrics that identify standard and guideline violations. Supported high-integrity software development standards include the DO-178, ISO 26262, IEC 61508, IEC 62304, ISO 25119, and MathWorks Advisory Board (MAB) style guidelines. Use edit-time checking, to identify compliance issues as you develop your model. And, when you are done editing, to assess whether your model complies with size, architecture, and compliance requirements, run the Metrics Dashboard. The Metrics Dashboard contains widgets that visualize the metric data. To obtain detailed results, drill in to the data.

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

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

In this tutorial, you will learn to:

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

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

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

Detect and Fix Model Advisor Check Violations

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

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

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

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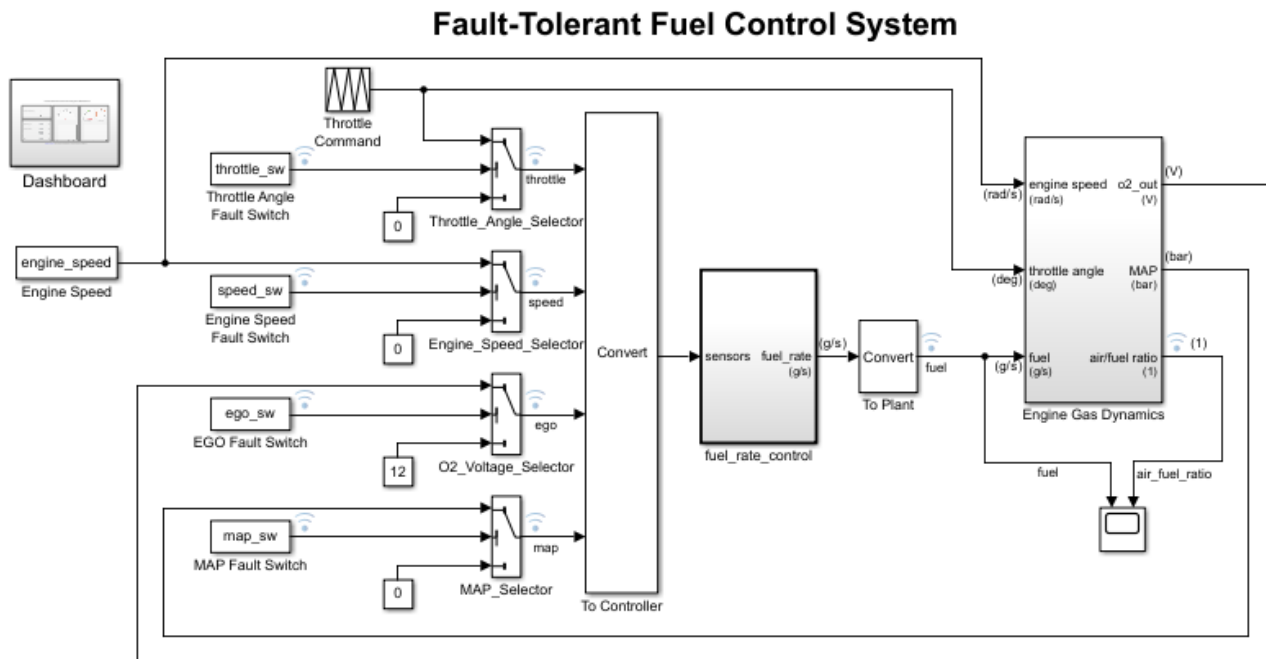
Fault-Tolerant Fuel Control System

Open the Dashboard subsystem to simulate any combination of sensor failures.   Copyright 1990-2017 The MathWorks, Inc.

Fuel Rate Control Subsystem
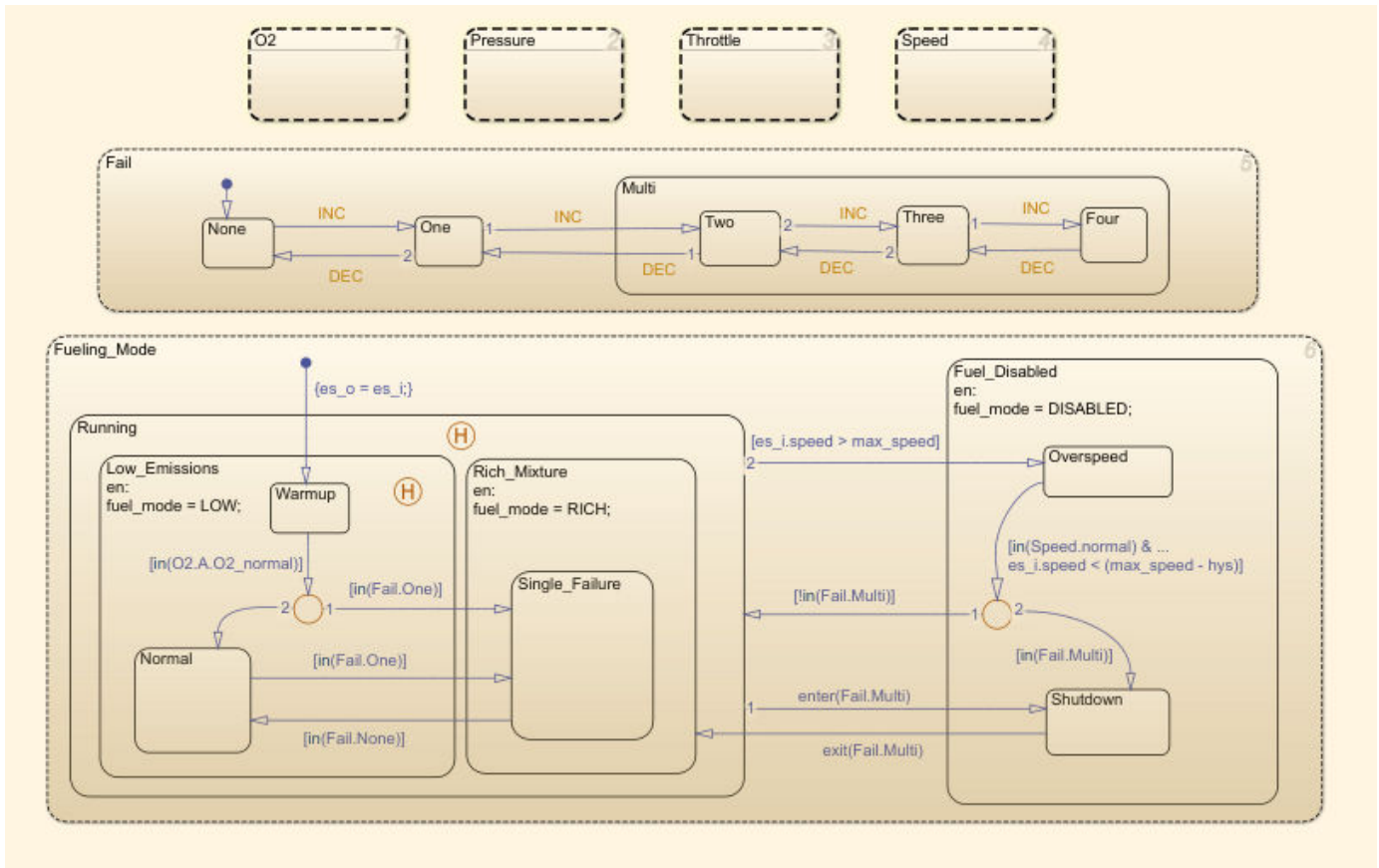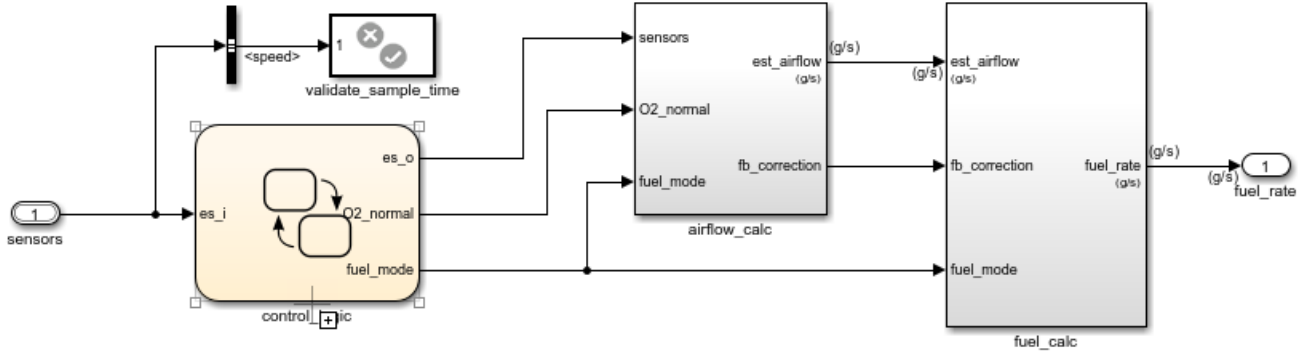
Detect and Fix Model Advisor Check Violations While You Edit

1   Set your current folder to a writeable directory.

2   Open the model `sldemo_fuelsys` by typing this command:

```
openExample('sldemo_fuelsys')
```

3   To use edit-time checking, on the **Modeling** tab, select **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the check box for **Edit-Time Checks**.

The highlighted blocks and subsystems indicate compliance issues.

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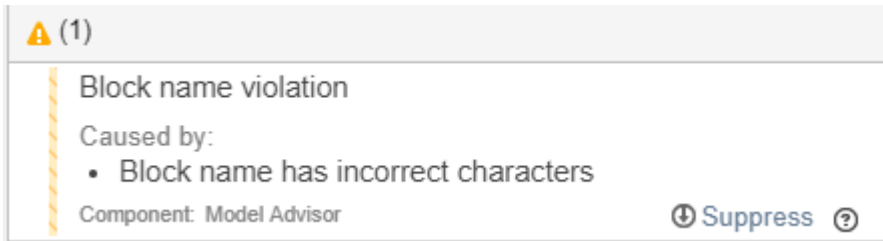
To exclude a block from a selected check, you can click **Suppress**.

5   Open the `Engine Gas Dynamics` subsystem by double-clicking it. Pause over the `air/fuel ratio` output port and click the warning icon.
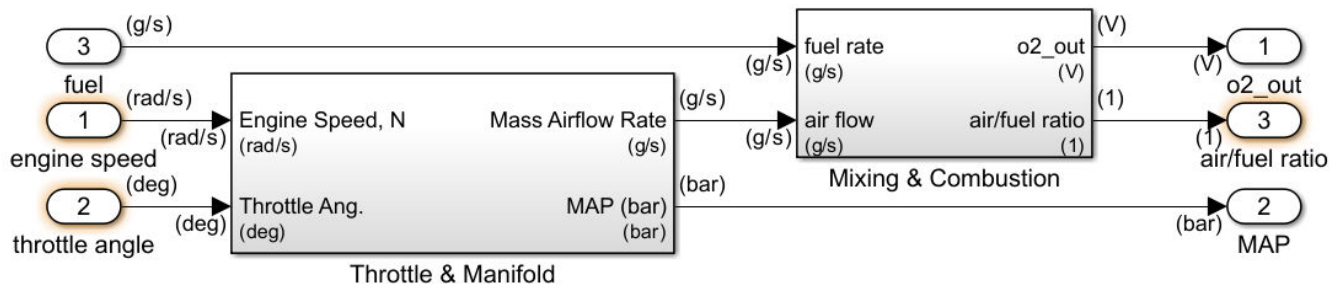
This output port returns warnings because its name violates two checks: "Check for unsupported block names" and "Check port block names".

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

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

Detect and Fix Model Advisor Check Violations Interactively

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

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

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

• **Check safety-related diagnostic settings for solvers**

• **Check safety-related diagnostic settings for sample time**

- **Check safety-related optimization settings for logic signals**

4 Right-click **DO-178C/DO-331 Checks** node, and then select **Run Selected Checks**.
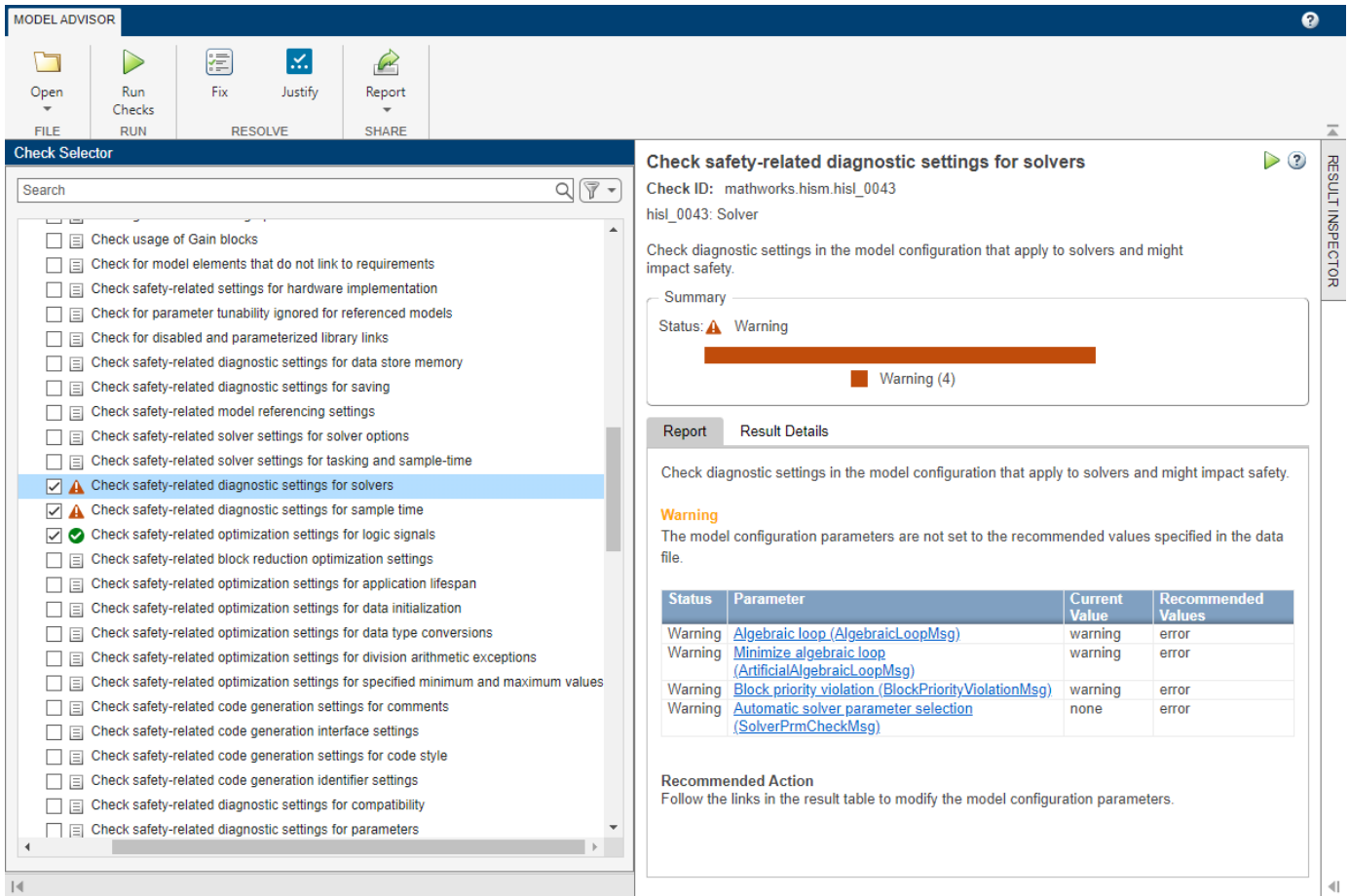
5 To review the configuration parameters that are not set to the recommended values, click **Check safety-related diagnostic settings for solvers**.

6 To update the parameters to the recommended values, in the toolstrip, click **Fix**.
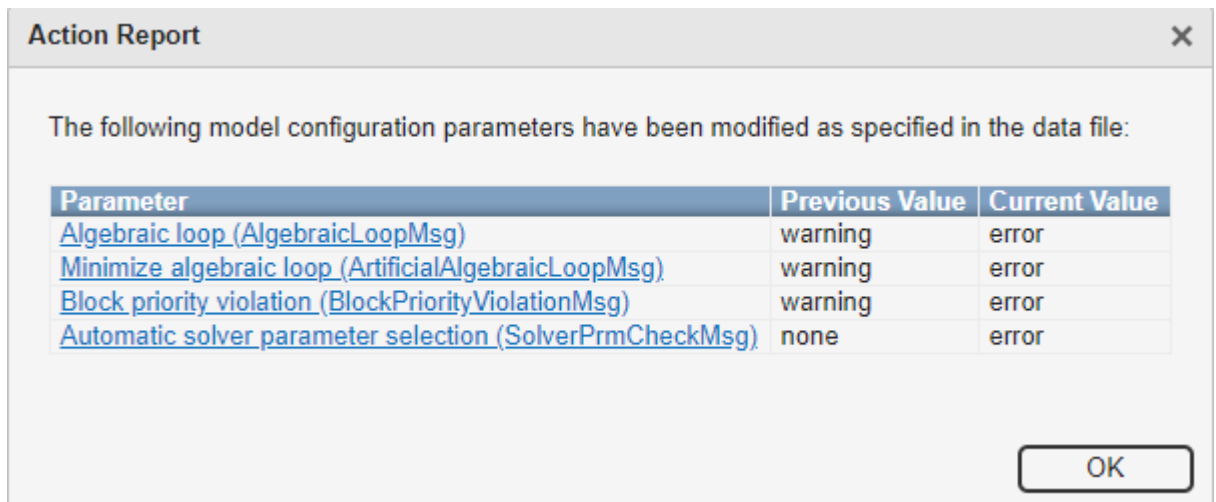
Action Report window displays the Model Advisor updates the parameters to the recommended values and details the result.

7 Repeat step 6 for the **Check safety-related diagnostic settings for sample time** check.

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

9 To generate a results report of the Simulink Check checks, select the **DO-178C/DO-331 Checks** node, and then, in the toolstrip, click **Report**.

10 Close the Model Advisor.

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

Collect Model Metric Data by Using the Metrics Dashboard

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

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

Analyze Metric Data

The Metrics Dashboard contains widgets that visualize metric data in these categories: size, modeling guideline compliance, and architecture. By default, some widgets contain metric threshold values. These values specify whether your metric data is compliant (which appears green in the widget) or produces a warning (which appears yellow in the widget). Metrics that do not have threshold values appear blue. Functions and classes are available for specifying noncompliant ranges and for changing the threshold values.
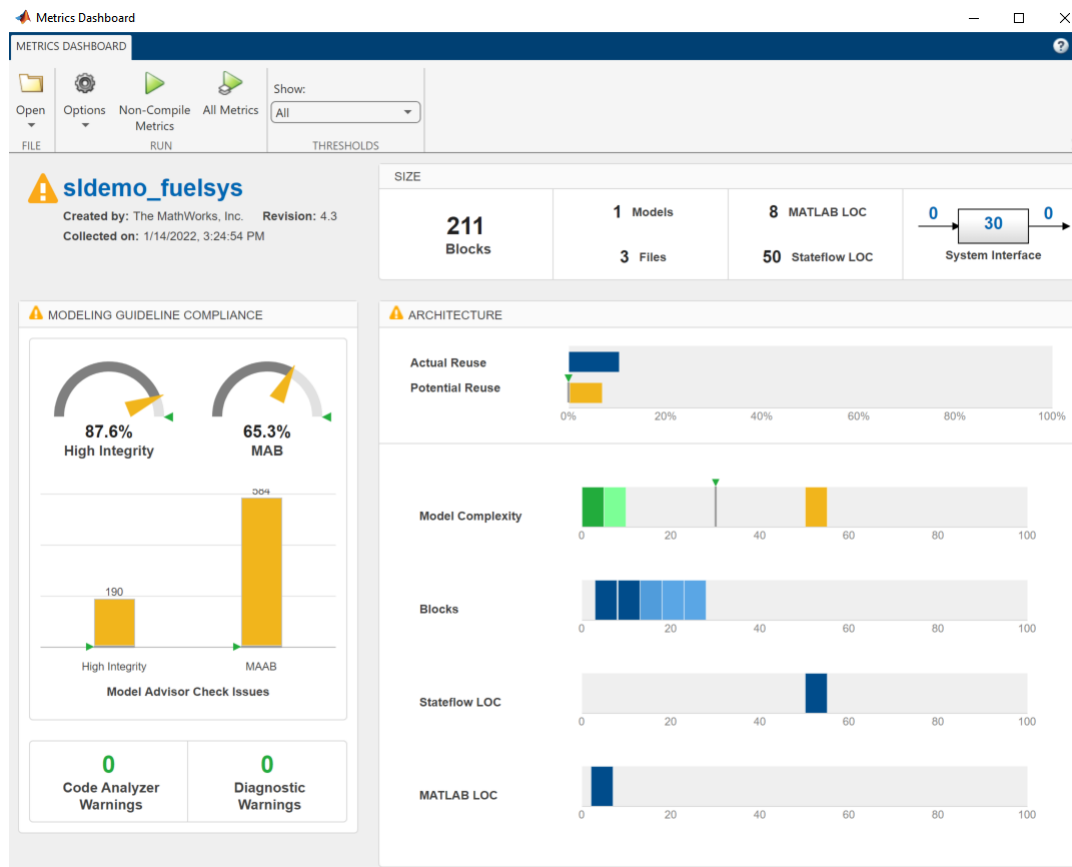
In the **Architecture** section of the dashboard, locate the **Model Complexity** widget. To view tooltips, pause over each vertical bar. This widget is a visual representation of the distribution of complexity

across the components in the model hierarchy. For each complexity range, a colored bar indicates the number of components that fall within that range. Darker green colors indicate more components. In this case, several components have a cyclomatic complexity value in the lowest range, while just one component has a higher complexity. This component has a cyclomatic complexity above 30. Components with cyclomatic complexity above 30 return warnings. For more information, see "Cyclomatic complexity metric"

Explore Metric Data

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

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

To see this component in the model, click the `control_logic` hyperlink.
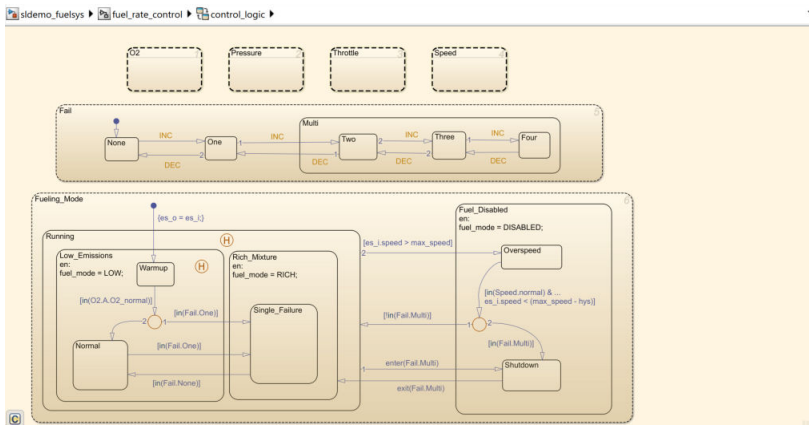
Refactor Model Based on Metric Data

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

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

Detect and Fix Compliance Issues

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

Explore Compliance Results in the Dashboard

1 In the Metrics Dashboard, locate the **Modeling Guideline Compliance** section. This section displays the percentage of high-integrity and MAB compliance checks that pass on all systems. The bar charts show the number of issues reported by the checks in each check group.
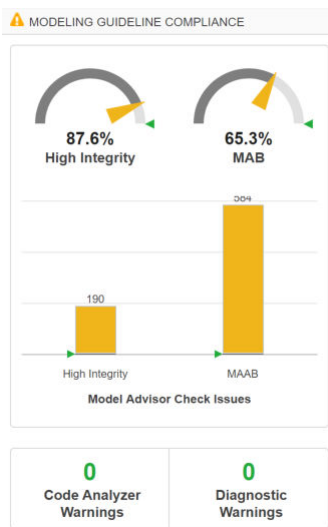
2 To see a table that details the number of compliance issues by component, click on the **High Integrity** bar chart. For more information on this metric, see "Model Advisor Check Issues for High-Integrity Systems".

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

Throttle Flow vs. Valve Angle and Pressure
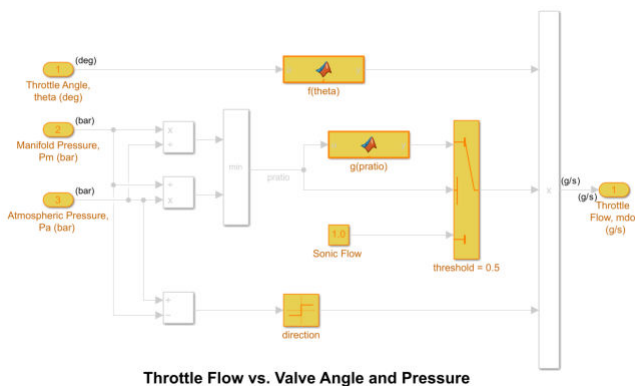
4 In the Metrics Dashboard, return to the main dashboard page by clicking the **Dashboard** icon.

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

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

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

7 Expand the `sldemo_fuelsys` node.

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

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

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

Update Model to Fix Compliance Issues

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

2 Change the parameter setting to what the **Recommended Value** column in the report indicates.

3 Click **Apply** and close the dialog box.

4 Close the Model Advisor report.

5 In the Model Advisor Highlighting dialog box, click the **Check safety-related diagnostic settings for compatibility** check.

The Model Advisor report opens.

6 For this check, repeat steps 1 through 4.

7 In the Model Advisor Highlighting dialog box, click the **Check safety-related diagnostic settings for bus connectivity** check.

8 For this check, repeat steps 1 through 4.

9 Close the Model Advisor Highlighting dialog box and return to the Metrics Dashboard table

Rerun Model Metrics

1 In the Metrics Dashboard table, return to the main dashboard page, by clicking **Dashboard**.

2 To rerun the model metrics, click **All Metrics**.

3 Confirm that the number of **High Integrity** check issues has reduced.
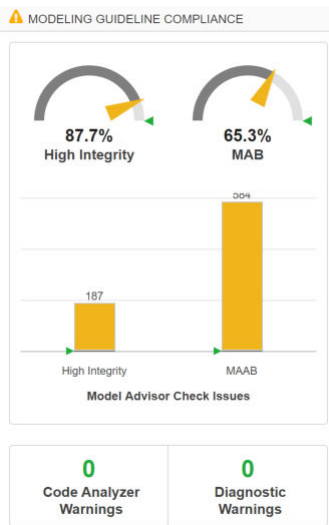
Next, use the **Actual Reuse** and **Potential Reuse** widgets to investigate and replace clones across a model hierarchy.

Refactor Models to Improve Component Reuse

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

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

Identify and Replace Clones with Links to Library Blocks

1   In the **Architecture** section, the blue bar in the Actual Reuse widget indicates the fraction of total number of subcomponents that are linked library blocks. Pause over the **Actual Reuse** widget to see more information. For this model, 10% of the total number of subcomponents are linked library blocks.

2   To see more details, click the blue bar. `System Lag`, `Throttle Command`, and `CheckRange` are linked library blocks.

Metrics Dashboard

METRICS DETAILS

Dashboard | Table | Tree

VIEW | VISUALIZATIONS

Library Linked Component Content
Metric that calculates the fraction of total number of subcomponents that are linked library blocks.

Clear Filters

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

3   Return to the main dashboard page.

4   In the **Architecture** section, the **Potential Reuse** bar indicates that the model contains clones. Pause over **Potential Reuse**. For this model, 7% of the subcomponents are clones.

5   To see more details, click the yellow bar. `Pressure.map_estimate` and `Throttle.throttle_estimate` are clones of each other.

6   To determine whether these clones are candidates for replacement with linked library blocks, click **Open Conversion Tool**.

The **Clone Detector** app opens as a new tab in the model.

7   In the **Clone Detection Results and Actions** pane, click the **Clone Results** tab.

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

8   Expand the clone group.

This clone group consists of two subcharts.
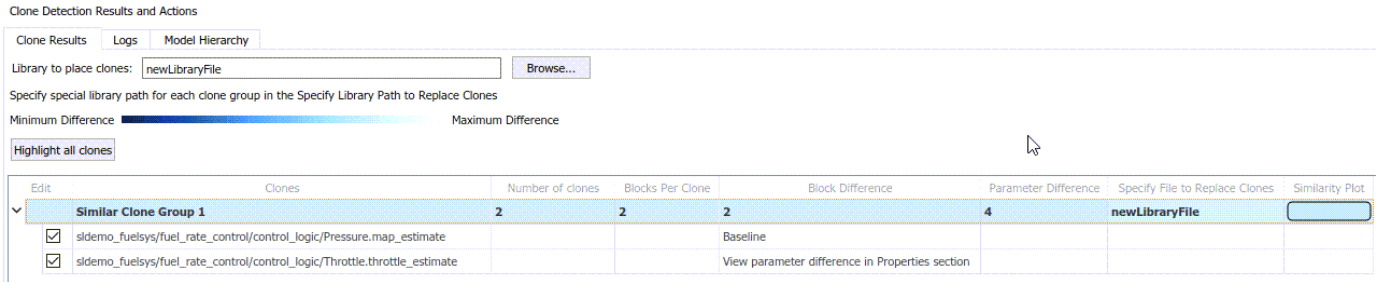
9   To determine parameter differences, in the **Block Difference** column, click **View parameter difference**.

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

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

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

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

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

12 Close the Metrics Dashboard and the model.

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

Explore Other Options

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

Task	Reference
Simplify and debug complex models.	"Highlight Functional Dependencies" on page 8-2
Run Model Advisor checks for compliance with safety standards associated with High-Integrity System Modeling and MAB Control Algorithm Modeling guidelines.	"Check Model Compliance by Using the Model Advisor" on page 3-2
Write custom Model Advisor checks.	"Define Custom Model Advisor Checks" on page 6-45
Create and deploy a custom Model Advisor configuration.	"Create and Deploy a Model Advisor Custom Configuration" on page 7-23 and "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3

Task	Reference
Learn more about how to use the Metrics Dashboard to collect and view metric data for quality assessment.	"Collect and Explore Metric Data by Using the Metrics Dashboard" on page 5-2
Configure compliance metrics, add metric thresholds, and customize the Metrics Dashboard layout.	"Customize Metrics Dashboard Layout and Functionality" on page 5-36
Use the Model Transformer tool and the Clone Detector app to refactor a model to improve model componentization and readability and enable reuse.	"Transform Model to Variant System" on page 3-23 and "Enable Component Reuse by Using Clone Detection" on page 3-33
Learn more about how to use Simulink products to test models and code, check for design errors, check against standards, measure coverage, and validate the system.	"Verification and Validation"

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

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

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

   `openExample('sldemo_mdlref_basic');`

2 To open the Model Slicer Manager, on the Model Verification, Validation, and Test section of the **Apps** tab, click **Model Slicer**.

3 In the Model Slice Manager, click the arrow to expand the **Slicer configuration list**.

4 Set the slice properties:

   • **Name**: `Slice1`

   • **Color**: ▮ (magenta)

   • **Signal Propagation**: `upstream`

   Model Slicer can also highlight the constructs downstream of or bidirectionally from a block in your model, depending on which direction you want to trace the signal propagation.

5 Add `CounterC` as a starting point. In the model, right-click `CounterC` and select **Model Slicer > Add as Starting Point**.
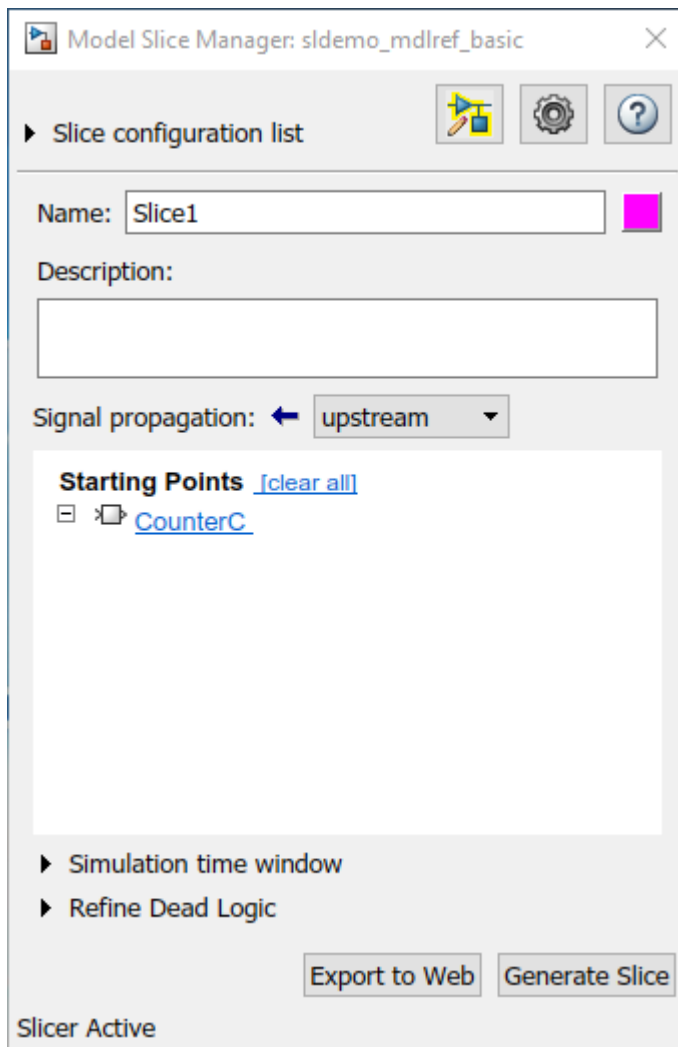
The Model Slicer now highlights the upstream constructs that affect `CounterC`.
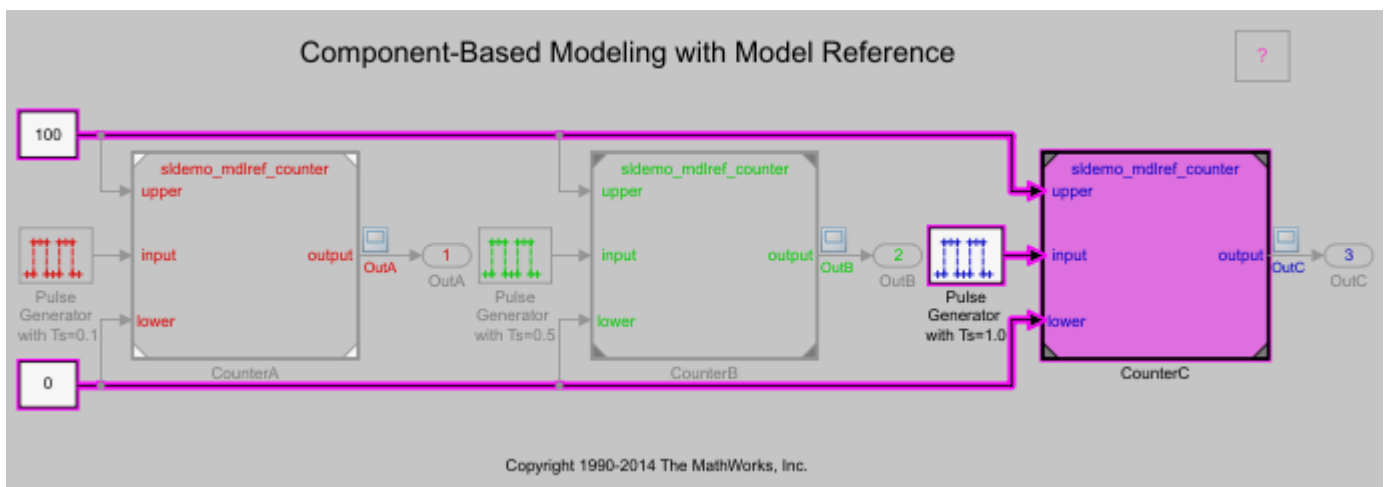
6 In the Model Slice Manager, click **Generate slice**.

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

Component-Based Modeling with Model Reference

Copyright 1990-2014 The MathWorks, Inc.

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

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

See Also

More About

- "Model Slicer Considerations and Limitations" on page 8-43
- "Highlight Functional Dependencies" on page 8-2
- "Refine Highlighted Model" on page 8-12

Assess Requirements-Based Testing Quality by Using the Model Testing Dashboard

You can assess the status of your model testing activities by using the metrics in the Model Testing Dashboard. When you test your models against requirements, you maintain traceability between the requirements, models, test cases, and results. The dashboard helps you to track the status of these artifacts and the traceability relationships between them. Each metric in the dashboard measures a different aspect of the quality of the testing artifacts and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. For more information, see "Assess the Completeness of Requirements-Based Testing in Accordance with ISO 26262" on page 5-76.

From the dashboard, you can identify and fix testing issues. Update the dashboard metrics to track your progress toward testing compliance.

Open the Project and Model Testing Dashboard

The Model Testing Dashboard shows data on the traceability and testing status of each *unit* in your project. A unit is a functional entity in your software architecture that you can execute and test independently or as part of larger system tests. You can label models as units in the Model Testing Dashboard. If you do not specify the models that are considered units, then the dashboard considers a model to be a unit if it does not reference other models. The dashboard considers each model in your project to represent a unit because you use models to design and test the algorithms.

1   Open the project that contains the models and testing artifacts. For this example, at the MATLAB command line, enter `dashboardCCProjectStart('incomplete')`:

    `dashboardCCProjectStart('incomplete')`

2   Open the Model Testing Dashboard by using one of these approaches:

    • On the **Project** tab, click **Model Testing Dashboard**.
    • At the MATLAB command line, enter `modelTestingDashboard`.

3   The first time that you open the dashboard for the project, the dashboard must identify the artifacts in the project and collect traceability information.
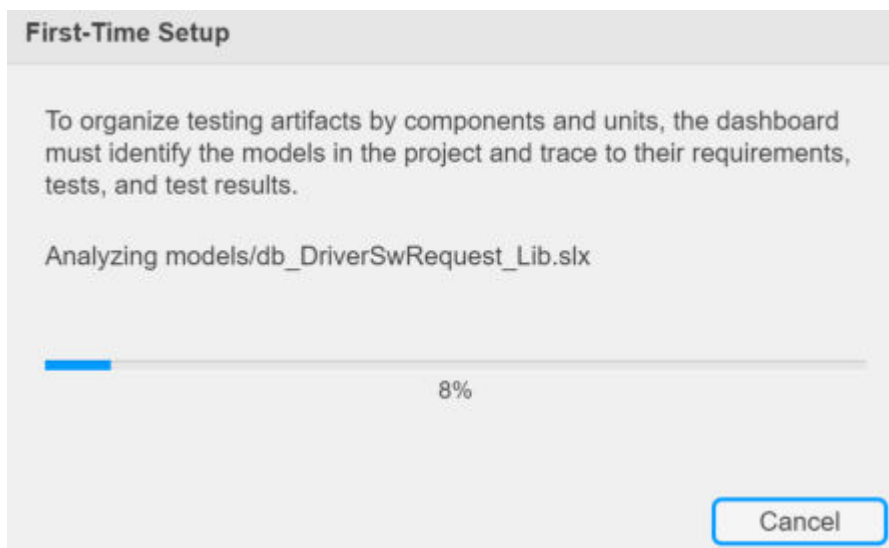
The dashboard performs an initial traceability analysis and collects metric results for the metrics available in your installation. Collecting results for all MathWorks® metrics requires licenses for Simulink Check, Requirements Toolbox™, and Simulink Test. If metric results have been collected, viewing the results requires only a Simulink Check license.
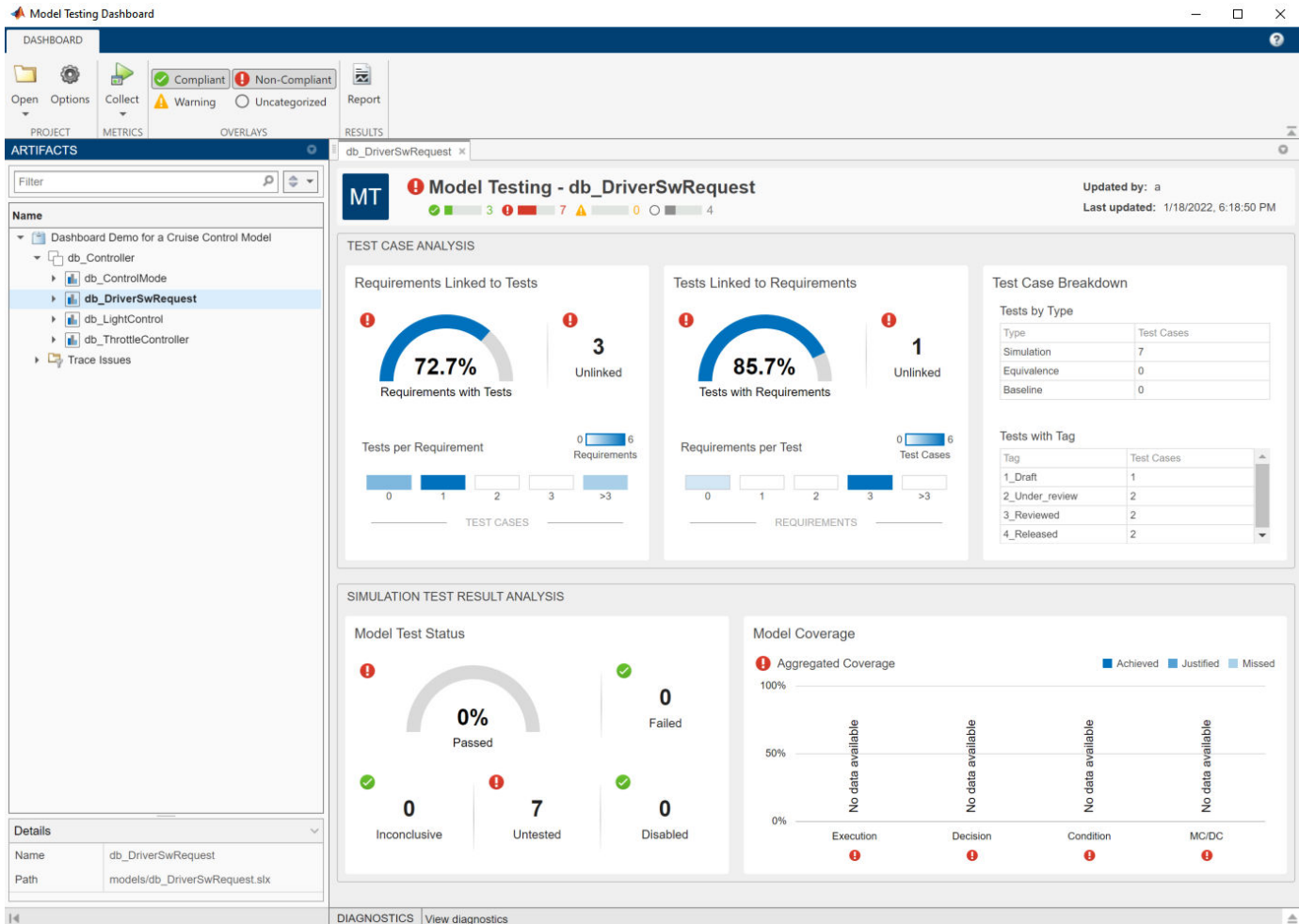
The dashboard analyzes the traceability links from the artifacts to the models in the project and populates the widgets with metric results for the unit that is selected in the **Artifacts** panel.

Assess Traceability of Artifacts

When the dashboard collects and reports metric data, it scopes the results to the artifacts in one unit in the project. Use the **Artifacts** panel to view each unit in the project, represented by the name of its model, and the artifacts that trace to it.

1   In the **Artifacts** panel, click the unit **db_DriverSwRequest**. The dashboard widgets populate with metric data from the artifacts in this unit.

2   In the **Artifacts** panel, expand the section for the unit. Click the arrow to the left of **db_DriverSwRequest**. The section below the unit shows the artifacts of each type that trace to the unit.

3 Expand the **Functional Requirements** section and then expand the **Implemented** and **Upstream** sections. This unit implements the requirements in the file db_SoftwareReqs.slreqx and links to the upstream, system-level requirements in db_SystemReqs.slreqx. Click the arrow to the left of a file name to see the individual requirements that trace to the model.

You can explore the units and sections in the **Artifacts** panel to see which requirements, test cases, and test results trace to each unit in the project. For more information on how the dashboard analyzes this traceability, see "Trace Artifacts to Units for Model Testing Analysis" on page 5-70.

Explore Metric Results for a Unit

1 On the **Artifacts** panel, click the unit **db_DriverSwRequest** to view the **Model Testing** results. The dashboard widgets populate with the metric results for the unit.

2 In the **Test Case Analysis** section of the dashboard, locate the **Tests with Requirements** widget. To view tooltips with details about the results, point to the sections of the gauge or to the percentage result.
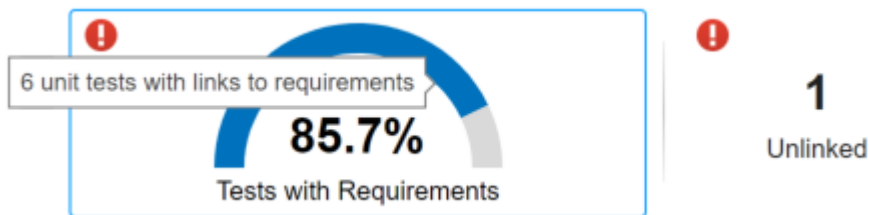
3 To explore the metric data in more detail, click an individual metric widget. For example, click the **Tests with Requirements** widget to view the **Metric Details** for the metric.

The table shows each test case for the unit, the test file containing each test case, and whether the test case is linked to requirements.

4 The table shows that the test case `Detect long decrement` is missing linked requirements. You can use the Model Testing Dashboard to open the test case in the Test Manager. In the **Artifact** column, click **Detect long decrement**.

5 At the top of the Model Testing Dashboard, there is a breadcrumb trail from the **Metric Details** back to the **Model Testing** results.
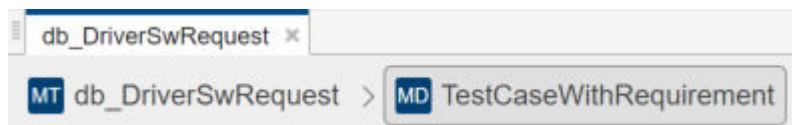
Click the breadcrumb button for **db_DriverSwRequest** to return to the **Model Testing** results for the unit.

You can click on any of the widgets in the dashboard to view the details of their metric results. Use the hyperlinks in the tables to open the artifacts and address testing gaps. For more information on using the data in the dashboard, see "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51.

Track Testing Status of a Project Using the Model Testing Dashboard

To use the Model Testing Dashboard to track your testing activities, set up and maintain your project using the best practices described in "Manage Requirements-Based Testing Artifacts for Analysis in the Model Testing Dashboard" on page 5-68. As you develop and test your models, use the dashboard to identify testing gaps, fix the underlying artifacts, and track your progress towards model testing completion. For more information on finding and addressing gaps in your model testing, see "Fix Requirements-Based Testing Issues" on page 5-61.

See Also
"Model Testing Metrics"

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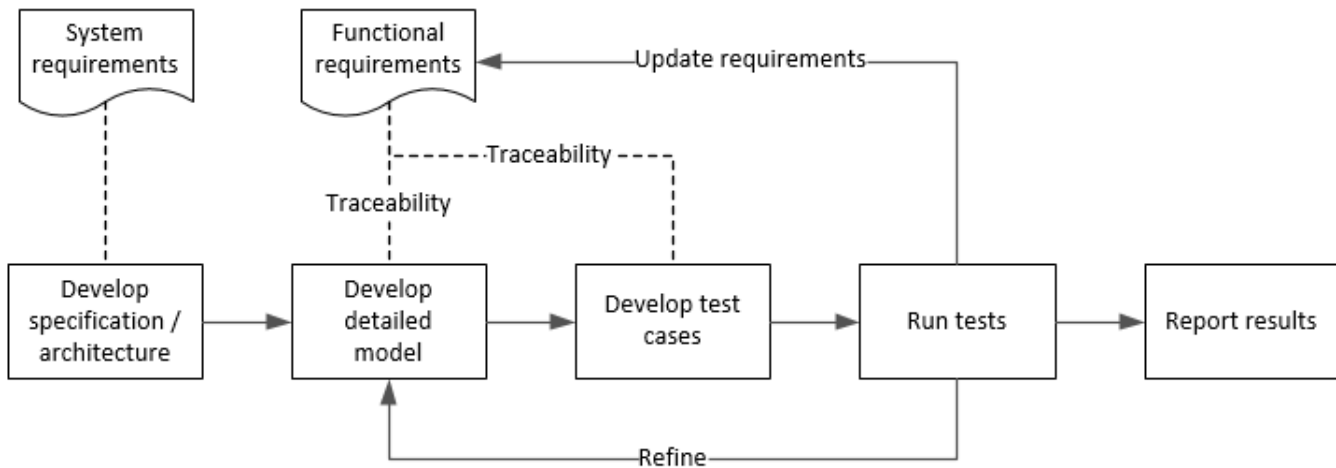
In this example, you conduct a simple test of two requirements in the set:

- 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

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

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

2   In the project `models` folder, open the `simulinkCruiseAddReqExample.slx` model.

3   Display the requirements. Click the ▥ icon in the lower-right corner of the model canvas, and select **Requirements**. The requirements appear below the model canvas.

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

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

Link Requirements to Tests

Link the requirements to the test case.

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

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

A link to the `Safety Tests` test case is added to **Verified by**. The yellow bars in the **Verified** column indicate that the requirements are not verified.
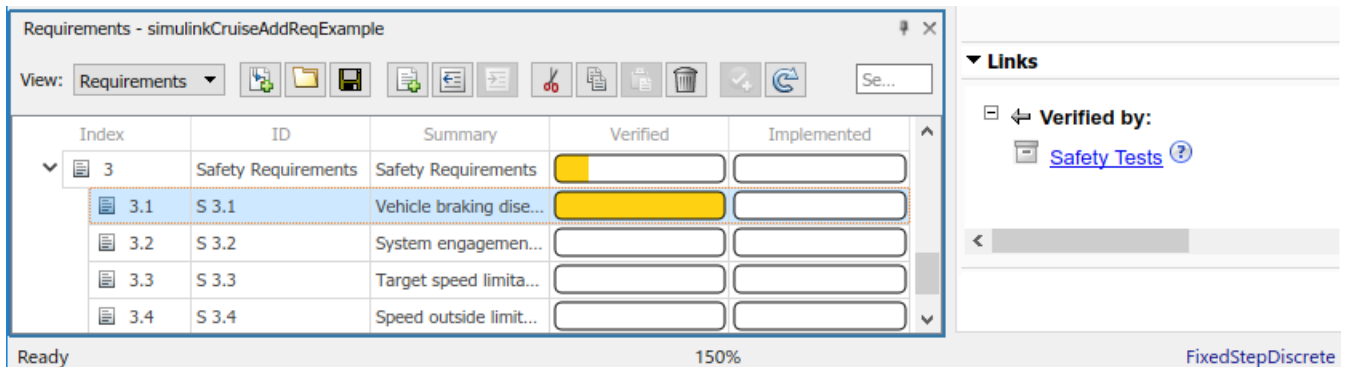
2 Also add a link for item S 3.4.

Run the Test

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

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

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

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

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

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

2 When the test finishes, review the results. The Test Manager shows that both assessments pass and the plot provides the detailed results of each `verify` statement.
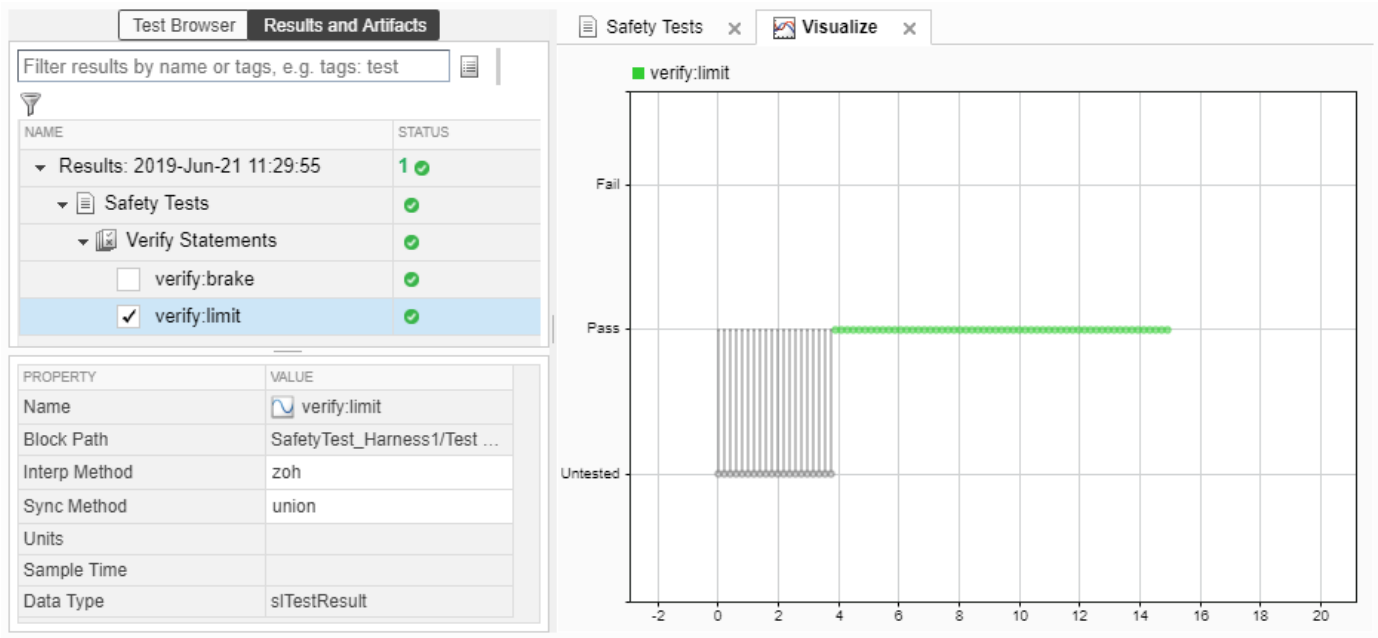
3 Return to the model and refresh the Requirements. The green bar in the **Verified** column indicates that the requirement has been successfully verified.
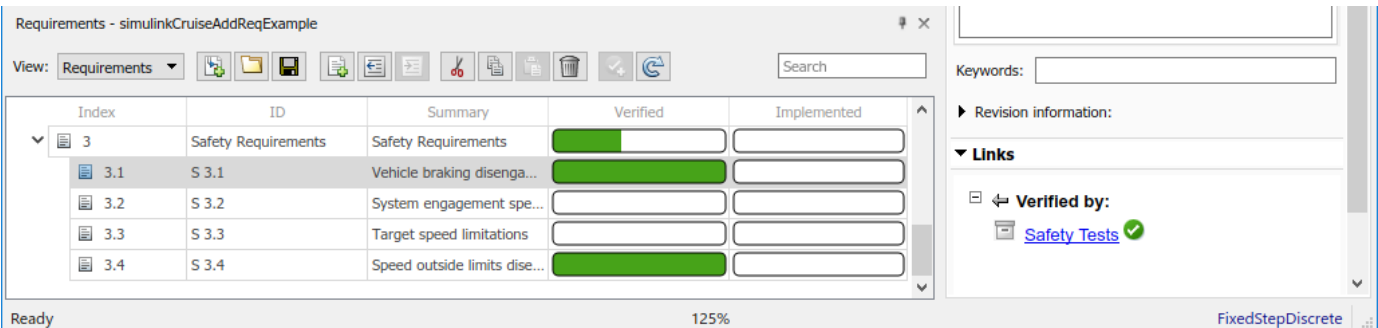
Report the Results

1 Create a report using a custom Microsoft Word template.

    a From the Test Manager results, right-click the test case name. Select **Create Report**.

    b In the Create Test Result Report dialog box, set the options:

- Title — `SafetyTest`
- Results for — `All Tests`
- File Format — `DOCX`
- For the other options, keep the default selections.

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

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

    e Click **Create**.

2 Review the report.

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

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

See Also

Related Examples

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

External Websites

- Requirements-Based Testing Workflow

Analyze a Model for Standards Compliance and Design Errors

Standards and Analysis Overview

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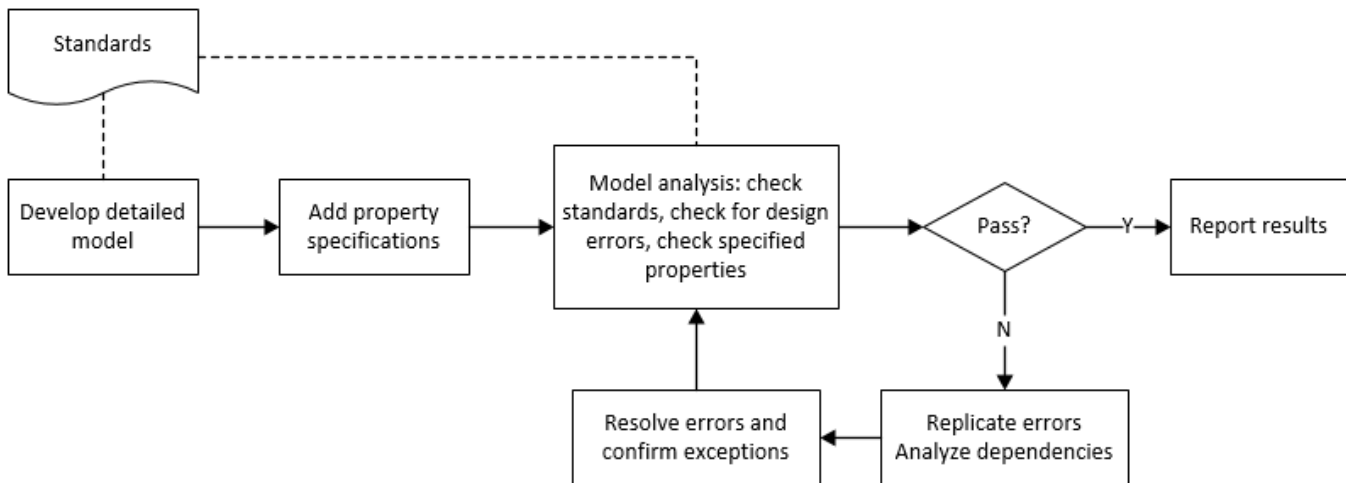
Check Model for Style Guideline Violations and Design Errors

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

Check Model for MAB Style Guideline Violations

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

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

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

2   Open the model. On the command line, enter

```
open_system simulinkCruiseErrorAndStandardsExample
```

3   In the **Modeling** tab, select **Model Advisor**.

4   Click OK to choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.

5   Check your model for MAB style guideline violations using Simulink Check.

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

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

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

    c   To review the configuration parameter settings that violate MAB style guidelines, run the **Check model diagnostic parameters** check.

    d   The analysis results appear in the right pane on the **Report** tab. Report displays the violation details and the recommended action.

    e   Click the parameter hyperlinks, which opens the Configuration Parameters dialog box, and update the model diagnostic parameters. Save the model.

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

    g   To generate a results report of the Simulink Check checks, select the **MAB Checks** node, and then, from the toolstrip, click**Report**.

Check Model for Design Errors

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

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

2   If not already checked, click the box beside **Design Error Detection**. All checks in the folder are selected.

3   From the tool strip, click **Run Check**.

4   After the Model Advisor analysis, from the tool strip, click **Report**. This generates a HTML report of the check analysis.

5   In the generated report, click a **Simulink Design Verifier Results Summary**hyperlink. The dialog box provides tools to help you diagnose errors and warnings in your model.

    a   Review the analysis results on the model. Click the `Compute target speed` subsystem. 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 or create one if it does not already exist.

    c   View tests and export test cases.

    d   Review the analysis report. To see a detailed analysis report, click **HTML** or **PDF**.

See Also

Related Examples

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

Perform Functional Testing and Analyze Test Coverage

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

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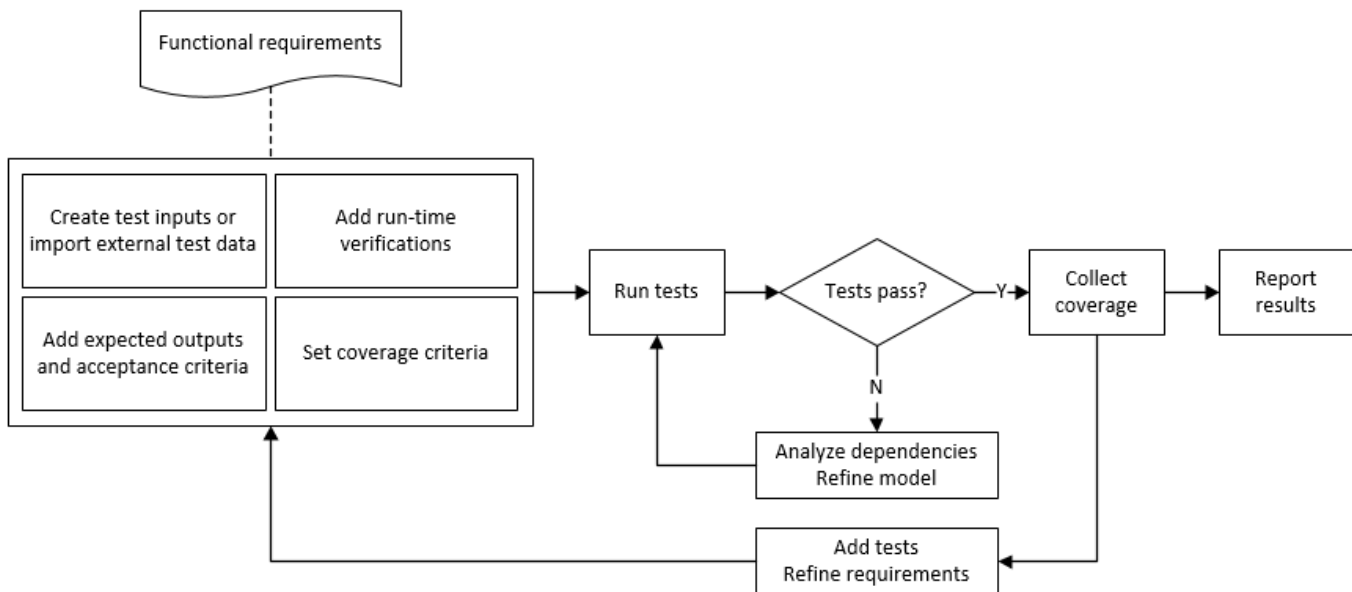
Incrementally Increase Test Coverage Using Test Case Generation

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

Explore the Test Harness and the Model

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

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'slVerificationCruiseStart'))
```
2 Open the model and the test harness. At the command line, enter:

```
open_system simulinkCruiseAddReqExample
sltest.harness.open('simulinkCruiseAddReqExample','SafetyTest_Harness1')
```
3 Load the test suite from "Test Model Against Requirements and Report Results" (Simulink Test) and open the Simulink Test Manager. At the command line, enter:

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

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

- Brake pedal is pressed

- Speed exceeds a limit

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

Measure Model Coverage

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

- Select **Record coverage for referenced models**

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

- Select **Decision**, **Condition**, and **MCDC**.

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

Generate Tests to Increase Model Coverage

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

Alternatively, you can create and use tests to increase coverage programmatically by using `sltest.testmanager.addTestsForMissingCoverage` and `sltest.testmanager.TestOptions`.

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

You can analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. For handwritten code, you typically 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, you refine the code and add tests.

In this example, you generate code and demonstrate that the code execution produces equivalent results to the model by using the same test cases and baseline results. Then you compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.
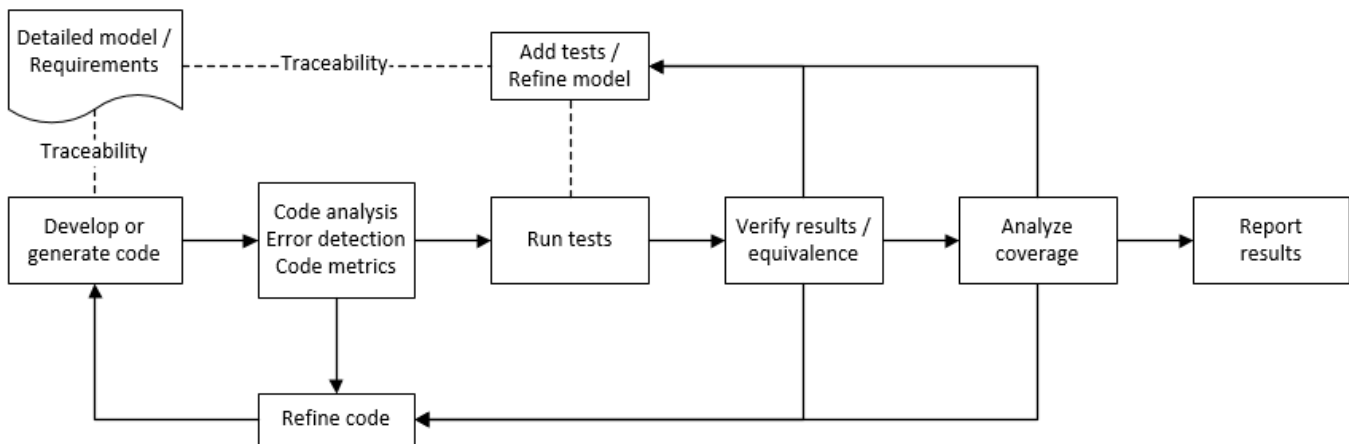
Analyze Code for Defects, Metrics, and MISRA C:2012

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 and defects. 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`.

Compute target speed
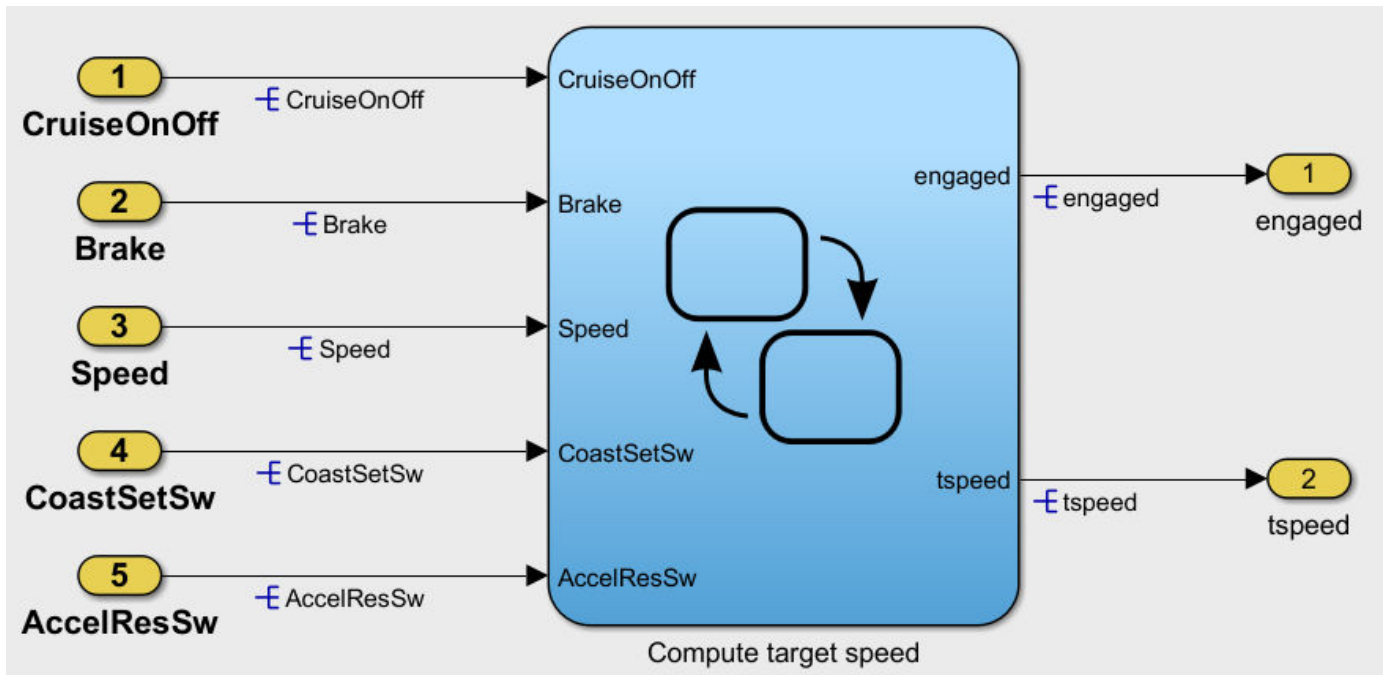
Run Code Generator Checks

Before you generate code from your model, use the Code Generation Advisor to check your model so that it generates code more compliant with MISRA C and more compatible with Polyspace.

1   Right-click Compute target speed and select **C/C++ Code > Code Generation Advisor**.

2   Select the Code Generation Advisor folder. In the right pane, move `Polyspace` to **Selected objectives - prioritized**. The `MISRA C:2012 guidelines` objective is already selected.

3   Click **Run Selected Checks**.

The Code Generation Advisor checks whether the model includes blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this

model, the check for incompatible blocks passes, but some configuration settings 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
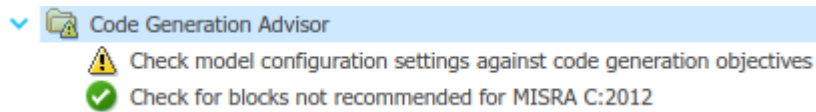
4 Click the check that did not pass. Accept the parameter changes by selecting **Modify Parameters**.

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

Run Model Advisor Checks

Before you generate code from your model, use the Model Advisor to check your model for MISRA C and Polyspace compliance. This example shows you how to use the Model Advisor to check your model before generating code.

1 At the bottom of the Code Generation Advisor window, select **Model Advisor**.

2 Under the **By Task** folder, select the **Modeling Standards for MISRA C:2012** advisor checks.

3 Click **Run Checks** and review the results.

4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

Generate and Analyze Code

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

1 In the Simulink editor, right-click Compute target speed and select **C/C++ Code > Build This Subsystem**.

2 Use the default settings for the tunable parameters and select **Build**.

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

4 Click **Configure** to choose more advanced Polyspace analysis options in the Polyspace configuration window.
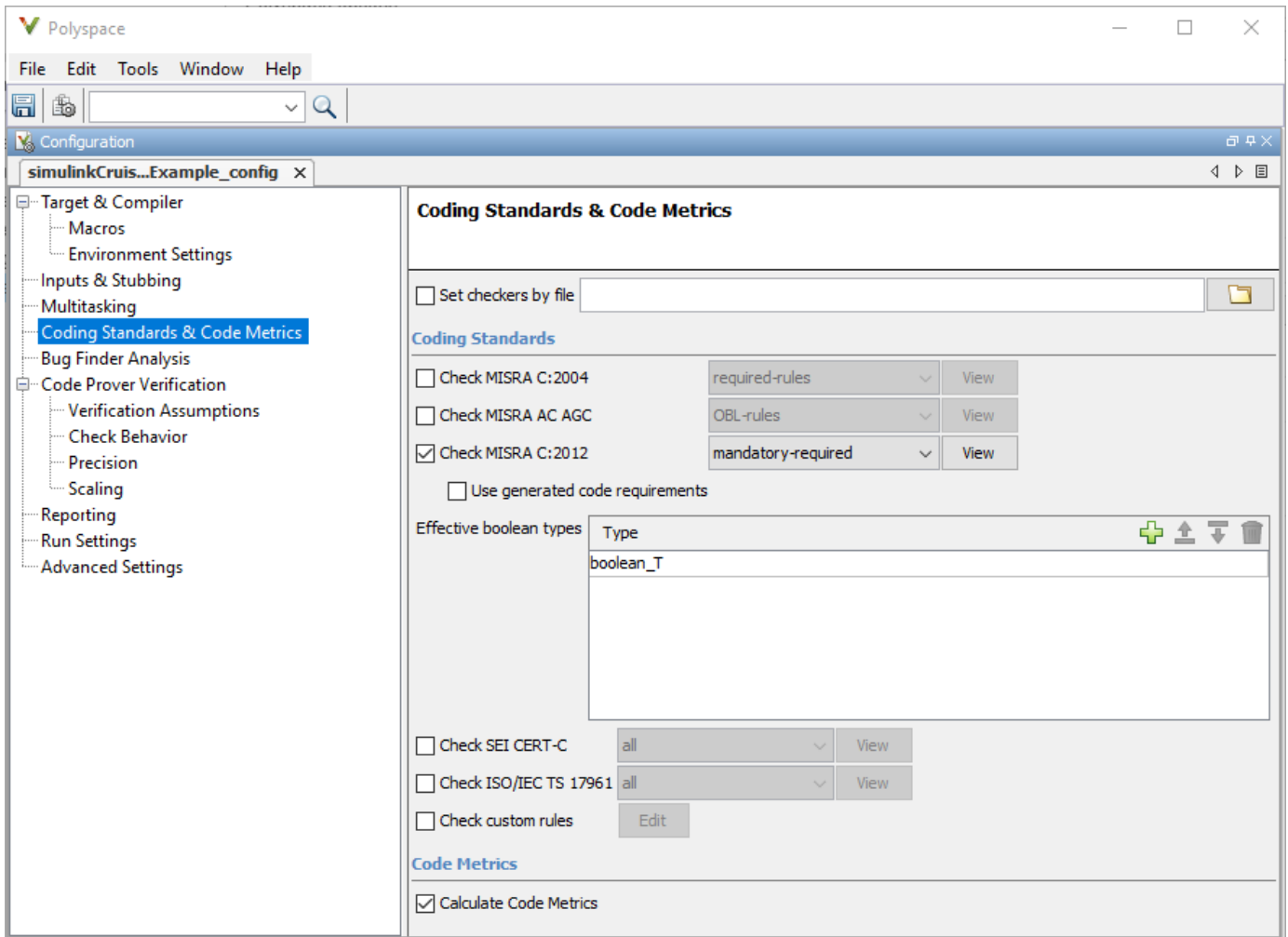
5   On the left pane, click **Coding Standards & Code Metrics**, then select **Calculate Code Metrics** to enable code metric calculations for your generated code.

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. You can see the progress of the analysis in the MATLAB Command Window. After the analysis finishes, the Polyspace environment opens.

Review Results

The Polyspace environment shows you the results of the static code analysis.

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

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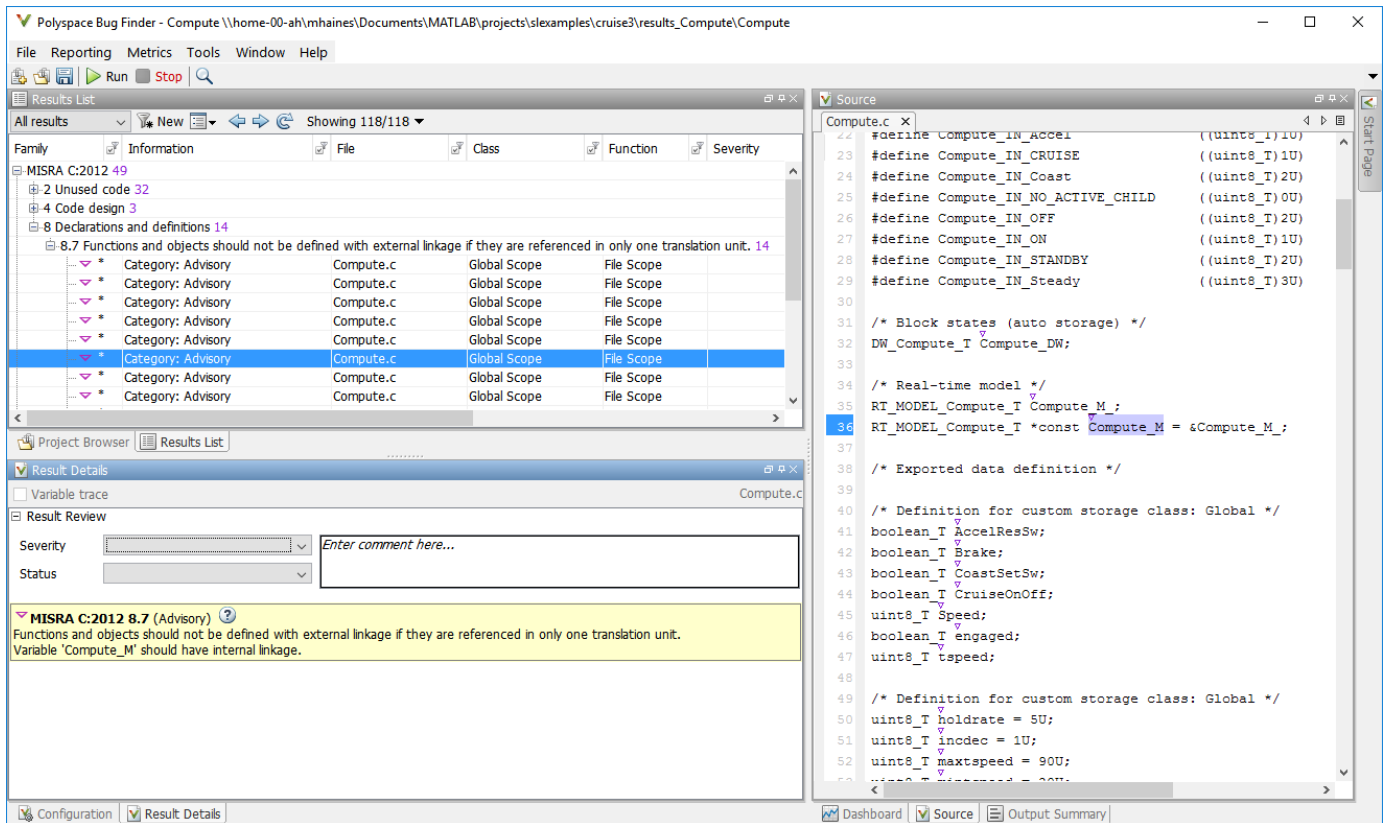
2   In your model, right-click Compute target speed and select **Polyspace > Options**.

3   Set the **Settings from** option to `Project configuration` to choose a subset of MISRA rules in the Polyspace configuration.

4   Click **Configure**.

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

6   Save and close the Polyspace configuration window.

7   Rerun the analysis with the new configuration.

The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, Polyspace found only two violations.
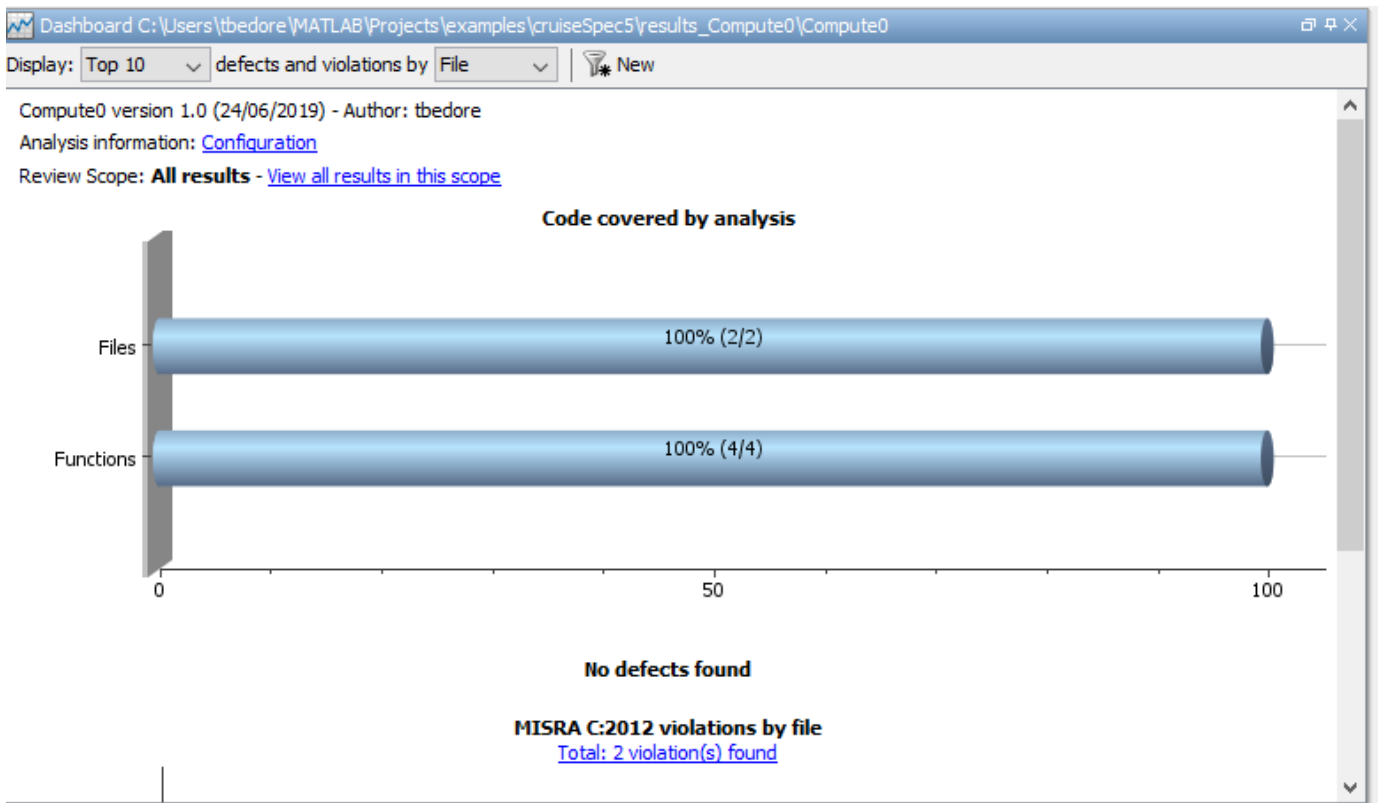
When you integrate this model with its parent model, you can add the rest of the MISRA C:2012 rules.

Generate Report

To demonstrate compliance with MISRA C:2012 and report on your generated code metrics, you must export your results. If you want to generate a report every time you run an analysis, see `Generate report` (Polyspace Bug Finder).

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**.

Test Code Against Model Using Software-in-the-Loop Testing

You previously showed that the model functionality meets its requirements by running test cases based on those requirements. Now run the same test cases on the generated code to show that the code produces equivalent results and fulfills the requirements. Then compare the code coverage to the model coverage to see the extent to which the tests exercised the generated code.

1   In MATLAB, in the project window, open the `tests` folder, then open `SILTests.mldatx`. The file opens in the Test Manager.

2 Review the test case. On the **Test Browser** pane, navigate to `SIL Equivalence Test Case`. This equivalence test case runs two simulations for the `simulinkCruiseErrorAndStandardsExample` model using a test harness.

- Simulation 1 is a model simulation in normal mode.
- Simulation 2 is a software-in-the-loop (SIL) simulation. For the SIL simulation, the test case runs the code generated from the model instead of running the model.

The equivalence test logs one output signal and compares the results from the simulations. The test case also collects coverage measurements for both simulations.

3 Run the equivalence test. Select the test case and click **Run**.

4 Review the results in the Test Manager. In the **Results and Artifacts** pane, select **SIL Equivalence Test Case** to see the test results. The test case passed and the results show that the code produced the same results as the model for this test case.

5 Expand the **Coverage Results** section of the results. The coverage measurements show the extent to which the test case exercised the model and the code. When you run multiple test cases, you can view aggregated coverage measurements in the results for the whole run. Use the coverage results to add tests and meet coverage requirements, as shown in "Perform Functional Testing and Analyze Test Coverage" on page 2-9.

You can also test the generated code on your target hardware by running a processor-in-the-loop (PIL) simulation. By adding a PIL simulation to your test cases, you can compare the test results and coverage results from your model to the results from the generated code as it runs on the target hardware. For more information, see "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" (Embedded Coder).

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" (Simulink Test)

Checking Systems Interactively

Check Model Compliance by Using the Model Advisor

Model Advisor Overview

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

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

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

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

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

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

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

While applying Model Advisor checks to your model increases the likelihood that your model does not violate certain modeling standards or guidelines, their application cannot guarantee that the system being developed will be safe or error-free. It is ultimately your responsibility to verify, using multiple methods, that the system being developed provides its intended functionality and does not include unintended functionality.

Model Advisor Checks Documentation

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

Product	Model Advisor Check Documentation
Simulink	"Simulink Checks"
Embedded Coder	"Embedded Coder Checks" (Embedded Coder)
AUTOSAR Blockset	"AUTOSAR Blockset Checks" (AUTOSAR Blockset)
Simulink Coder™	"Simulink Coder Checks" (Simulink Coder)
HDL Coder™	"HDL Code Advisor Checks" (HDL Coder)
Simulink Code Inspector™	"Simulink Code Inspector Checks" (Simulink Code Inspector)
Simulink Check	"DO-178C/DO-331 Checks" "IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Checks" "Model Checks for DO-254 Standard Compliance" on page 3-56 "High Integrity System Modeling Checks" "Model Advisor Checks for MAB and JMAAB Compliance" "MISRA C:2012 Checks" "Secure Coding Checks for CERT C, CWE, and ISO/IEC TS 17961 Standards" "Model Metrics"
Simulink Design Verifier	"Simulink Design Verifier Checks" (Simulink Design Verifier)
Simulink PLC Coder™	"PLC Model Advisor Checks" (Simulink PLC Coder)
Requirements Toolbox	"Requirements Consistency Checks" (Requirements Toolbox)
Simscape™	Documentation is available only in the Model Advisor. To review the documentation for the check, in the Model Advisor, right-click on the check title and select **What's This?**
Simulink Control Design™	"Simulink Control Design Checks" (Simulink Control Design)
IEC Certification Kit	"IEC Certification Kit Checks" (IEC Certification Kit)

Product	Model Advisor Check Documentation
DO Qualification Kit	"DO Qualification Kit Checks" (DO Qualification Kit)

Run Model Advisor Checks and Review Results

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

1   Open the Model Advisor example model `sldemo_mdladv`.

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

3   In the left pane of the Model Advisor, select the checks you want to run on your model:

   a   You can select the checks by using the **By Product** or **By Task** folders.

   • **Show By Product Folder** — Displays checks available for each product
   • **Show By Task Folder** — Displays checks related to specific tasks

   Checks with the icon ⬚ trigger an update of the model diagram.

   Checks with the icon ⬚ trigger an extensive analysis of the model. Checks that trigger extensive analysis of the model use additional analysis techniques, such as analysis with Simulink Design Verifier.

4   Click on the folder that contains the checks and, on the toolstrip, select **Run Checks** to execute the analysis. To run a single check, right-click the check in the folder and select **Run This Check**.

5   View the results on the Model Advisor User Interface. This table shows the common check status results; notice that different icons are used depending on the parameter set for **Check result when issues are flagged** in the Model Advisor Configuration Editor (requires a Simulink Check license). For more information about this parameter, see "Specify Parameters for Check Customization" on page 7-6.

Check Result Status	Icon	Description
Passed	✅	Model does not have any violations for the given check(s).
Failed	❌	Check has identified severe violations.
Warning	⚠️	Check has identified violations.
Justified	☑️	Check violations are justified.
Not Run	📄	Check not selected for Model Advisor analysis.
Incomplete	⛔	Check analysis is incomplete or check execution has resulted in exceptions.

6   Fix the warnings or failures as desired. For more information, see "Address Model Check Results".

7 Use the **Exclusions** tab to review checks that were marked for exclusion from the analysis.

8 View and save the report. For additional information, see "Save and View Model Advisor Check Reports" and "Generate Model Advisor Reports" on page 3-20.

See Also

Related Examples

- "Address Model Check Results"
- "Generate Model Advisor Reports" on page 3-20
- "Save and View Model Advisor Check Reports"
- "Find Model Advisor Check IDs"
- "Archive and View Results" on page 4-6
- "Exclude Blocks from the Model Advisor Check Analysis" on page 3-9
- "Check Model Compliance Using Edit-Time Checking" on page 3-6
- "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3

Check Model Compliance Using Edit-Time Checking

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

Configure Your Model to Use Edit-Time Checking

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

- In the **Debug** tab, select **Diagnostics > Edit-Time Errors and Warnings**.
- In the **Modeling** tab, select **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the check box for **Edit-Time Checks**.
- Enable edit-time checking through the command line using the `edittime.setAdvisorChecking` function.
- If you have an Embedded Coder or Simulink Coder license, you can use edit-time checks to evaluate your model for issues that are specific to code generation. To enable these checks, open the **Embedded Coder** app and select the **C/C++ Code Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the check box for **Edit-Time Checks**.

When edit-time checking is enabled, the Model Advisor highlights blocks in your model that violate Model Advisor checks.
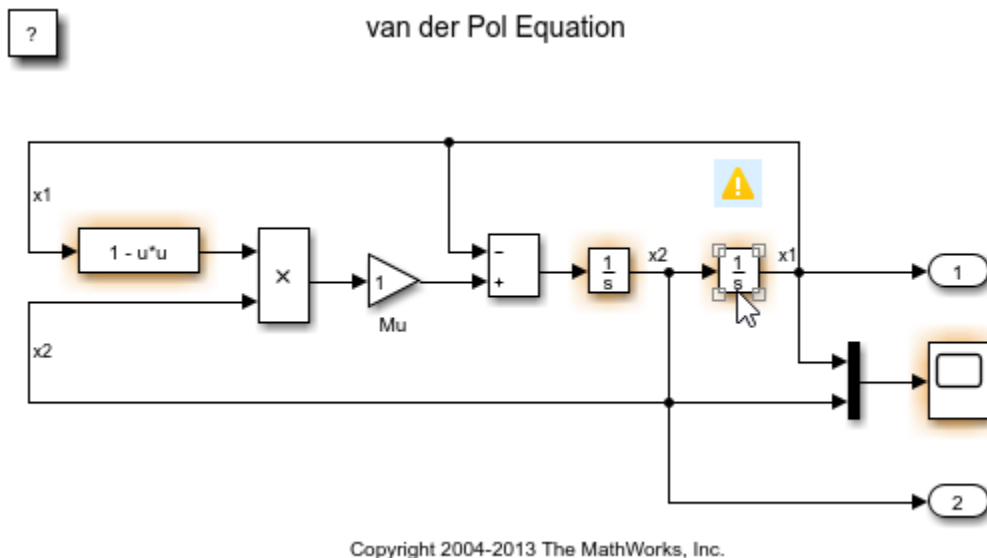
To review a check violation, click the error or warning icon above the highlighted block. A diagnostics window opens, which provides information about the modeling issue that violates the Model Advisor check. When a block violates multiple checks, you can use the diagnostics window to review issues.

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

- Review the cause and explore suggested options for fixing the issue, if any.

- Click the question mark to access detailed documentation about the violated Model Advisor check.
- Ignore the warning and add the block to the exclusion list for that check by clicking **Suppress**.

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

1 Open a model. For this example, at the command prompt, type: `sldemo_fuelsys`.

2 To enable the edit-time checking, in the **Modeling** tab, select **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the check box for **Edit-Time Checks**.

3 The Model Advisor highlights several blocks. Place your cursor over the warning of the **Throttle Angle Fault Switch** block to discover the issue.

Block name has incorrect characters

Caused by:
- Use only alphanumeric characters and underscores in block name.

Component: Model Advisor   ⊕ Suppress ⑦
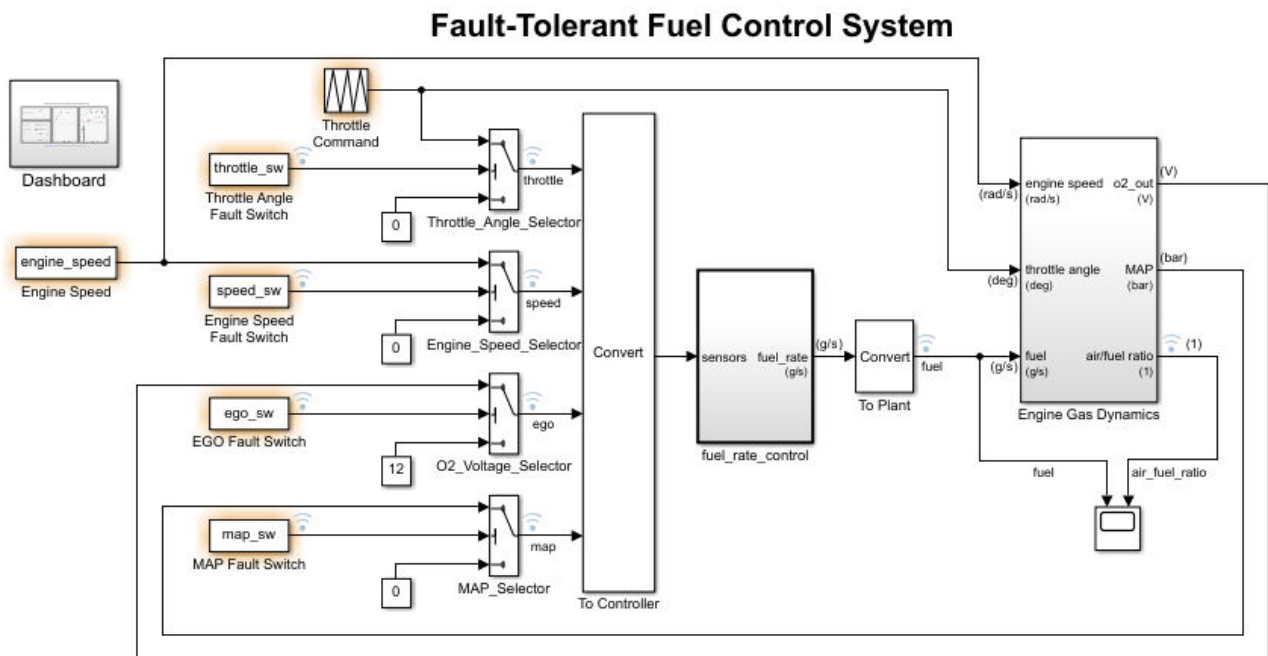
Fault-Tolerant Fuel Control System

Open the Dashboard subsystem to simulate any combination of sensor failures.   Copyright 1990-2017 The MathWorks, Inc.

4 Select the warning. The Model Advisor indicates that the block name has an incorrect character. Replace the space with an underscore character and the warning goes away.

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

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

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

1   In the Simulink editor, click the **Modeling** tab and select **Model Advisor > Configuration Editor**.

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

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

> **Note** When a check is included in multiple folders of your Model Advisor hierarchy, for edit-time checking, the Model Advisor prioritizes the check in your custom folder. If the check is not in your custom folder, priority goes to the check in the **By Task** folder, and finally to the check in your **By Product** folder.

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

See Also

Related Examples
- "Address Model Check Results"
- "Generate Model Advisor Reports" on page 3-20
- "Save and View Model Advisor Check Reports"
- "Find Model Advisor Check IDs"
- "Archive and View Results" on page 4-6
- "Exclude Blocks from the Model Advisor Check Analysis" on page 3-9

Exclude Blocks from the Model Advisor Check Analysis

Model Advisor Exclusion Overview

To save time during model development and verification, you can limit the scope of the 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. Alternatively, 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:

- Filter Identifier
- Filter Type
- Summary
- Check ID(s)
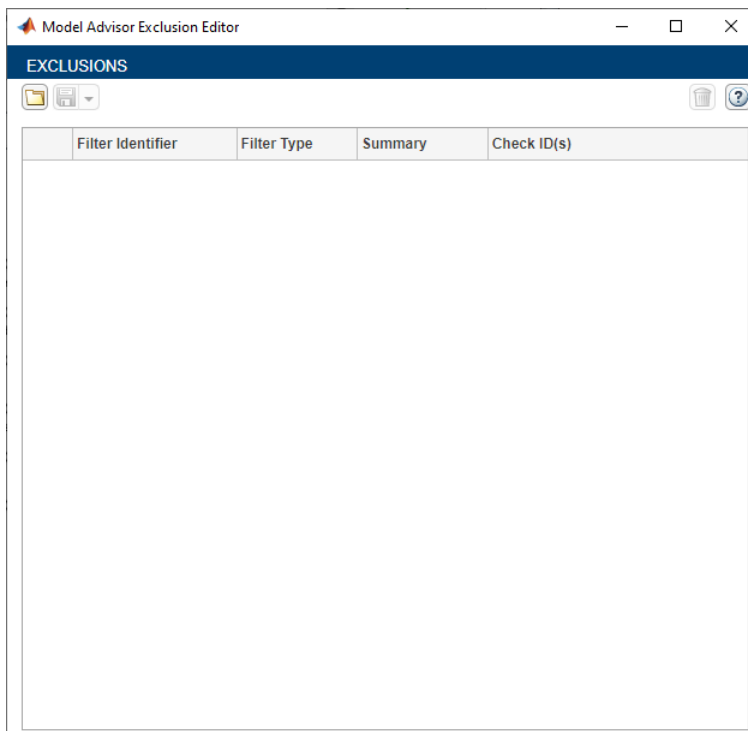
Field	Description
Filter Identifier	Name and path of the block or subsystem that is excluded. The path of the blocks are hyper-linked and clicking on them will highlight the concerned blocks on the model canvas.
Filter Type	Defines the type of the excluded item. Example: If the filter type is **Block**, on that particular Simulink block is excluded. If the type is **Subsystem**, then all the contents inside that subsystem are excluded.
Summary	Editable field to enter notes or the reason for exclusion. By default, this field defines whether a specific block is excluded or all blocks of a given type are excluded.
Check ID (s)	Names of the checks for which the block exclusion applies. Check Selector can be invoked from this cell by clicking on the edit (✎) button.

Note If you comment out blocks, they are excluded from both simulation and Model Advisor analysis.

Filter Types

Filter type defines the type of the excluded entity. Model Advisor Exclusion Editor currently supports exclusion of the following entities:

Filter Type	Description
Simulink	
Block	Exclude a Simulink block.
BlockType	Exclude all blocks of a type.
Subsystem	Exclude all blocks inside a Subsystem.
Library	Exclude all instances of a Library block.
MaskType	Exclude blocks or subsystem of mask type.
Stateflow	Exclude Stateflow blocks in Simulink.
Stateflow	
Chart	Exclude every entity inside a Stateflow Chart.
State	Exclude a Stateflow State.
Transition	Exclude a Stateflow Transition.
Junction	Exclude a Stateflow Junction.
GraphicalFunction	Exclude a Stateflow Graphical Function.
MATLABFunction	Exclude a Stateflow MATLAB Function.
SimulinkFunction	Exclude a Stateflow Simulink Function.
TruthTable	Exclude a Stateflow Truth Table.
SimulinkBasedState	Exclude a Stateflow Simulink based state.

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.

Task	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 of type <block_type> > All Checks**
Exclude the block from selected checks.	**a** **Exclude block only > Select Checks**. **b** In the Check Selector dialog box, select the checks. Click **OK**.
Exclude all blocks of this type from selected checks.	**a** **Exclude all blocks of type <block_type> > Select Checks**. **b** In the Check Selector dialog box, select the checks. Click **OK**.
Exclude the block from all failed checks. This option is available only after a Model Advisor analysis.	**Exclude block only > Only failed checks**
Exclude all blocks of this type from all failed checks. This option is available only after a Model Advisor analysis.	**Exclude all blocks of type <block_type> > Only failed checks**
Exclude the block from a failed check. This option is available only after a Model Advisor analysis.	**Exclude block only > <name of failed check>**
Exclude all blocks of this type from a failed check. This option is available only after a Model Advisor analysis.	**Exclude all blocks of type <block_type> > <name of failed check>**

2 In the Model Advisor Exclusion Editor dialog box, save the exclusions to the model or an exclusion file by using one of the processes below.

You can create as many Model Advisor exclusions as you want by right-clicking model blocks and selecting the options under **Model Advisor**.

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, click on the save icon(🖫). 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 save Model Advisor exclusions to the exclusion `.xml` file:

1 Open the Model Advisor Exclusion Editor.

2 click the drop-down next to the save icon( ).

3 Select **Save as** and enter the desired name for the exclusion file in the **File name** field.

4 Click **Save**.

Unless you specify a different folder, the Model Advisor saves exclusion files in the current folder.

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.

Check Selector

The Check Selector window allows you to select the checks to exclude for a specific block or all blocks of a specified type. Open the Check Selector by right-clicking a block and selecting either:

• **Model Advisor > Exclude block only > Select checks**
• **Model Advisor > Exclude all blocks of type <block_type> > Select checks**

In the Check Selector, you can use the search functionality to search for a check that needs to be excluded. Check Selector can be also invoked from the Model Advisor Exclusion Editor window by clicking on the edit ( ) button in the **Check Id(s)** column.
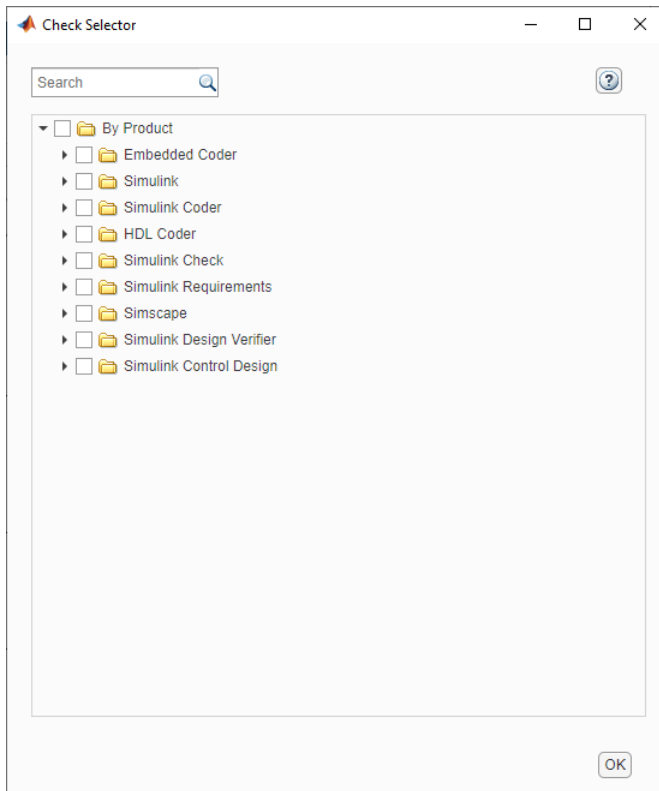
Review Model Advisor Exclusions

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.
- After the Model Advisor analysis, you can view exclusion information for individual checks in the report file.

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

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

Manage Exclusions

Load an Exclusion File

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

1   In the Model Advisor Exclusion Editor dialog box, click **Load** icon.
2   Navigate to the exclusion file that you want to use with your model. Select **Open**.
3   In the Model Advisor Exclusion Editor dialog box, click **OK** to associate the exclusion file with your model.

Remove an Exclusion

1   In the Model Advisor Exclusion Editor dialog box, select the exclusions that you want to remove.
2   Click the **Delete Exclusion Row** button ().

Add Summary 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 **Summary** field for the exclusion.
2   Delete the existing text.
3   Add the summary for excluding this object.

Compatibility Considerations after R2020b

In R2020b, the format in which exclusions are stored changed. In releases after R2020b, Model Advisor can read format used in previous releases, and convert it to new format. Conversion happens only once. The model is updated with the new file path, which Exclusion editor subsequently uses.

When you open a exclusion file created in R2020a or earlier, the files and models are updated based on whether the exclusion file was saved in the model or as a separate file, following are the sequence of actions that will be performed in each case:

Exclusion File Saved Inside the Model

1   The original exclusion file is read and written to a new file.
2   The new file is saved in the model SLX file when you save the model.

These changes are done automatically, and without notification that the file has been updated.

Exclusion File Is Saved in a Separate Exclusion File

1   The old file is read and, you can choose to overwrite the existing exclusion file or save the exclusion file with a new name in the same location.
2   The model updates the new file path when you save the model.

The Exclusion Editor reports that a change has taken place.

Programmatically Change Model Advisor Exclusions

The Model Advisor Exclusion Editor can now be used with the following functions:

Task	Function
Add a new exclusion in Model Advisor.	`Advisor.addExclusion`
Remove exclusions from Model Advisor.	`Advisor.removeExclusion`
Clear all exclusions from Model Advisor.	`Advisor.clearExclusion`
Get exclusions for a model or a filter.	`Advisor.getExclusion`
Save exclusions to the default option or to a new file.	`Advisor.saveExclusion`
Load default exclusions stored inside the model or according to the path settings.	`Advisor.loadExclusion`

The Model Advisor Exclusion file path is tracked by a model parameter called **MAModelFilterFile**. Use the `set_param` API to update this parameter.

```
set_param('<model name>','MAModelFilterFile', '<new_file_path>');
```

After the model is saved and reopened, the changes are reflected in the Exclusion Editor. If `<new_file_path>` is an empty character vector, Model Advisor Exclusion Editor assumes the file is stored inside the model SLX file.

See Also

Related Examples

- "Exclude Blocks From Custom Checks" on page 6-57
- "Run Model Advisor Checks and Review Results" on page 3-4
- "Address Model Check Results"
- "Generate Model Advisor Reports" on page 3-20
- "Save and View Model Advisor Check Reports"
- "Find Model Advisor Check IDs"
- "Archive and View Results" on page 4-6
- "Limit Model Checks by Excluding Gain and Outport Blocks"
- "Exclude Blocks from Edit Time Checking"

More About

- "Check Your Model Using the Model Advisor"

Justify Violated Blocks from the Model Advisor Check Analysis

Model Advisor Justification Overview

Justifications allows you to add a rationale for violations observed from the Model Advisor analysis to exist. The justified violations are displayed in the report with a new status `Justified`.

You can add a Model Advisor Justification to any Simulink elements.

Create Model Advisor Justifications

You can justify a violation when you are building a Simulink diagram (Edit-time violations), or when you complete the Model Advisor check analysis.

Justify violations post Model Advisor check analysis

To justify the Model Advisor violations displayed post check analysis, use these steps:

1 From the Check Selector section, select the violated check(s).
2 Click **Justify** icon from the toolstrip.
3 Enter the rationale for justification in the **Justifications** field on the **Result Inspector** tab.

4   Click **Add Justification**.

Justify edit-time check violations

To justify the edit-time violations displayed during model creation, use these steps:

1   On the Simulink canvas, hover over a violated block.

2   Click on the warning icon displayed above the violated block.

Violation summary is displayed along with the title of the violated check.
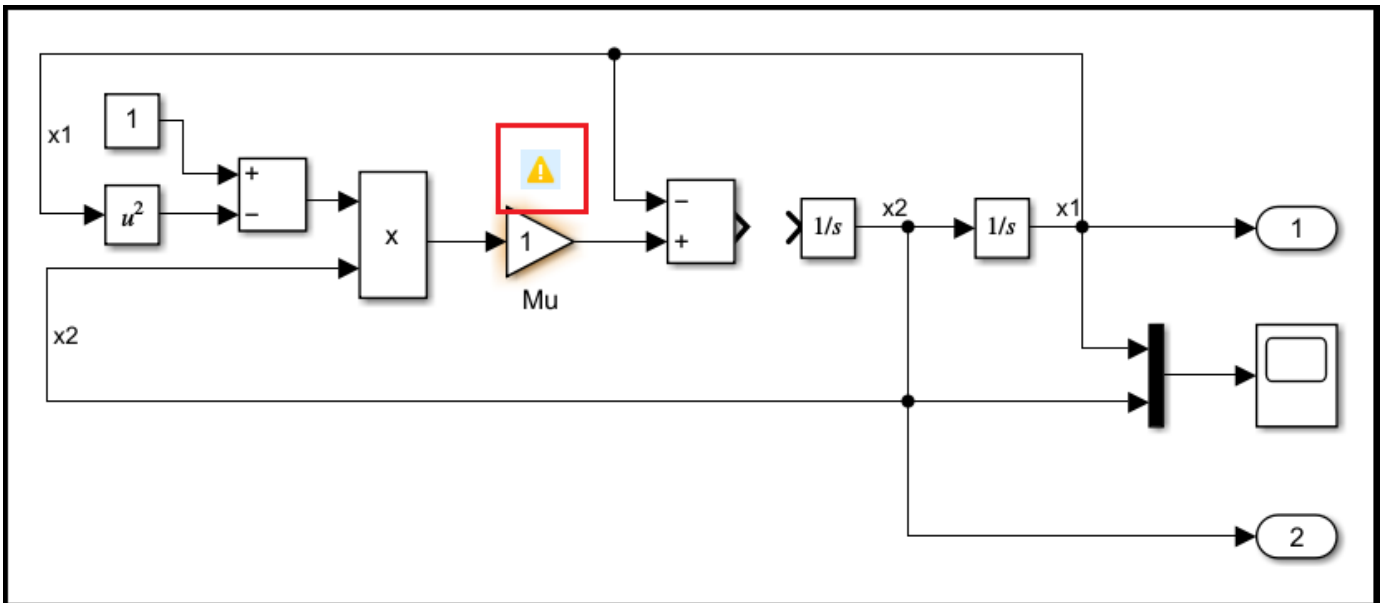
3   Click **Suppress**. A description field to enter rationale is displayed.
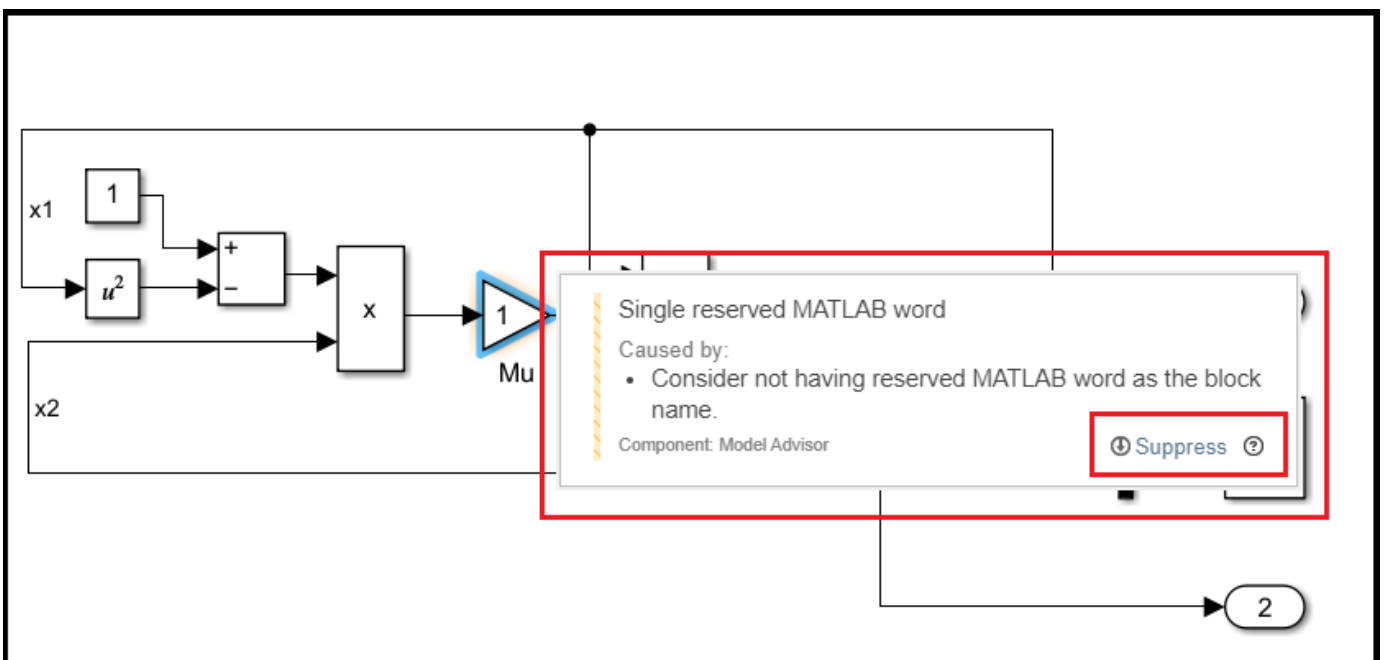
4   Enter rationale for the justification.

5   Click **Add Justification**.

Manage Justifications

You can view and manage the justifications created when using edit time checking, and post Model Advisor check analysis using the Model Advisor UI. You can perform save, edit, delete actions for the justifications.

Note To view the Justifications created during edit-time in the Model Advisor UI, you need to run Model Advisor checks from the user interface.

Save Justifications

Justifications are auto-saved in the directory where the model is located. The justification file is by default named with syntax `<modelname>_justifications.json`.

Note The path in which a `justifications.json` is saved is displayed over a banner on the simulink canvas when you a first justification is created.

Load Justifications

Justifications created and saved for a model are automatically loaded when the model is opened for the next time. Justifications are saved in a `json` file in the directory where the model is located. The justification file is by default named with syntax `<modelname>_justifications.json`.

Edit a Justification

To edit the rationale for the justifications:

1   From the check selector tab, select the justified check.

2   Open the **Result Inspector** tab.

3   Click **Edit**. Field to edit the existing rationale is enabled.

4   Edit the rationale.

5   Click **Add Justification**.

Delete a Justification

You can delete justifications and set the justified violations to the existing **Error** or **Warning** statuses respectively.

To delete a justification:

1   From the check selector tab, select the justified check.

2   Open the **Result Inspector** tab.

3   Click **Delete**.

See Also

Related Examples

- "Generate Model Advisor Reports" on page 3-20
- "Run Model Advisor Checks and Review Results" on page 3-4
- "Exclude Blocks From Custom Checks" on page 6-57
- "Address Model Check Results"
- "Save and View Model Advisor Check Reports"
- "Find Model Advisor Check IDs"
- "Archive and View Results" on page 4-6
- "Limit Model Checks by Excluding Gain and Outport Blocks"
- "Exclude Blocks from Edit Time Checking"

More About

- "Check Your Model Using the Model Advisor"

Generate Model Advisor Reports

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

The beginning of the Model Advisor reports contain the:

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

Generate Results Report After Executing Model Advisor Checks

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

1   In the left pane of the Model Advisor, select the checks you want to run. Click on the folder that contains the checks and, from the toolstrip, click **Run Checks**.

2   When complete, from the toolstrip, click **Report**.

3   In the **Save Report** dialog box:

- Enter the path to the folder where you want to generate the report.
- Provide a file name.
- Click **Save** to generate a report in HTML format.

4   Use can change the **File format** to PDF, or WORD using the drop-down options of the **Report** button.

5   The Model Advisor generates the report and saves it to the designated location.

Modify Template for Model Advisor Check Results Report

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

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

Template Field	In generated report, displays
ModelName	Model name
SimulinkVersion	Simulink version
SystemName	System name
TreatAsMdlRef	Whether or not model is treated as a referenced model

Template Field	In generated report, displays
ModelVersion	Model version
CurrentRun	Model Advisor analysis time stamp
PassCount	Number of checks that pass
JustifiedCount	Number of checks that are justified
IncompleteCount	Number of checks that fail to run to completion
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 From the Model Advisor toolstrip, click **Report** drop-down, and select Template File.

5 In the **Select Template for Report** dialog box, enter the path to the folder where your custom template is placed. in our case, the path is C:/work/ma_format/mytemplate.dotx.

6 Click **OK**.

7 From the toolstrip, click **Report** drop-down, and select PDF. The Model Advisor generates the report in PDF format with the custom header.

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

ModelAdvisor.summaryReport | viewReport

Related Examples

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

Transform Model to Variant System

You can use the Model Transformer tool to improve model componentization by replacing qualifying modeling patterns with Variant Source and Variant Subsystem, Variant Model blocks. The Model Transformer reports the qualifying modeling patterns. You choose which modeling patterns the tool replaces with a Variant Source block or Variant Subsystem block.

The Model Transformer can perform these transformations:

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

For the Model Transformer tool to perform the transformation, the control input to Multiport Switch or Switch blocks and the inputs to If or Switch Case blocks must be either of the following:

- A Constant block in which the **Constant value** parameter is a `Simulink.Parameter` object of scalar type.
- Constant blocks in which the **Constant value** parameters are `Simulink.Parameter` objects of scalar type and some other combination of blocks that form a supported MATLAB expression. The MATLAB expressions in "Types of Operators in Variant Blocks for Different Activation Times" are supported except for bitwise operations.

Example Model

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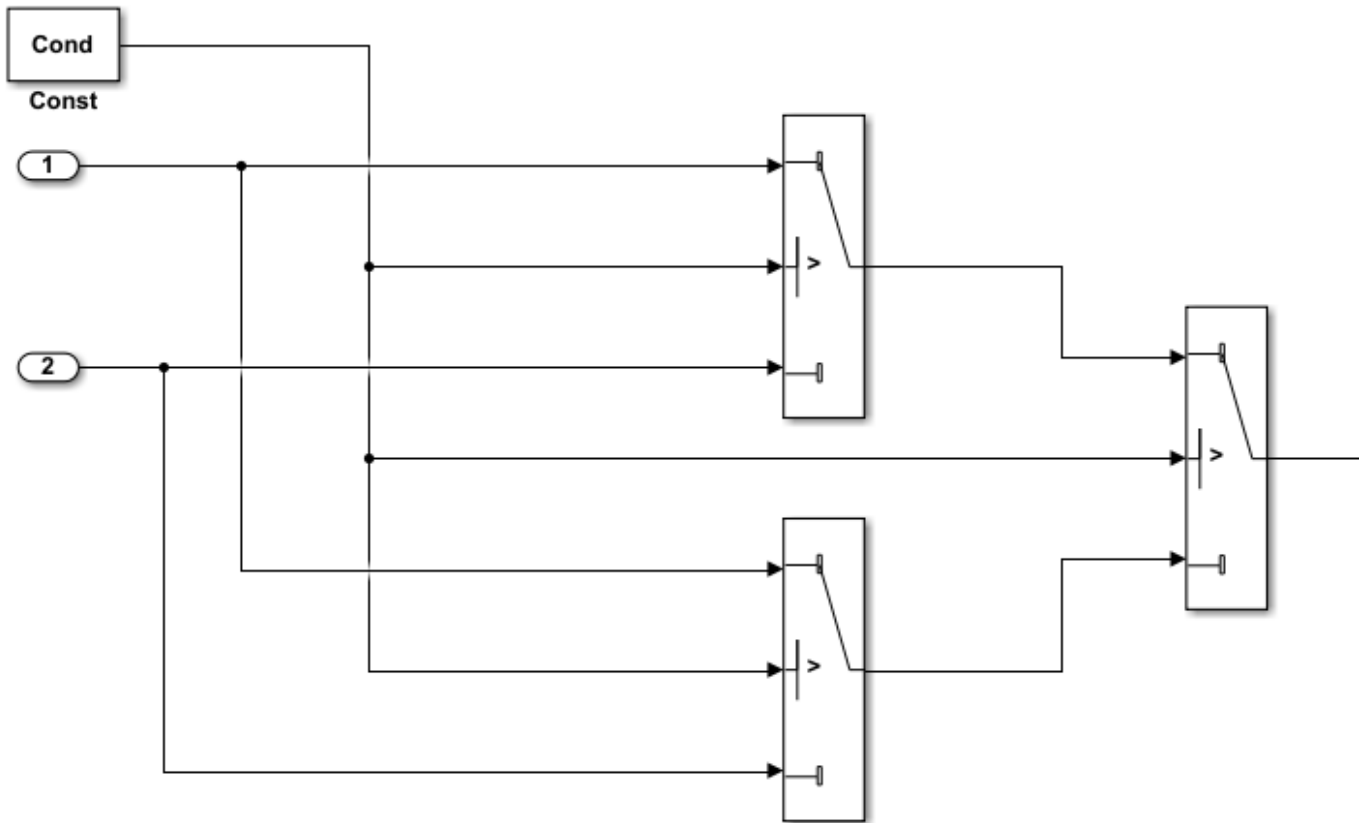
1. Open the model. In the Command Window, type `rtwdemo_controlflow_opt`.

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

3. Repeat step 2 for the Switch blocks `Switch1`, `Switch2`, and `Switch3`.

4. Save the model to your working folder.

Perform Variant Transform on Example Model

1. In the **Apps** tab, Open the Model Transformer by selecting **Model Transformer**. Or, in the Command Window, type:
   `mdltransformer('rtwdemo_controlflow_opt')`

2. Select the check "Replace Modeling Patterns with Variant Blocks".
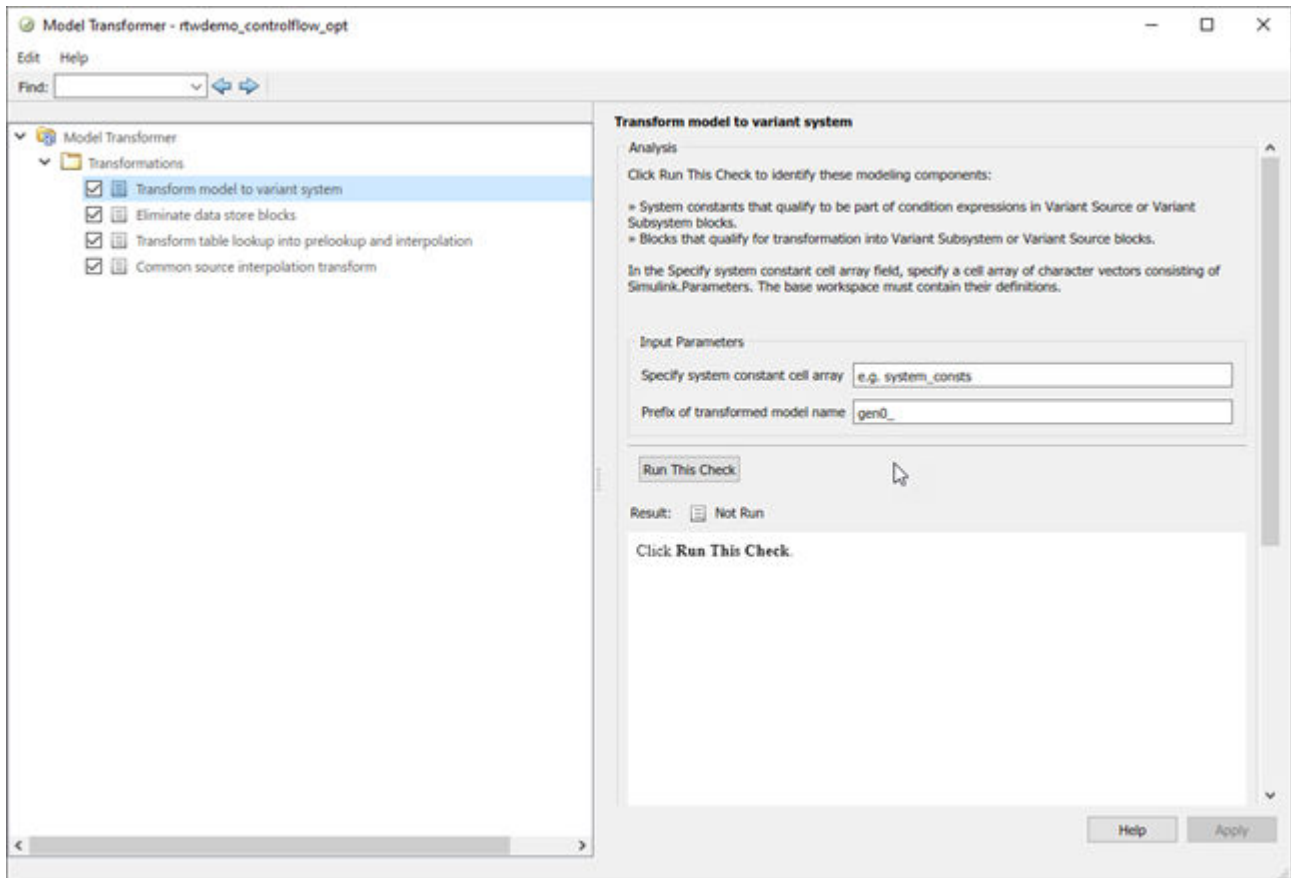
3   In the **Specify system constant cell array** field, you can specify a cell array of character vectors consisting of `Simulink.Parameters`. The base workspace must contain their definitions.

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 Model block that references a protected model.

  - A model contains a Variant Source block with the **Variant activation time** parameter set to `update diagram`.

- After you run one or more tasks, you cannot rerun the tasks because the **Run this Check** 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"

Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks

You can use the Model Transformer tool to refactor a modeling pattern to improve the efficiency of generated code. The Model Transformer identifies and merges multiple Interpolation Using Prelookup blocks that have same input signals connected from the outputs of Prelookup blocks into a single Interpolation Using Prelookup block.

The Model Transformer works if the properties of Interpolation Using Prelookup blocks are same except for **Table data**. Reducing the number of Interpolation Using Prelookup blocks in a model reduces the number of variable assignments in the code, which improves the efficiency of the generated code. You can use the Model Transformer app or programmatic commands to refactor the model.

The Model Transformer can replace multiple Interpolation Using Prelookup that:

- Have the same input signals connected to Prelookup blocks with the same index and fraction parameters
- Have the output signals connected to the same Multiport Switch block
- Have the same breakpoint specification, values, and data types
- Have the same algorithm parameters
- Have the same data type for fraction parameters

Merge Interpolation Using Prelookup Blocks Using the Model Transformer App

This example shows how to use the Model Transformer app and APIs to identify redundant Interpolation Using Prelookup blocks, and then refactor the model.
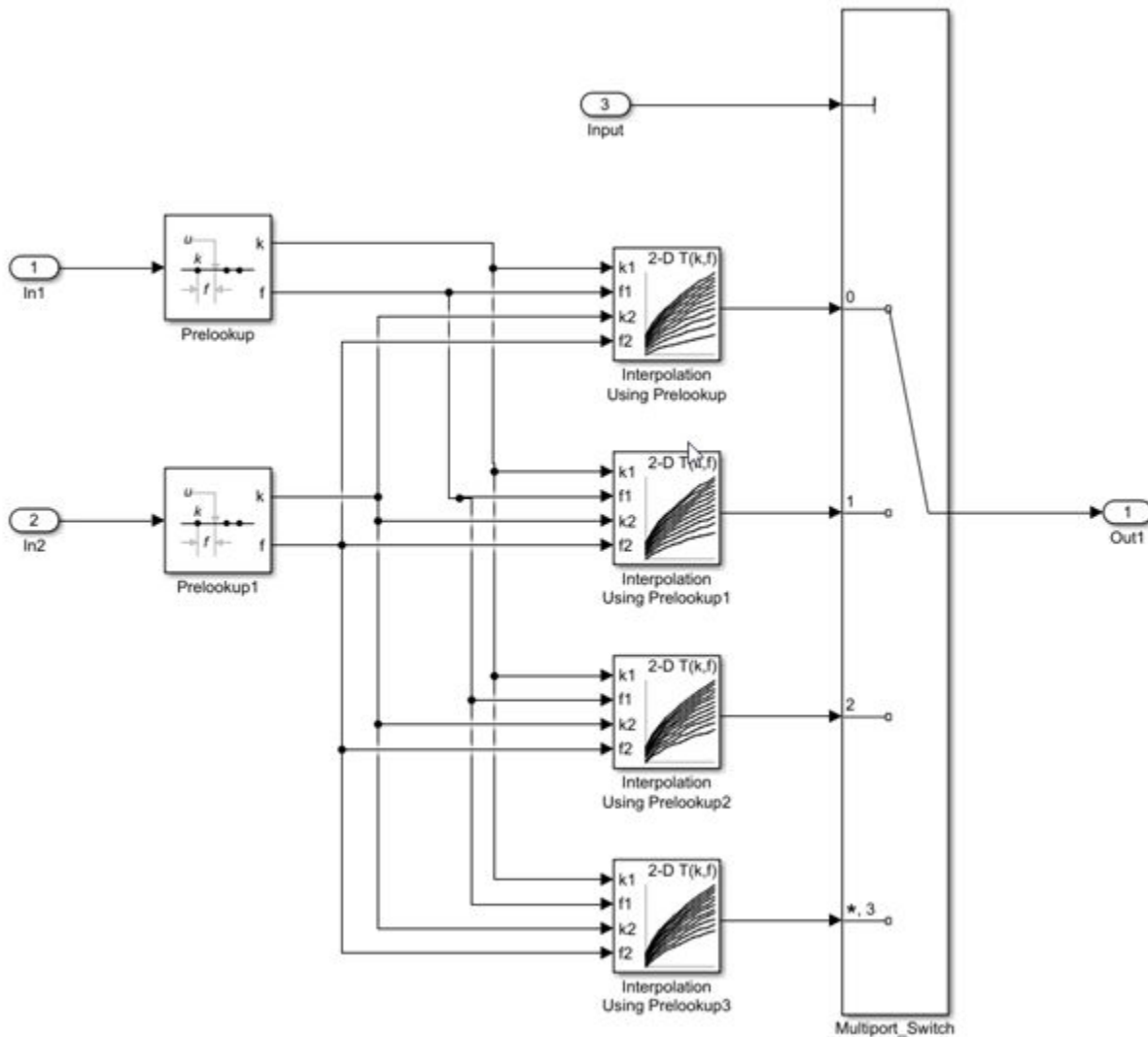
The model `mInterpolationOptim` uses Prelookup blocks to input signals to several Interpolation Using Prelookup blocks. The output of these Interpolation blocks are connected to a Multiport Switch block.

In this example, you identify Interpolation Using Prelookup blocks that qualify for transformation and replace them with a single Interpolation Using Prelookup block and Constant blocks connected to Multiport Switch block.

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

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

2   Save the model to your working folder.

3   On the **Apps** tab, click **Model Transformer**.

4   In the **Transformations** folder, select the "Replace Interpolation Using Prelookup Blocks" check.

5   Select the **Skip Interpolation-ND blocks in libraries from this transformation** option to avoid replacing Interpolation Using Prelookup blocks that are linked to a library.

6   In the **Prefix of transformed model** field, specify a prefix for the refactored model.

7   Click **Run This Check**. The top **Result** table contains hyperlinks to the Interpolation Using Prelookup blocks and the corresponding Multiport Switch block port indices.

8   Clear the check boxes under **Candidate Groups** for the groups that you do not want to transform.

9   Click **Refactor Model**. The **Result** table contains a hyperlink to the new model. The table contains hyperlinks to the shared Interpolation Using Prelookup blocks and corresponding Multiport Switch block ports. The tool also creates an `m2m_mInterpolationOptim` folder that contains the new `gen_mInterpolationOptim.slx` model.
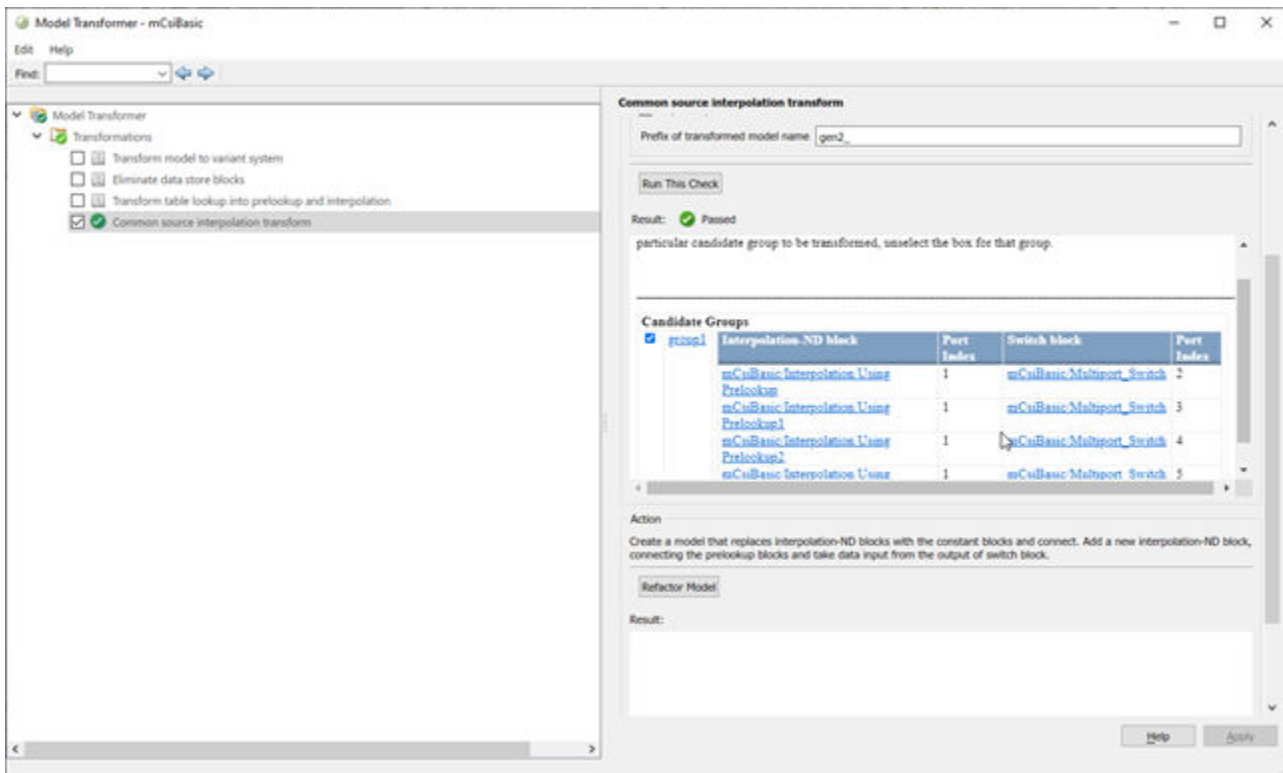
The two Prelookup blocks `Prelookup` and `Prelookup1` in the `gen_mInterpolationOptim.slx` model connect to the single Interpolation Using Prelookup block and the Constant blocks are connected to the Multiport Switch block port to give the **Table data** as input.
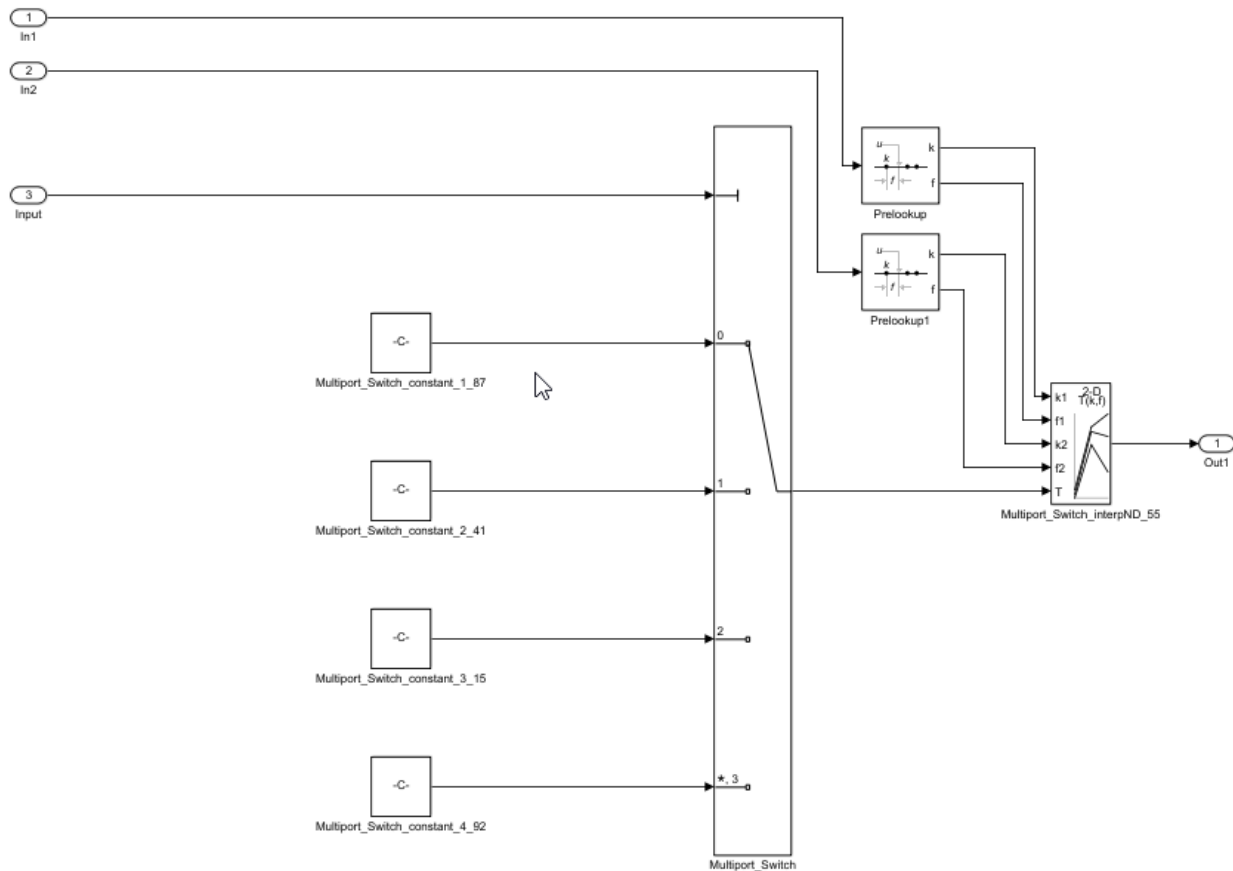
Merge Interpolation Using Prelookup Blocks Programmatically

To use the Model Transformer programmatically, use:

- `Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates` to identify candidates for transformation
- `Simulink.ModelTransform.CommonSourceInterpolation.refactorModel` to refactor the model

1   Save the model in the current working directory.

    addpath(fullfile(docroot,'toolbox','simulink','examples'))
    mInterpolationOptim

2   To identify candidates qualified for transformation, use the function `Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates` to create the object `transformResults`.

    transformResults = Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates('mInterpolationOptim')

    transformResults =

      Results with properties:

```
      Candidates: [1×1 struct]
```

The `transformResults` object has one property, `Candidates`, that is a structure with two fields, `InterpolationPorts` and `SwitchPorts`.

```
transformResults.Candidates =

  struct with fields:

        InterpolationPorts: [4×1 struct]
              SwitchPorts: [4×1 struct]
```

3 View the `InterpolationPorts` field.

```
transformResults.Candidates.InterpolationPorts =


  4×1 struct array with fields:

    Block
    Port
```

The `InterpolationPorts` field consists of two arrays, `Block` and `Port`. Similarly, the `SwitchPorts` has the same properties.

4 Convert the `Candidates.InterpolationPorts` and `Candidates.SwitchPorts` fields to tables.

```
struct2table(transformResults.Candidates.InterpolationPorts)
struct2table(transformResults.Candidates.SwitchPorts)

ans =
  4×2 table
                          Block                          Port
    _____    ____

    {'mInterpolationOptim/Interpolation↵Using Prelookup' }     0
    {'mInterpolationOptim/Interpolation↵Using Prelookup1'}     0
    {'mInterpolationOptim/Interpolation↵Using Prelookup2'}     0
    {'mInterpolationOptim/Interpolation↵Using Prelookup3'}     0


ans =
  4×2 table
                 Block                      Port
    _____    ____

    {'mInterpolationOptim/Multiport_Switch'}     1
    {'mInterpolationOptim/Multiport_Switch'}     2
    {'mInterpolationOptim/Multiport_Switch'}     3
    {'mInterpolationOptim/Multiport_Switch'}     4
```

Use the `SwitchPorts` to see which Interpolation Using Prelookup blocks are connected to which Multiport Switch block port.

5 To refactor the model, use the function `Simulink.ModelTransform.CommonSourceInterpolation.refactorModel`. This function uses the object `transformResults` from `identifyCandidate` function.

```
refactorResults = Simulink.ModelTransform.CommonSourceInterpolation.refactorModel(transformResults)

refactorResults =
```

```
RefactorResults with properties:

         ModelName: 'mInterpolationOptim'
    ModelDirectory: ''
  TraceabilityInfo: [4×1 containers.Map]
```

The `ModelName` and `ModelDirectory` properties of the `refactorResults` object list the name and location of the refactored model. `TraceabilityInfo` is a `containers.Map` object that lists the block tracing information.

Conditions and Limitations

The Model Transformer cannot replace Interpolation Using Prelookup blocks if:

- The Interpolation Using Prelookup blocks are in commented-out regions or inactive variants.
- The Interpolation Using Prelookup blocks are masked.
- The Model Transformer app does not replace Interpolation Using Prelookup blocks across the boundaries of atomic subsystems, referenced models, or library-linked blocks.

See Also

Related Examples

- "Refactor Models"
- "Using the Prelookup and Interpolation Blocks"

Enable Component Reuse by Using Clone Detection

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

Exact Clones and Similar Clones

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

Exact clones and similar clones can have these differences:

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

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

Specify Where to Detect Clones

The Clone Detector app supports two options for detecting clones in a model. You can search for clones in a subsystem or anywhere across the model using clone detection Settings.

- Subsystem clones: Identifies clones only in a subsystems.
- Clones across model: Identifies clones across the model.

Identify Exact and Similar Clones

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

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

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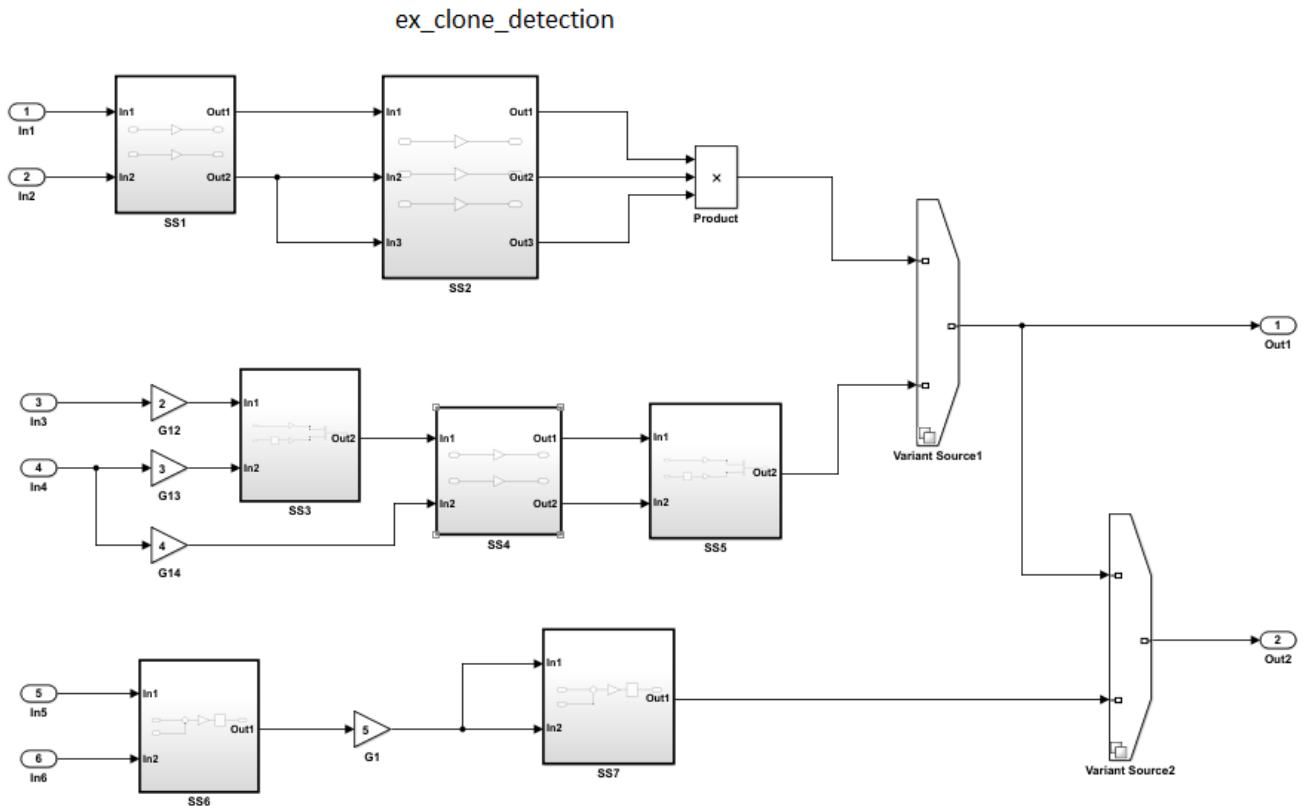
ex_clone_detection

Copyright 2017 The MathWorks Inc.

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

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

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

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

Set Up panes for Clone Detection

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

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

Set the Parameters for Clone Detection

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

- Select **Ignore differences in > Signal Names** to identify and classify clones when the signal names are different.
- Select **Ignore differences in > Block Properties** to identify and classify clones when the block properties are different. For more information about block properties, see "Specify Block Properties".
- Click **Replace Exact Clones With Subsystem References** to find and replace exact clones with subsystem reference blocks.
- Click **Exclude Components** to access the **Exclude model references**, **Exclude library Links**, and **Exclude inactive and commented out regions** options. Enabling the **Exclude inactive and commented out regions** option identifies variable number clones because of Variant Source block in the model. For more information, see "Exclude Components from Clone Detection". Enabling the **Exclude model references** and **Exclude library Links** options will lead to identification of fewer clones, depending on the model.
- Click **Match Patterns with Libraries** and select an external library to look for clones. For more information, see "Identify and Replace Clones in Model Libraries" on page 3-38.
- The **Maximum number of unmatched block parameters** is 50 by default. This represents the number of parameters that can vary among subsystems and still be classified as similar clones. You may reduce this number to identify and classify fewer similar clones. Setting the value to zero, will identify only exact clones.
- Click **Detect Clones Across Model** to enable detect clones anywhere across the model. You can choose the values of **Minimum Region Size** and **Minimum Clone Group Size** to detect the clones with these matching blocks. The default size is set to 2.

Identify Subsystem Clones in the Model

1   To find clones within the model, click on the subsystem that you want to analyse. In the **Detect** section, the selected subsystem name appears under **Find Clones in System** tab. Use the pin to remember the selection.

2   Click **Find Clones** to identify clones.
3   The color of the subsystems changes to reflect the similar and exact clones identified. The red highlighting represents exact clones and the different shades of blue highlighting represent similar clones.
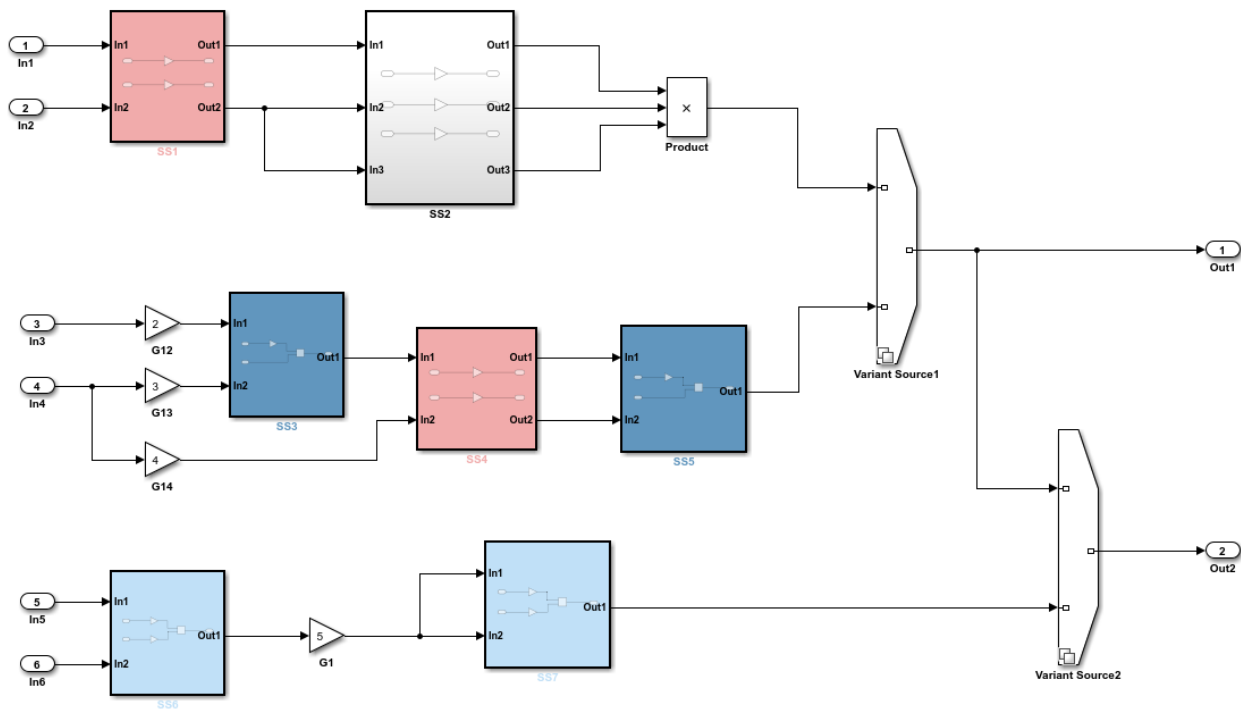
Clone Detector app creates a backup folder in the working directory. The backup folder name has the prefix m2m_*<model name>*. It saves the clones data in a MAT-file. You can also find the backup of the original model in this folder after refactoring the model to replace clones with links to library blocks.

Analyze the Clone Detection Results

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

1. In the **Clone Detection Results and Actions** panel, on the **Clone Results** tab, a list of clone groups are displayed.

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

3. In the **Clone Detection Results and Actions** pane, click the **Logs** tab. Click the hyperlink on the **Logs** pane.

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

4. Click the **Model Hierarchy** tab. Click the hyperlinks to highlight the particular subsystems in the model. To go back to highlighting all clones, on the **Clone Results** tab, click the **Highlight all clones**.

5. On the **Clone Results** tab, expand Similar Clone Group 1 and click the **View Parameter Difference** hyperlink.

6 On the **Detected Clone Properties** panel, click the `ex_clone_detection/SS5/G9` hyperlink, which opens the gain block `G9` in the subsystem `SS5`, where you can access the parameter that are different from the baseline subsystem.

7 Change value of the gain parameter from A to B and click **Find Clones**. This will reclassify `Similar Clone Group 1` to `Exact Clone Group 2` because you resolved the difference in the subsystems and converted it into an exact clone.

8 Under the **Refactor Benefits** panel, you can consider the percentage of different types of clones present.

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

Replace Clones

1 You may use the default library name or change the name of the library file and its location on the **Clone Results** tab before replacing the clones.

2 Click **Replace Clones**.

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

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

Identify and Replace Clones in Model Libraries

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

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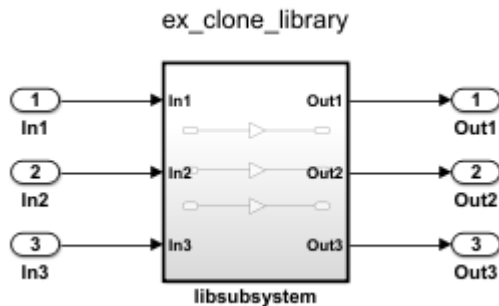

2 Click **Settings > Match Patterns with Libraries** and select `ex_clone_library.slx`. Then click **Find Clones**.

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

3 Click **Replace Clones**.

The model is refactored with the exact clones replaced.

Check the Equivalency of the Model

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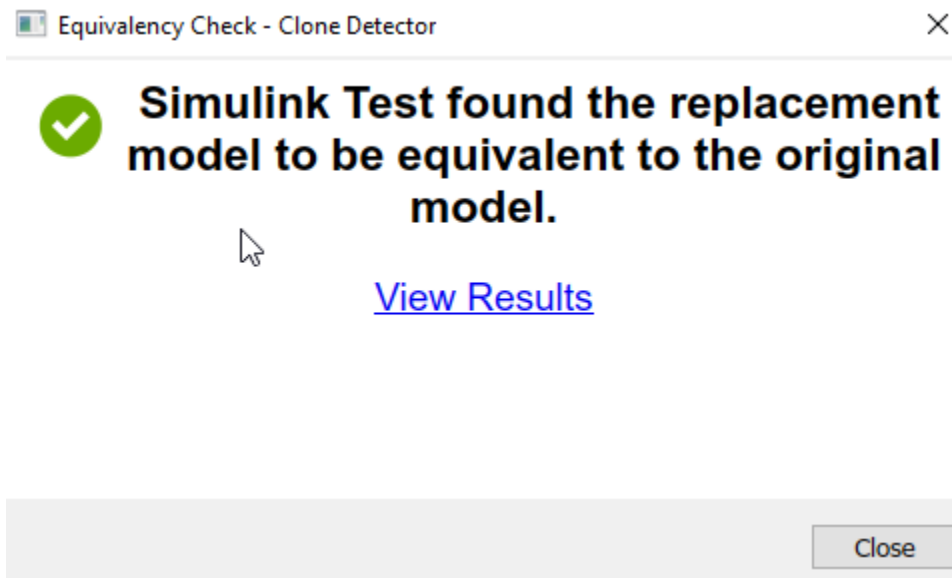
See Also

Related Examples

- "Custom Libraries"
- "Generate Reusable Code from Library Subsystems Shared Across Models" (Simulink Coder)
- Clone Detector
- "Replace Exact Clones with Subsystem Reference" on page 3-86

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.

Data Store Memory

Data Store Read

Constant

Add

Data Store Write

Data Store Read1

Out1

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.

Data Store Memory1

In1

Bias

Data Store Write1

Data Store Read2

Out2

Copyright 2017 The MathWorks, Inc.

Replace Data Store Blocks

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

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

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

2   Save the model to your working folder.

3   On the **Apps** tab, click **Model Transformer**. Alternatively, on the MATLAB command prompt, type this command:

    ```
    mdltransformer('ex_data_store_elimination')
    ```

    .

4   In the **Transformations** folder, select the **Eliminate data store blocks** check.

5   In the **Prefix of refactored model** field, specify a prefix for the refactored model.

6   Click the **Run This Check** button. The top **Result** table contains hyperlinks to the Data Store Memory blocks and the corresponding Data Store Read and Data Store Write blocks that qualify for elimination.

7   Click the **Refactor Model** button. The bottom **Result** table contains a hyperlink to the new model. The tool creates an `m2m_ex_data_store_replacement` folder. This folder contains the `gen_ex_data_store_replacement.slx` model.
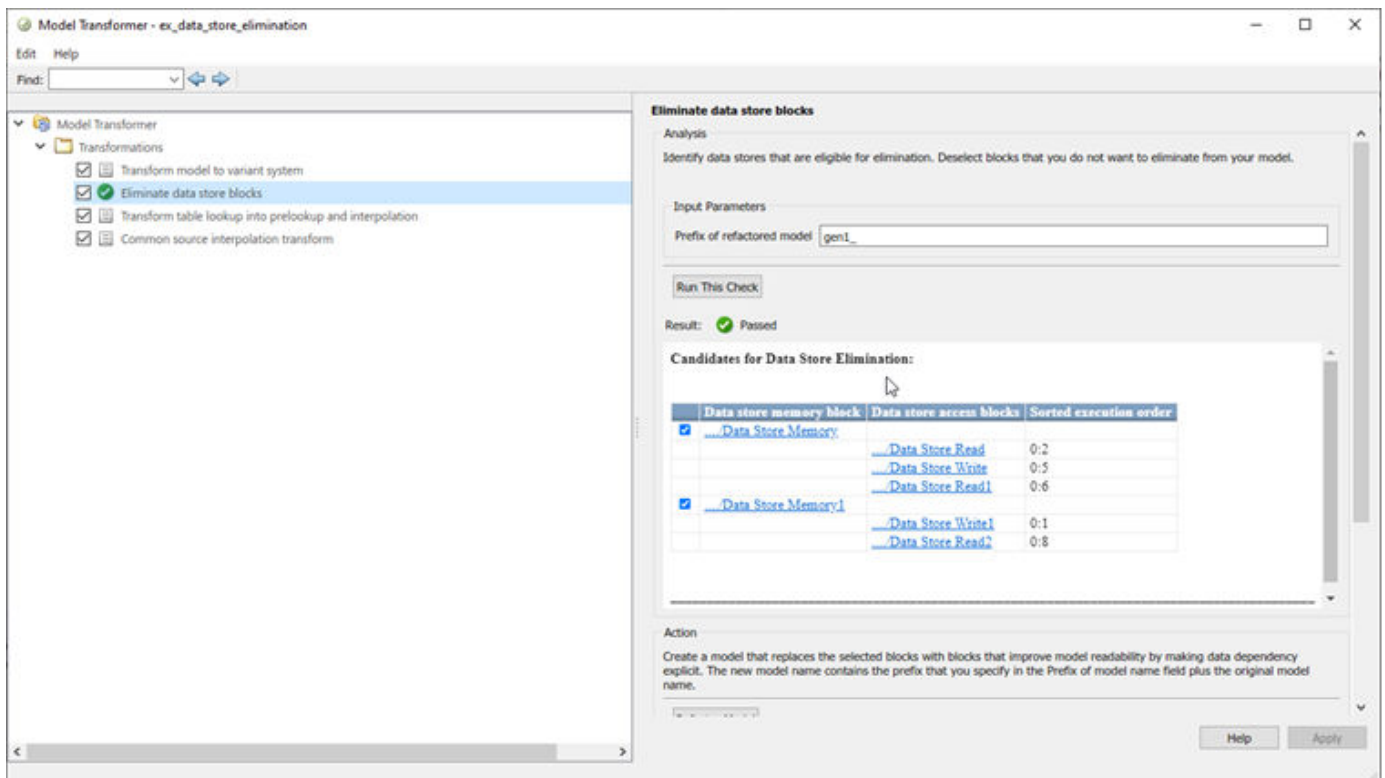
For local data store A, `gen_ex_bus_struct_in_code.slx` contains a Delay block in place of the Data Store Write block and a direct signal connection in place of the Data Store Read block. For local data store B, `gen_ex_bus_struct_in_code.slx` contains a direct signal connection from the Bias block to `Out2`.

Limitations

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

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

See Also

Related Examples

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

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

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

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

- The same source drives the Lookup Table blocks.
- The Lookup Table blocks share the same breakpoint specification, values, and data types.
- The breakpoint input port of the Lookup Tables is the connected to the same input source.
- The Lookup Table blocks share the same algorithm parameters in the block parameters dialog box.
- The Lookup Table blocks share the same data type for fractions parameters in the block parameters dialog box.

Example Model

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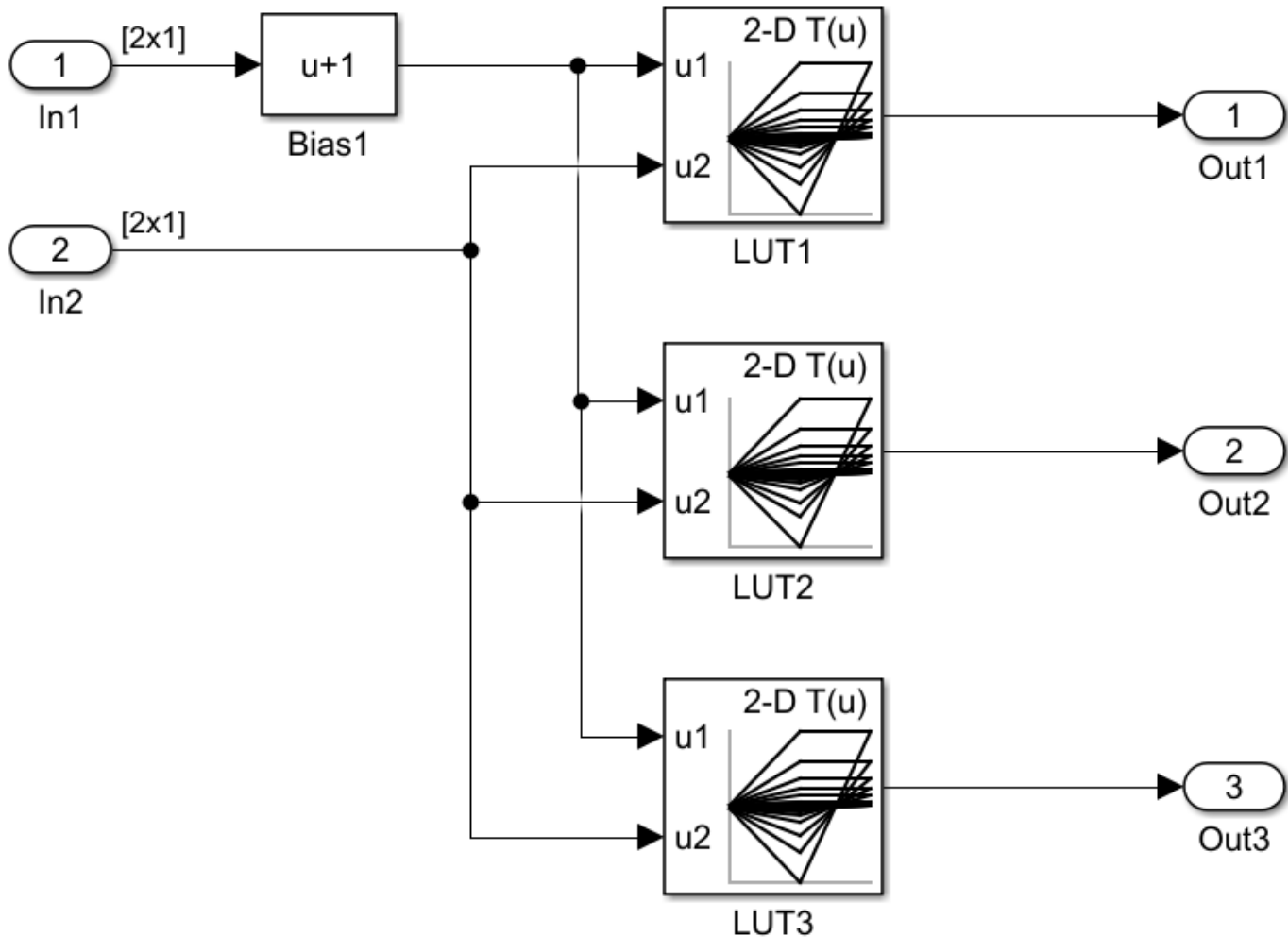
Merge Prelookup Operation

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

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

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

```
mLutOptim
```

2 Save the model to your working folder.

3 On the **Apps** tab, click **Model Transformer**.

4 In the **Transformations** folder, select the "Transform Table Lookup Blocks to Prelookup and Interpolation Using Prelookup Blocks" 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 and interpolation
    - ☑ 📄 Common source interpolation transform

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_`

[Run This Check]

Result: 📄   Not Run

Click **Run This Check**.

Action

[Help]   [Apply]

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.
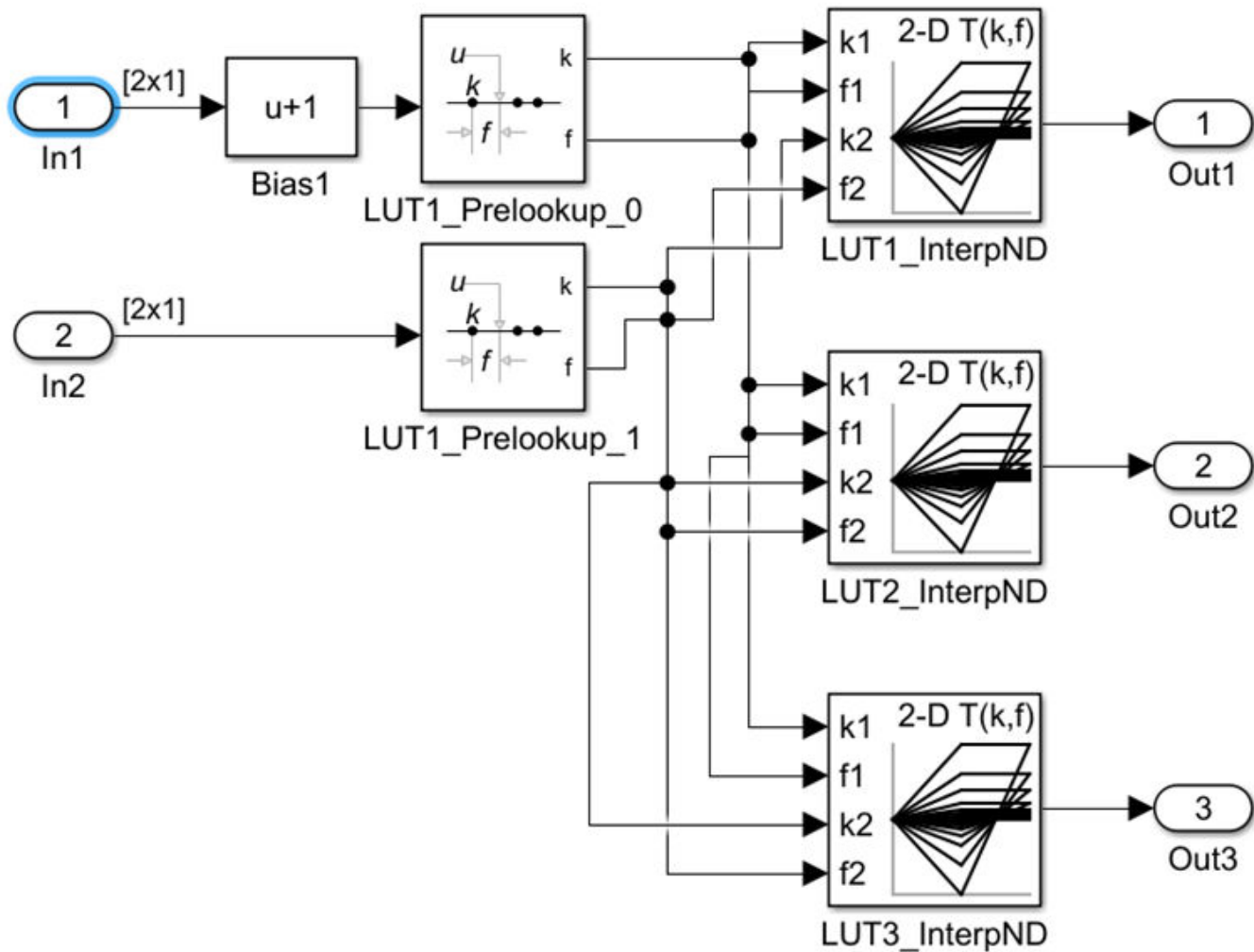
Conditions and Limitations

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 Blocks to Prelookup and Interpolation Using Prelookup Blocks"

Model Checks for DO-178C/DO-331 Standard Compliance

You can check that your model or subsystem complies with selected aspects of the DO-178C safety standard by running the Model Advisor.

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

The table lists the DO-178C/DO-331 checks.

Subfolder	Model Advisor Checks
N/A	"Display model version information"
MISRA	"Check for missing error ports for AUTOSAR receiver interfaces"
	"Check for unsupported block names"
	"Check bus object names that are used as bus element names"
	"Check for equality and inequality operations on floating-point values"
	"Check for bitwise operations on signed integers"
	"Check integer word length"
	"Check for missing const qualifiers in model functions"
	"Check for recursive function calls"
	"Check for switch case expressions without a default case"
Bug Reports	"Display bug reports for DO Qualification Kit" (DO Qualification Kit)
	"Display bug reports for Simulink Check" (DO Qualification Kit)
	"Display bug reports for Simulink Coverage" (DO Qualification Kit)
	"Display bug reports for Requirements Toolbox" (DO Qualification Kit)
	"Display bug reports for Simulink Code Inspector" (DO Qualification Kit)
	"Display bug reports for Simulink Report Generator" (DO Qualification Kit)
	"Display bug reports for Simulink" (DO Qualification Kit)
	"Display bug reports for Simulink Test" (DO Qualification Kit)
	"Display bug reports for Simulink Design Verifier" (DO Qualification Kit)
	"Display bug reports for Embedded Coder" (DO Qualification Kit)
	"Display bug reports for Polyspace Bug Finder" (DO Qualification Kit)
	"Display bug reports for Polyspace Bug Finder Server" (DO Qualification Kit)
	"Display bug reports for Polyspace Code Prover" (DO Qualification Kit)
	"Display bug reports for Polyspace Code Prover Server" (DO Qualification Kit)

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

Model Checks for High Integrity Systems Modeling

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

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

- **By Task > Modeling Standards for DO-178C/DO-331 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 61508 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 62304 > High-Integrity Systems**
- **By Task > Modeling Standards for EN 50128/EN 50657 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 26262 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 25119 > High-Integrity Systems**

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

This 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".

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 usage of variant blocks	"hisl_0023: Verification of variant blocks"
Check for root Inports with missing properties	"hisl_0024: Inport interface definition"
Check usage of Relational Operator blocks	"hisl_0017: Usage of blocks that compute relational operators (2)"
Check relational comparisons on floating-point signals	"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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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 > Code Generation > Optimization > Data initialization"
Check safety-related optimization settings for data type conversions	"hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values"
Check safety-related optimization settings for division arithmetic exceptions	"hisl_0054: Configuration Parameters > Code Generation > 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 identifier settings	"hisl_0049: Configuration Parameters > Code Generation > Identifiers"
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 remainder and reciprocal operations"
Check usage of Math Function blocks (log and log10 functions)	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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 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 MATLAB Function interfaces with inherited properties	"himl_0002: Strong data typing at MATLAB function boundaries"
Check MATLAB Function metrics	"himl_0003: Complexity of user-defined MATLAB Functions"
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 > Data Validity > 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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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 > Data Validity > 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 > Data Validity > Merge blocks"
Check safety-related diagnostic settings for Stateflow	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
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"
Check usage of logical operators and functions in MATLAB Function blocks	"himl_0010: MATLAB code with logical operators and functions"
Check type and size of conditional expressions	"himl_0011: Data type and size of condition expressions"
Check naming of ports in Stateflow charts	"hisf_0016: Stateflow port names"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check scoping of Stateflow data objects	"hisf_0017: Stateflow data object scoping"
Check usage of Gain blocks	"hisl_0066: Usage of Gain blocks"
"Check for divide-by-zero calculations"	"hisl_0067: Protect against divide-by-zero calculations"
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"
Check safety-related optimization settings for specified minimum and maximum values	"hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values"
"Check usage of remainder and reciprocal operations"	"hisl_0002: Usage of remainder and reciprocal operations"
"Check usage of square root operations"	"hisl_0003: Usage of square root operations"
"Check usage of log and log10 operations"	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
"Check usage of Reciprocal Sqrt blocks"	"hisl_0028: Usage of Reciprocal Square Root blocks"
"Check safety-related settings for hardware implementation"	"hisl_0071: Configuration Parameters > Hardware Implementation >Inconsistent hardware implementation settings"
"Check usage of recursions"	hisf_0004: Protect against recursive function calls to improve code compliance
"Check MATLAB functions not supported for code generation"	"himl_0012: Usage of MATLAB functions for code generation"
"Metrics for generated code complexity"	"himl_0013: Limitation of built-in MATLAB Function complexity"
"Check for parameter tunability ignored for referenced models"	"hisl_0072: Usage of tunable parameters for referenced models"
"Check usage of bit-shift operations"	"hisl_0073: Usage of bit-shift operations"
"Check safety-related diagnostic settings for variants"	"hisl_0074: Configuration Parameters > Diagnostics > Modeling issues related to variants"
"Check for disabled and parameterized library links"	"hisl_0075: Usage of library links"
"Check for unreachable and dead code"	"hisl_0101: Avoid operations that result in dead logic to improve code compliance"

See Also

Related Examples

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

Model Checks for DO-254 Standard Compliance

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

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

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

The table below lists the DO-254 checks.

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

Model Checks for High Integrity Systems Modeling

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

To check compliance with High Integrity System Model standards, run the high-integrity checks from **By Task > Modeling Standards for DO-254 > High-Integrity Systems**

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

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
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 variant blocks"
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 Checks	Applicable High-Integrity System Modeling Guidelines
Check relational comparisons on floating-point signals	"hisl_0016: Usage of blocks that compute relational operators"
Check usage of Logical Operator blocks	"hisl_0018: Usage of Logical Operator block"
Check sample time-dependent blocks	"hisl_0007: Usage of For Iterator or While Iterator subsystems"
Check safety-related block reduction optimization settings	"hisl_0046: Configuration Parameters > Simulation Target > Block reduction"
Check usage of Abs blocks	"hisl_0001: Usage of Abs block"
Check usage of Assignment blocks	"hisl_0029: Usage of Assignment 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 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 unary operators	"hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance"
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 parameters	"hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters"
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 for model initialization	"hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization"
Check safety-related diagnostic settings for model referencing	"hisl_0310: Configuration Parameters > Diagnostics > Model Referencing"

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for saving	"hisl_0036: Configuration Parameters > Diagnostics > Saving"
Check safety-related diagnostic settings for Stateflow	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
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 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 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 data type of loop control variables	"hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance"

HDL Code Advisor Checks

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

- Model configuration settings
- Ports and Subsystem settings

- Blocks and block settings
- Native Floating Point support
- Industry standard guidelines

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

HDL Code Advisor Checks	Description
"Check for infinite and continuous sample time sources" (HDL Coder)	Check source blocks with continuous sample time.
"Check for unsupported blocks" (HDL Coder)	Check for unsupported blocks for HDL code generation.
"Check for large matrix operations" (HDL Coder)	Check for large matrix operations.
"Identify unconnected lines, input ports, and output ports"	Check for unconnected lines or ports.
"Identify disabled library links"	Search model for disabled library links.
"Identify unresolved library links"	Search the model for unresolved library links, where the specified library block cannot be found.
"Check for MATLAB Function block settings" (HDL Coder)	Check HDL compatible settings for MATLAB Function blocks.
"Check for Stateflow chart settings" (HDL Coder)	Check HDL compatible settings for Stateflow Chart blocks.
"Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition"	Identify Delay, Unit Delay, or Zero-Order Hold blocks that are used for rate transition. Replace these blocks with actual Rate Transition blocks.
"Check for unsupported storage class for signal objects" (HDL Coder)	Check whether signal object storage class is `'ExportedGlobal'` or `'ImportedExtern'` or `'ImportedExternPointer'`
"Check file extension" (HDL Coder)	Check file extensions of VHDL files containing entities.
"Check naming conventions" (HDL Coder)	Check standard keywords used by EDA tools.
"Check top-level subsystem/port names" (HDL Coder)	Check top-level module/entity and port names.
"Check module/entity names" (HDL Coder)	Check module/entity names.
"Check signal and port names" (HDL Coder)	Check signal and port name lengths.
"Check package file names" (HDL Coder)	Check file name containing packages.
"Check generics" (HDL Coder)	Check generics at top-level subsystem.
"Check clock, reset, and enable signals" (HDL Coder)	Check naming convention for clock, reset, and enable signals.
"Check architecture name" (HDL Coder)	Check VHDL architecture name in the generated HDL code.
"Check entity and architecture" (HDL Coder)	Check whether the VHDL entity and architecture are described in the same file.

HDL Code Advisor Checks	Description
"Check clock settings" (HDL Coder)	Check constraints on clock signals.
"Check for global reset setting for Xilinx and Altera devices" (HDL Coder)	Check asynchronous reset setting for Altera® devices and synchronous reset setting for Xilinx® devices.
"Check inline configurations setting" (HDL Coder)	Check whether you have `InlineConfigurations` enabled.
"Check algebraic loops" (HDL Coder)	Check model for algebraic loops.
"Check for visualization settings" (HDL Coder)	Check model for display settings: port data types and sample time color coding.
"Check delay balancing setting" (HDL Coder)	Check Balance Delays is enabled.
"Check for model parameters suited for HDL code generation" (HDL Coder)	Check for model parameters set up for HDL code generation.
"Check for double datatypes in the model with Native Floating Point" (HDL Coder)	Check for `double` data types in the model.
"Check for Data Type Conversion blocks with incompatible settings" (HDL Coder)	Check conversion mode of Data Type Conversion blocks.
"Check for HDL Reciprocal block usage" (HDL Coder)	Check HDL Reciprocal blocks are not using floating point types.
"Check for Relational Operator block usage" (HDL Coder)	Check Relational Operator blocks which use floating point types have boolean outputs.
"Check for unsupported blocks with Native Floating Point" (HDL Coder)	Check for unsupported blocks with native floating-point.
"Check for blocks that have nonzero output latency" (HDL Coder)	Check for blocks that have nonzero output latency with native floating-point.
"Check blocks with nonzero ulp error" (HDL Coder)	Check for blocks that have nonzero ulp error with native floating-point.
"Check for single datatypes in the model" (HDL Coder)	Check for `single` data types in the model.
"Check initial conditions of enabled and triggered subsystems" (HDL Coder)	Check for initial condition of enabled and triggered subsystems.
"Check for invalid top level subsystem" (HDL Coder)	Check for subsystems that cannot be at the top level for HDL code generation.

See Also

Related Examples

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

Model Checks for MAB and JMAAB Compliance

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

Accessing the MAB and JMAAB Model Advisor Checks

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

The MAB checks are available from these Model Advisor folders:

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

The JMAAB checks are available from these Model Advisor folders:

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

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

Modeling Guidelines and Model Advisor Checks for MAB and JMAAB

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

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

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

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

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

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

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

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

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

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0081: Inport and Outport block icon display	jc_0081	"Check display for port blocks"
na_0011: Scope of Goto and From blocks	na_0011	"Check scope of From and Goto blocks"
jc_0161: Definition of Data Store Memory blocks	jc_0161	"Check for usage of Data Store Memory blocks"
jc_0141: Usage of the Switch blocks	jc_0141	"Check usage of Switch blocks"
jc_0650: Block input/output data type with switching function	jc_0650	"Check input and output datatype for Switch blocks"
jc_0630: Usage of Multiport Switch blocks	jc_0630	"Check settings for data ports in Multiport Switch blocks"
na_0020: Number of inputs to variant subsystems	na_0020	"Check for missing ports in Variant Subsystems"
na_0036: Default variant	na_0036	"Check use of default variants"
na_0037: Use of single variable for variant condition	na_0037	"Check use of single variable variant conditionals"
db_0123: Stateflow port names	No corresponding JMAAB guideline	"Check for names of Stateflow ports and associated signals"
db_0125: Stateflow local data	db_0125	"Check definition of Stateflow data"
db_0126: Defining Stateflow events	db_0126	"Check definition of Stateflow events"
jc_0701: Usable number for first index	jc_0701	"Check usable number for first index"
jc_0712: Execution timing for default transition path	jc_0712	"Check execution timing for default transition path"
jc_0722: Local data definition in parallel states	jc_0722	"Check scope of data in parallel states"
jc_0797: Unconnected transitions / states / connective junctions	jc_0797	"Check for unconnected objects in Stateflow Charts"
db_0137: States in state machines	db_0137	"Check for state in state machines"
jc_0721: Usage of parallel states	jc_0721	"Check usage of parallel states"
db_0129: Stateflow transition appearance	db_0129	"Check for Stateflow transition appearance"
jc_0531: Default transition	jc_0531	"Check default transition placement in Stateflow charts"

Model Checks for MAB and JMAAB Compliance

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0723: Prohibited direct transition from external state to child state	jc_0723	"Check usage of transitions to external states"
jc_0751: Backtracking prevention in state transition	jc_0751	"Check for unexpected backtracking in state transitions"
jc_0760: Starting point of internal transition	jc_0760	"Check starting point of internal transition in Stateflow"
jc_0763: Usage of multiple internal transitions	jc_0763	"Check usage of internal transitions in Stateflow states"
jc_0762: Prohibition of state action and flow chart combination	jc_0762	"Check prohibited combination of state action and flow chart"
db_0132: Transitions in flow charts	db_0132	"Check transitions in Stateflow Flow charts"
jc_0773: Unconditional transition of a flow chart	jc_0773	"Check usage of unconditional transitions in flow charts"
jc_0775: Terminating junctions in flow charts	jc_0775	"Check terminal junctions in Stateflow"
jc_0738: Usage of Stateflow comments	jc_0738	"Check usage of Stateflow comments"
jc_0790: Action language of Chart block	jc_0790	"Check Stateflow chart action language"
jc_0702: Use of named Stateflow parameters and constants	jc_0702	"Check usage of numeric literals in Stateflow"
jm_0011: Pointers in Stateflow	jm_0011	"Check for pointers in Stateflow charts"
jc_0491: Reuse of Stateflow data	jc_0491	*Not checkable*
jm_0012: Usage restrictions of events and broadcasting events	jm_0012	"Check for usage of events and broadcasting events in Stateflow charts"
jc_0733: Order of state action types	jc_0733	"Check order of state action types"
jc_0734: Number of state action types	jc_0734	"Check repetition of Action types"
jc_0740: Limitation on use of exit state action	jc_0740	"Check if state action type 'exit' is used in the model"
jc_0741: Timing to update data used in state chart transition conditions	jc_0741	"Check updates to variables used in state transition conditions"

3-67

MAB Modeling Guideline (Version 5.0)	Corresponding JMAAB Modeling Guideline (Version 5.1)	Model Advisor Check
jc_0772: Execution order and transition conditions of transition lines	jc_0772	"Check usage of transition conditions in Stateflow transitions"
jc_0753: Condition actions and transition actions in Stateflow	jc_0753	"Check condition actions and transition actions in Stateflow"
jc_0711: Division in Stateflow	jc_0711	*Not checkable*
db_0127: Limitation on MATLAB commands in Stateflow blocks	db_0127	"Check for MATLAB expressions in Stateflow charts"
jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow	jc_0481	"Check usage of floating-point expressions in Stateflow charts"
na_0001: Standard usage of Stateflow operators	na_0001	"Check Stateflow operators"
jc_0655: Prohibition of logical value comparison in Stateflow	jc_0655	"Check prohibited comparison operation of logical type signals"
jc_0451: Use of unary minus on unsigned integers	jc_0451	"Check usage of unary minus operations in Stateflow charts"
jc_0802: Prohibited use of implicit type casting in Stateflow	jc_0802	"Check for implicit type casting in Stateflow"
jc_0803: Passing values to library functions	jc_0803	*Not checkable*
jc_0732: Distinction between state names, data names, and event names	jc_0732	"Check uniqueness of Stateflow State and Data names"
jc_0730: Unique state name in Stateflow blocks	jc_0730	"Check uniqueness of State names"
jc_0731: State name format	jc_0731	"Check usage of State names"
jc_0501: Format of entries in a State block	jc_0501	"Check entry formatting in State blocks in Stateflow charts"
jc_0736: Uniform indentations in Stateflow blocks	jc_0736	"Check indentation of code in Stateflow states"
jc_0739: Describing text inside states	jc_0739	"Check for usage of text inside states"
jc_0770: Position of transition label	jc_0770	"Check placement of Label String in Transitions"
jc_0771: Comment position in transition labels	jc_0771	"Check position of comments in transition labels"

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

See Also

- "Run Model Advisor Checks and Review Results"

Model Checks for High Integrity Systems Modeling

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

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

- **By Task > Modeling Standards for DO-178C/DO-331 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 61508 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 62304 > High-Integrity Systems**
- **By Task > Modeling Standards for EN 50128/EN 50657 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 26262 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 25119 > High-Integrity Systems**

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

This 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".

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 usage of variant blocks	"hisl_0023: Verification of variant blocks"
Check for root Inports with missing properties	"hisl_0024: Inport interface definition"
Check usage of Relational Operator blocks	"hisl_0017: Usage of blocks that compute relational operators (2)"
Check relational comparisons on floating-point signals	"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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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 > Code Generation > Optimization > Data initialization"
Check safety-related optimization settings for data type conversions	"hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values"
Check safety-related optimization settings for division arithmetic exceptions	"hisl_0054: Configuration Parameters > Code Generation > 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 identifier settings	"hisl_0049: Configuration Parameters > Code Generation > Identifiers"
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 remainder and reciprocal operations"
Check usage of Math Function blocks (log and log10 functions)	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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 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 MATLAB Function interfaces with inherited properties	"himl_0002: Strong data typing at MATLAB function boundaries"
Check MATLAB Function metrics	"himl_0003: Complexity of user-defined MATLAB Functions"
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 > Data Validity > 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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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 > Data Validity > 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 > Data Validity > Merge blocks"
Check safety-related diagnostic settings for Stateflow	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
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"
Check usage of logical operators and functions in MATLAB Function blocks	"himl_0010: MATLAB code with logical operators and functions"
Check type and size of conditional expressions	"himl_0011: Data type and size of condition expressions"
Check naming of ports in Stateflow charts	"hisf_0016: Stateflow port names"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check scoping of Stateflow data objects	"hisf_0017: Stateflow data object scoping"
Check usage of Gain blocks	"hisl_0066: Usage of Gain blocks"
"Check for divide-by-zero calculations"	"hisl_0067: Protect against divide-by-zero calculations"
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"
Check safety-related optimization settings for specified minimum and maximum values	"hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values"
"Check usage of remainder and reciprocal operations"	"hisl_0002: Usage of remainder and reciprocal operations"
"Check usage of square root operations"	"hisl_0003: Usage of square root operations"
"Check usage of log and log10 operations"	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
"Check usage of Reciprocal Sqrt blocks"	"hisl_0028: Usage of Reciprocal Square Root blocks"
"Check safety-related settings for hardware implementation"	"hisl_0071: Configuration Parameters > Hardware Implementation >Inconsistent hardware implementation settings"
"Check usage of recursions"	hisf_0004: Protect against recursive function calls to improve code compliance
"Check MATLAB functions not supported for code generation"	"himl_0012: Usage of MATLAB functions for code generation"
"Metrics for generated code complexity"	"himl_0013: Limitation of built-in MATLAB Function complexity"
"Check for parameter tunability ignored for referenced models"	"hisl_0072: Usage of tunable parameters for referenced models"
"Check usage of bit-shift operations"	"hisl_0073: Usage of bit-shift operations"
"Check safety-related diagnostic settings for variants"	"hisl_0074: Configuration Parameters > Diagnostics > Modeling issues related to variants"
"Check for disabled and parameterized library links"	"hisl_0075: Usage of library links"
"Check for unreachable and dead code"	"hisl_0101: Avoid operations that result in dead logic to improve code compliance"

Model Checks for IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Standard Compliance

You can check that your model or subsystem complies with selected aspects of the following standards by running the Model Advisor:

- ISO 26262:2018 *Road vehicles — Functional safety*
- ISO 25119:2018 *Tractors And Machinery For Agriculture And Forestry — Safety-Related Parts Of Control Systems*
- IEC 61508:2010 *Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems*
- EN 50128:2011 *Railway applications - Communication, Signalling and Processing Systems - Software for Railway Control and Protection Systems*
- EN 50657: 2017 *Railways Applications. Rolling stock applications. Software on Board Rolling Stock*
- IEC 62304:2015 *Medical Device Software – Software Life Cycle Processes*
- MISRA C:2012 *Guidelines for the Use of the C Language in Critical Systems*

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

- **By Task > Modeling Standards for ISO 26262**
- **By Task > Modeling Standards for ISO 25119**
- **By Task > Modeling Standards for IEC 61508**
- **By Task > Modeling Standards for EN 50128/EN 50657**
- **By Task > Modeling Standards for IEC 62304**

The table lists the IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 checks.

Subfolder	Model Advisor Checks
N/A	"Display configuration management data"
	"Display model metrics and complexity report"
	"Check for unconnected objects"
MISRA	"Check for missing error ports for AUTOSAR receiver interfaces"
	"Check for unsupported block names"
	"Check bus object names that are used as bus element names"
	"Check for equality and inequality operations on floating-point values"
	"Check for bitwise operations on signed integers"
	"Check integer word length"
	"Check for missing const qualifiers in model functions"
	"Check for recursive function calls"
	"Check for switch case expressions without a default case"
Bug Reports	"Display bug reports for IEC Certification Kit" (IEC Certification Kit)

Subfolder	Model Advisor Checks
	"Display bug reports for Simulink Check" (IEC Certification Kit)
	"Display bug reports for Simulink Coverage" (IEC Certification Kit)
	"Display bug reports for Requirements Toolbox" (IEC Certification Kit)
	"Display bug reports for Simulink Design Verifier" (IEC Certification Kit)
	"Display bug reports for Simulink Test" (IEC Certification Kit)
	"Display bug reports for Embedded Coder" (IEC Certification Kit)
	"Display bug reports for AUTOSAR Blockset" (IEC Certification Kit)
	"Display bug reports for Simulink PLC Coder" (IEC Certification Kit)
	"Display bug reports for HDL Coder" (IEC Certification Kit)
	"Display bug reports for Polyspace Bug Finder" (IEC Certification Kit)
	"Display bug reports for Polyspace Bug Finder Server" (IEC Certification Kit)
	"Display bug reports for Polyspace Code Prover" (IEC Certification Kit)
	"Display bug reports for Polyspace Code Prover Server" (IEC Certification Kit)

Following are the High-Integrity System Modeling checks that are applicable for the IEC 61508, IEC 62304, ISO 26262, ISO 25119, EN 50128, and EN 50657 standards.

Model Checks for High Integrity Systems Modeling

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

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

- **By Task > Modeling Standards for DO-178C/DO-331 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 61508 > High-Integrity Systems**
- **By Task > Modeling Standards for IEC 62304 > High-Integrity Systems**
- **By Task > Modeling Standards for EN 50128/EN 50657 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 26262 > High-Integrity Systems**
- **By Task > Modeling Standards for ISO 25119 > High-Integrity Systems**

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

This 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".

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 usage of variant blocks	"hisl_0023: Verification of variant blocks"
Check for root Inports with missing properties	"hisl_0024: Inport interface definition"
Check usage of Relational Operator blocks	"hisl_0017: Usage of blocks that compute relational operators (2)"
Check relational comparisons on floating-point signals	"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 > Code Generation > Optimization > Data initialization"
Check safety-related optimization settings for data type conversions	"hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values"
Check safety-related optimization settings for division arithmetic exceptions	"hisl_0054: Configuration Parameters > Code Generation > 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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related code generation identifier settings	"hisl_0049: Configuration Parameters > Code Generation > Identifiers"
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 remainder and reciprocal operations"
Check usage of Math Function blocks (log and log10 functions)	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
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 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 MATLAB Function interfaces with inherited properties	"himl_0002: Strong data typing at MATLAB function boundaries"
Check MATLAB Function metrics	"himl_0003: Complexity of user-defined MATLAB Functions"
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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
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 > Data Validity > 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 > Data Validity > 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 > Data Validity > Merge blocks"
Check safety-related diagnostic settings for Stateflow	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
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"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
Check if/elseif/else patterns in MATLAB Function blocks	"himl_0006: MATLAB code if / elseif / else patterns"
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 logical operators and functions in MATLAB Function blocks	"himl_0010: MATLAB code with logical operators and functions"
Check type and size of conditional expressions	"himl_0011: Data type and size of condition expressions"
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 for divide-by-zero calculations"	"hisl_0067: Protect against divide-by-zero calculations"
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"
Check safety-related optimization settings for specified minimum and maximum values	"hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values"
"Check usage of remainder and reciprocal operations"	"hisl_0002: Usage of remainder and reciprocal operations"
"Check usage of square root operations"	"hisl_0003: Usage of square root operations"
"Check usage of log and log10 operations"	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"

High Integrity System Model Check	Applicable High-Integrity System Modeling Guidelines
"Check usage of Reciprocal Sqrt blocks"	"hisl_0028: Usage of Reciprocal Square Root blocks"
"Check safety-related settings for hardware implementation"	"hisl_0071: Configuration Parameters > Hardware Implementation >Inconsistent hardware implementation settings"
"Check usage of recursions"	hisf_0004: Protect against recursive function calls to improve code compliance
"Check MATLAB functions not supported for code generation"	"himl_0012: Usage of MATLAB functions for code generation"
"Metrics for generated code complexity"	"himl_0013: Limitation of built-in MATLAB Function complexity"
"Check for parameter tunability ignored for referenced models"	"hisl_0072: Usage of tunable parameters for referenced models"
"Check usage of bit-shift operations"	"hisl_0073: Usage of bit-shift operations"
"Check safety-related diagnostic settings for variants"	"hisl_0074: Configuration Parameters > Diagnostics > Modeling issues related to variants"
"Check for disabled and parameterized library links"	"hisl_0075: Usage of library links"
"Check for unreachable and dead code"	"hisl_0101: Avoid operations that result in dead logic to improve code compliance"

See Also

Related Examples
•   "Run Model Advisor Checks and Review Results" on page 3-4

Model Checks for MISRA C:2012 Compliance

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

- "Check usage of Assignment blocks"
- "Check for blocks not recommended for MISRA C:2012"
- "Check for blocks not recommended for C/C++ production code deployment"
- "Check for unsupported block names"
- "Check configuration parameters for MISRA C:2012"
- "Check for equality and inequality operations on floating-point values"
- "Check for bitwise operations on signed integers"
- "Check for recursive function calls"
- "Check for switch case expressions without a default case"
- "Check for missing error ports for AUTOSAR receiver interfaces"
- "Check for missing const qualifiers in model functions"
- "Check integer word length"
- "Check bus object names that are used as bus element names"

See Also

Related Examples

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

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

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

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

From R2021b, these Model Advisor checks are now compliant with CERT C modelling guidelines:

- "Check configuration parameters for MISRA C:2012"
- Check usage of Abs blocks
- "Check usage of remainder and reciprocal operations"
- "Check usage of square root operations"
- Check usage of While Iterator blocks
- Check for blocks not recommended for C/C++ production code deployment
- Check data types for blocks with index signals
- "Check usage of Reciprocal Sqrt blocks"
- Check global variables in graphical functions
- "Check usage of bit-shift operations"
- Check safety-related optimization settings for data type conversions
- Check safety-related optimization settings for division arithmetic exceptions
- Check model file name
- Check model object names

See Also

Related Examples

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

Model Checks for Requirements Links

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

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

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

For more information about these Model Advisor checks, see "Requirements Consistency Checks" (Requirements Toolbox)

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" (Requirements Toolbox)
- "Run Model Advisor Checks and Review Results" on page 3-4
- "High-Integrity System Modeling"

Replace Exact Clones with Subsystem Reference

Exact clones are modeling patterns that have identical block types, connections, and parameter values. The Clone Detector identifies these clones across referenced model boundaries. You can then reuse components by replacing exact clones with library links and Subsystem Reference blocks. To replace exact clones with library links, see "Enable Component Reuse by Using Clone Detection" on page 3-33. This example demonstrates how to replace subsystem clones with "Subsystem Reference" blocks.

Identify Exact Clones

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

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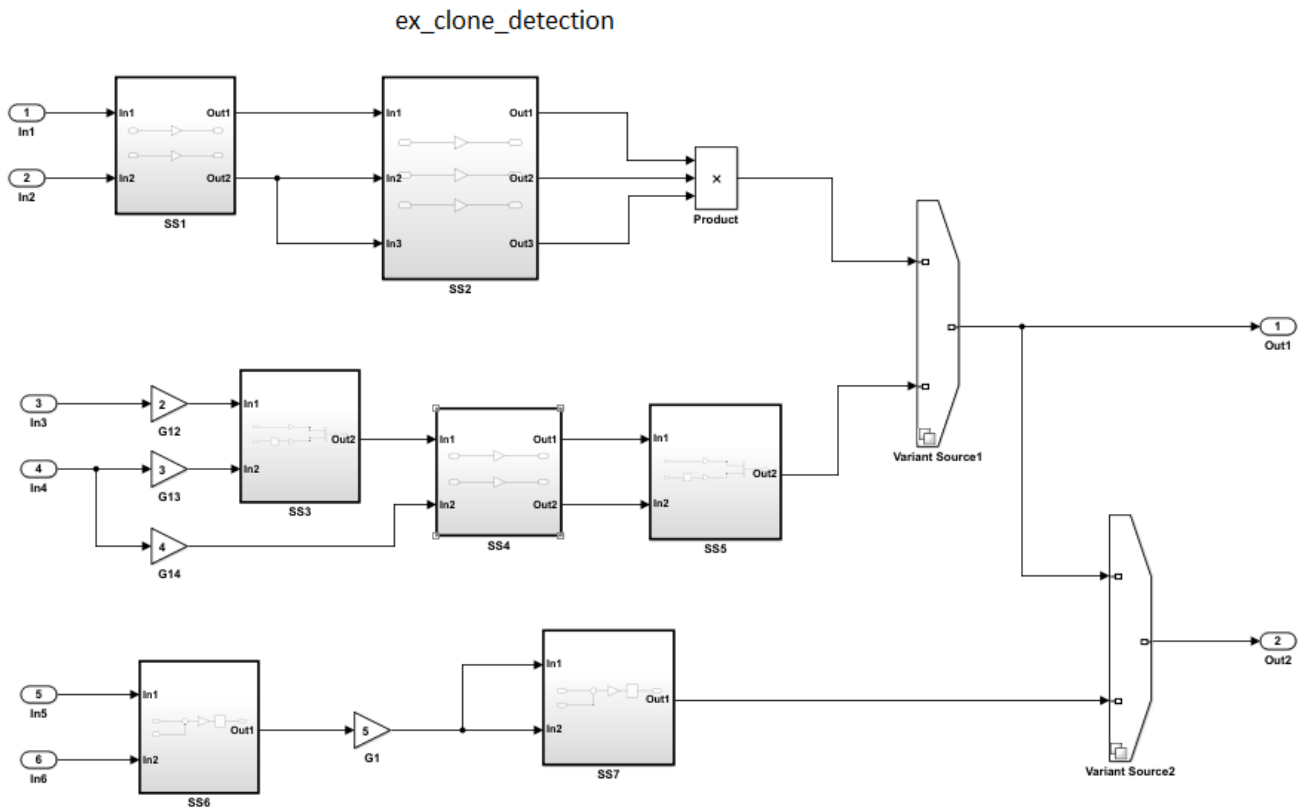

ex_clone_detection

Copyright 2017 The MathWorks Inc.

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

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

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

4 The app opens the **Clone Detector** tab. This example takes you through each section.
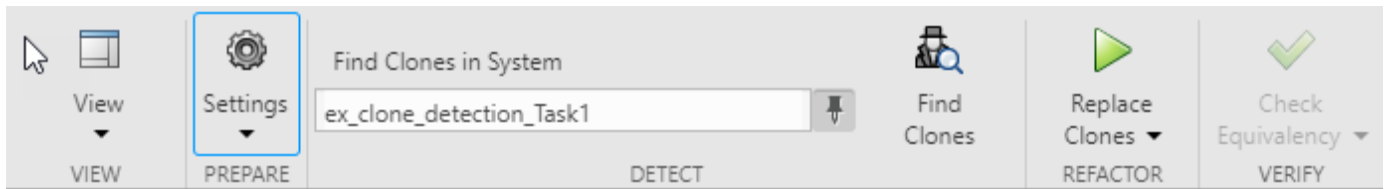
Set Up Panes for Clone Detection

The app displays information on three panes. To open the panes, use the **View** menu. The panes are:

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

Set the Parameters for Clone Detection

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

- Select **Replace Exact Clones with Subsystem References**.
- Click **Exclude Components** to access the **Exclude model references**, **Exclude library Links**, and **Exclude inactive and commented out regions** options. Enabling the **Exclude inactive and commented out regions** option, leads to exact clone SS1 not being identified because of Variant Source block in the model. For more information, see "Exclude Components from Clone Detection". Keep the **Exclude inactive and commented out regions** option cleared.
- Click **Detect Clones Across Model** to enable detect clones anywhere across the model. You can choose the values of **Minimum Region Size** and **Minimum Clone Group Size** to detect the clones with these matching blocks. The default size is set to 2.

Identify Clones in the Model

1 Click **Find Clones** to identify clones.
2 The color of subsystems SS1 and SS4 changes to red to indicate that they are exact clones.

Analyze the Clone Detection Results

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

1 In the **Clone Detection Results and Actions** panel, on the **Clone Results** tab, a clone group Exact Clone Group 1 is displayed.
2 Click the > symbol next to Exact Clone Group 1 to see all of the subsystems that are exact clones and the number of blocks per clone.
3 In the **Clone Detection Results and Actions** pane, click the **Logs** tab. Click the hyperlink on the **Logs** pane.

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

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

5   In the **Detected Clone Properties** pane, in the **Refactor Benefits** section, you can consider the
    percentage of exact clones present.

    Refactoring the model reduces 6.4516% of the model reuse.
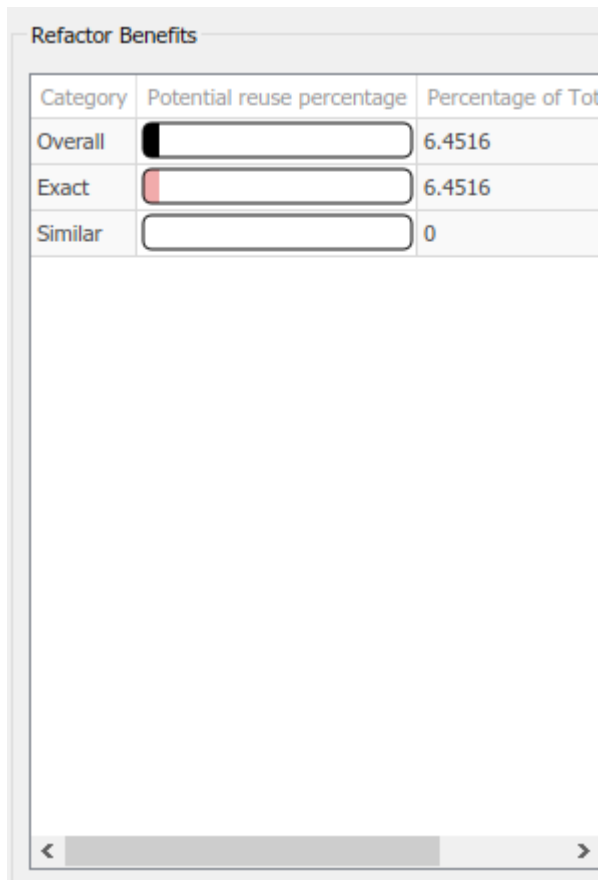
Replace Clones

1   In the **Clone Detector** tab, click **Replace Clones**. The exact clones are replaced with Subsystem
    Reference blocks. You can see the `.slx` files of the subsystem references in your working folder.

2   You can restore the model to its original configuration in the **Clone Detection Results and
    Actions** pane. Open the **Logs** tab, click the hyperlink for the version you want to restore, and
    click **Restore**.

Check the Equivalency of the Model

If you have a Simulink Test, you can check equivalence of the new model is to the original model in
the **Clone Detection** tab by clicking **Check Equivalency**.
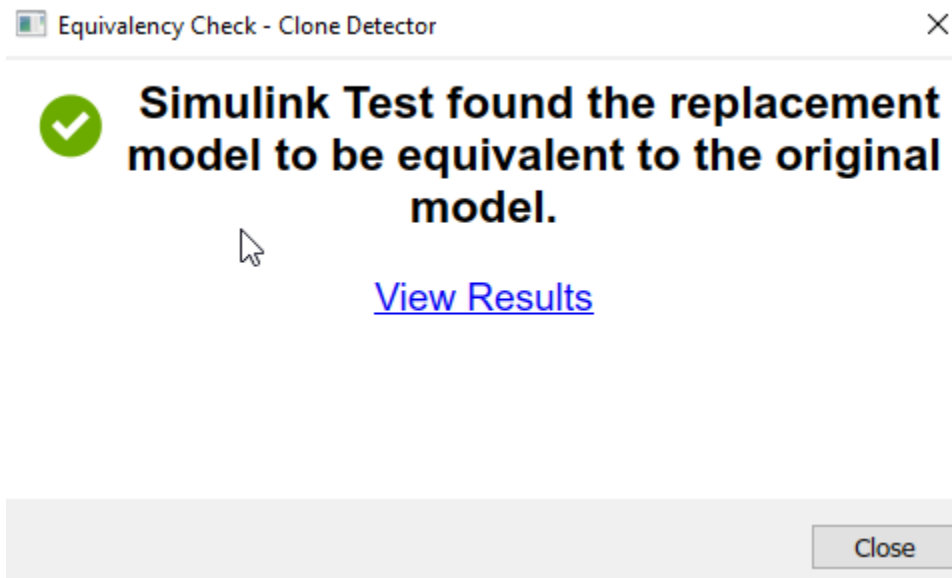
See Also

Related Examples

- "Generate Reusable Code from Library Subsystems Shared Across Models" (Simulink Coder)
- Clone Detector
- "Enable Component Reuse by Using Clone Detection" on page 3-33

Detect and Replace Subsystem Clones Programmatically

Clones are modeling patterns that have identical block types and connections. You can refactor your model by creating library blocks from subsystem clones and replacing the clones with links to those library blocks, which enable you to reuse components. For more information about clones, see Enable Component Reuse by Using Clone Detection on page 3-33.

Programmatically checking for clones during the model submission process helps you identify opportunities to reuse subsystems before the model is deployed into the main product branch. When updating your model, you can use the Clone Detector app and clone detector API simultaneously. When you use the clone detector API, the detected clones appear in the Simulink Editor.

This example shows how to use the clone detection APIs to identify and replace clones in a single model by creating a library file with subsystem blocks and replacing the clones with links to blocks in the library file.

In this example, you learn how to use:

- Simulink.CloneDetection.findClones to find clones in a model.
- Simulink.CloneDetection.replaceClones to replace clones in a model.
- Simulink.CloneDetection.checkEquivalency to check the equivalency of the updated model with the original model.
- Simulink.CloneDetection.Settings to add conditions to the `findClones` operation.
- Simulink.CloneDetection.ReplacementConfig to add conditions to the `replaceClones` operation.

Identify Clones in a Model

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

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

Save the model in the current working directory.

2 To find subsystem clones, use the function `Simulink.CloneDetection.findClones()`. This function creates an object called `cloneResults`.

```
cloneResults = Simulink.CloneDetection.findClones('ex_clone_detection_B')

cloneResults =
  Results with properties:
        Clones: [1×1 struct]
    ExceptionLog: ''
```

The `cloneResults` object has `Clones`, which is a structure with two fields, `Summary` and `CloneGroups`.

```
cloneResults.Clones

ans =
  struct with fields:
        Summary: [1×1 struct]
    CloneGroups: [1×2 struct]
```

3 View the `Summary` field.

```
cloneResults.Clones.Summary
```

```
ans =
  struct with fields:
                 CloneGroups: 2
               SimilarClones: 5
                 ExactClones: 0
                      Clones: 5
    PotentialReusePercentage: [1×1 struct]
```

In this example, the model has two `CloneGroups` with matching subsystem patterns, five `SimilarClones`, and zero `ExactClones`, and the five subsystem `Clones`.

4   View the `CloneGroups` field.

```
cloneResults.Clones.CloneGroups
```

```
ans =
  1×2 struct array with fields:
    Name
    Summary
    CloneList
```

The model in this example returns an array of two `CloneGroups`. Each array includes the `Name`, `Summary` and `CloneList`.

5   View the details of first clone group.

```
cloneResults.Clones.CloneGroups(1)
```

```
ans =
  struct with fields:
        Name: 'Similar Clone Group 1'
     Summary: [1×1 struct]
    CloneList: {3×1 cell}
```

6   View the `Summary`.

```
cloneResults.Clones.CloneGroups(1).Summary
```

```
ans =
  struct with fields:
    ParameterDifferences: [1×1 struct]
                  Clones: 3
           BlocksPerClone: 8
                CloneType: 'Similar'
          BlockDifference: 1
```

7   View the `CloneList` of the first CloneGroup.

```
cloneResults.Clones.CloneGroups(1).CloneList
```

```
ans =
  3×1 cell array
    {'ex_clone_detection_B/Subsystem1'}
    {'ex_clone_detection_B/Subsystem2'}
    {'ex_clone_detection_B/Subsystem3'}
```

Similarly, You can find the results of other `CloneGroups` using the above steps.

Replace Clones in a Model

1 To replace clones in a model, use the function
Simulink.CloneDetection.replaceClones(). This function uses the cloneResults object
from the findClones function.

```
cloneReplacementResults = Simulink.CloneDetection.replaceClones(cloneResults)
```

```
cloneReplacementResults =
  ReplacementResults with properties:
    ReplacedClones: [1×5 struct]
    ExcludedClones: {}
```

The cloneReplacementResults object includes two properties, ReplacedClones and
ExcludedClones.

2 View the contents of ReplacedClones property.

```
cloneReplacementResults.ReplacedClones
```

```
ans =
  1×5 struct array with fields:
    Name
    ReferenceSubsystem
```

The 1-by-5 array indicates that the function replaced five subsystem clones in the model.

3 View the list of replaced subsystem clones.

```
struct2table(cloneReplacementResults.ReplacedClones)
```

```
ans =
  5×2 table
                 Name                        ReferenceSubsystem
    _____    _____
    {'ex_clone_detection_B/Subsystem1'}    {'newLibraryFile/Subsystem1'}
    {'ex_clone_detection_B/Subsystem2'}    {'newLibraryFile/Subsystem1'}
    {'ex_clone_detection_B/Subsystem3'}    {'newLibraryFile/Subsystem1'}
    {'ex_clone_detection_B/SS3'       }    {'newLibraryFile/SS1'       }
    {'ex_clone_detection_B/SS4'       }    {'newLibraryFile/SS1'       }
```

Identify Clones Using Subsystem Reference Blocks

1 Save the model and library file in the current working directory.

```
ex_clone_detection_E
libname
```

2 Use the Simulink.CloneDetection.Settings() class to create an object that specifies
certain conditions for finding clones in a model.

```
cloneDetectionSettings = Simulink.CloneDetection.Settings()
```

```
cloneDetectionSettings =

  Settings with properties:

                    IgnoreSignalName: 0
                  IgnoreBlockProperty: 0
              ExcludeModelReferences: 0
                  ExcludeLibraryLinks: 0
                ExcludeInactiveRegions: 0
```

```
                         SelectedSystemBoundary: ''
                        DetectClonesAcrossModel: 0
                 FindClonesRecursivelyInFolders: 1
                          ParamDifferenceThreshold: 50
        ReplaceExactClonesWithSubsystemReference: 0
                                       Libraries: {}
                                         Folders: {}
```

3   Set the `ParamDifferenceThreshold` parameter. This parameter specifies the number of differences that subsystems must have to be considered clones.

```
cloneDetectionSettings.ParamDifferenceThreshold = 0
```

```
cloneDetectionSettings =

  Settings with properties:
                              IgnoreSignalName: 0
                           IgnoreBlockProperty: 0
                         ExcludeModelReferences: 0
                             ExcludeLibraryLinks: 0
                           ExcludeInactiveRegions: 0
                          SelectedSystemBoundary: ''
                        DetectClonesAcrossModel: 0
                 FindClonesRecursivelyInFolders: 1
                          ParamDifferenceThreshold: 0
        ReplaceExactClonesWithSubsystemReference: 0
                                       Libraries: {}
                                         Folders: {}
```

A value of 0 indicates the subsystems must be identical.

4   Add a reference library file to use to match the clone patterns in the `cloneDetectionSettings` object. In this example, SSL1 and SSL2 are subsystem patterns in the library `libName`.

```
cloneDetectionSettings = cloneDetectionSettings.addLibraries('libname')
```

```
cloneDetectionSettings =

  Settings with properties:
                              IgnoreSignalName: 1
                           IgnoreBlockProperty: 0
                         ExcludeModelReferences: 0
                             ExcludeLibraryLinks: 0
                           ExcludeInactiveRegions: 0
                          SelectedSystemBoundary: ''
                        DetectClonesAcrossModel: 0
                 FindClonesRecursivelyInFolders: 1
                          ParamDifferenceThreshold: 50
        ReplaceExactClonesWithSubsystemReference: 0
                                       Libraries: {'C:\Users\Examples\libname.slx'}
                                         Folders: {}
```

5   To find clones, execute the function `Simulink.CloneDetection.findClones()` using the model name and `cloneDetectionSettings` object.

```
cloneResults = Simulink.CloneDetection.findClones('ex_clone_detection_E', cloneDetectionSettings)
```

```
cloneResults =
  Results with properties:
    Clones: [1×1 struct]
```

```
cloneResults.Clones.Summary
```

```
ans =
  struct with fields:
```

```
              CloneGroups: 2
            SimilarClones: 5
              ExactClones: 0
                   Clones: 5
    PotentialReusePercentage: [1×1 struct]
```

In this example, the model has two `CloneGroups`, five `SimilarClones`, zero `ExactClones`, and five subsystem `Clones`.

6 View the details of first `CloneGroup`.

```
 cloneResults.Clones.CloneGroups(1)
```

```
ans =
  struct with fields:
        Name: 'libname/SSL1'
     Summary: [1×1 struct]
   CloneList: {3×1 cell}
```

Replace Clones with Conditions

1 1. To specify conditions for `replaceClones` function, create a handle using the `Simulink.CloneDetection.ReplacementConfig()` class:

```
 cloneReplacementConfig = Simulink.CloneDetection.ReplacementConfig()
```

```
cloneReplacementConfig =
  ReplacementConfig with properties:
    LibraryNameToAddSubsystemsTo: 'newLibraryFile'
                   IgnoredClones: {}
```

2 Add subsystems to the `IgnoredClones` list. In this example, ignore `Subsystem1` to avoid replacing it with a clone.

```
cloneReplacementConfig.addCloneToIgnoreList('ex_clone_detection_E/Subsystem1')
```

```
ans =
  ReplacementConfig with properties:
    LibraryNameToAddSubsystemsTo: 'newLibraryFile'
                   IgnoredClones: {'ex_clone_detection_E/Subsystem1'}
```

3 To replace clones, use the `replaceClones` function with `cloneResults` and `cloneReplacementConfig` as the input arguments.

```
cloneReplacementResults = Simulink.CloneDetection.replaceClones(cloneResults, cloneReplacementConfig)
```

```
cloneReplacementResults =
  ReplacementResults with properties:
    ReplacedClones: [1×4 struct]
    ExcludedClones: [1×1 struct]
```

4 View the `ReplacedClones` property.

```
struct2table(cloneReplacementResults.ReplacedClones)
```

```
ans =
  4×2 table
                 Name                    ReferenceSubsystem
    _____    _____
    {'ex_clone_detection_E/SS3'   }    {'libname/SSL1'}
    {'ex_clone_detection_E/SS4'   }    {'libname/SSL1'}
```

```
                   {'ex_clone_detection_E/Subsystem1'}     {'libname/SSL2'}
                   {'ex_clone_detection_E/Subsystem2'}     {'libname/SSL2'}
```

The **SSL1** and **SSL2** Reference Subsystem blocks from the reference library replaced the subsystem clones in the model.

5 View the `ExcludedClones` property.

```
struct2table(cloneReplacementResults.ExcludedClones)

ans =
  1×2 table
                          Name                    ReferenceSubsystem
               _____     _____
               {'ex_clone_detection_E/Subsystem1'}    {'unselected'}
```

Check the Equivalency of the Model

You can check if the updated model is equivalent with the original model by using the `Simulink.CloneDetection.checkEquivalency()` function. This function uses Simulink Test Manager to compare the simulation results of the saved original model with the updated model and saves the results in the `checkEquiResults` handle.

```
 checkEquiResults = Simulink.CloneDetection.checkEquivalency(cloneReplacementResults)

[21-Dec-2020 16:35:13] Running simulations...
[21-Dec-2020 16:35:32] Completed 1 of 2 simulation runs
[21-Dec-2020 16:35:33] Completed 2 of 2 simulation runs

checkEquiResults =
  EquivalencyCheckResults with properties:
    List: [1×1 struct]
```

View the check equivalence results.

```
 checkEquiResults.List

ans =
  struct with fields:
    IsEquivalencyCheckPassed: 1
             OriginalModel: 'm2m_ex_clone_detection_E/snapshot_2020_12_21_16_35_06_ex_clone_detection_E.slx'
              UpdatedModel: 'ex_clone_detection_E.slx'
```

The property `IsEquivalencyCheckPassed` is 1, which suggests that the models are equivalent. The `OriginalModel` and `UpdatedModel` properties show which models the function checked.

See Also

Related Examples

• Clone Detector
• "Enable Component Reuse by Using Clone Detection" on page 3-33

Find Clones Across the Model

Clones are modeling patterns that have identical block types and connections. You can refactor your model by creating library blocks from these clone patterns and replacing the clones with links to the library blocks, which enable you to reuse the components. For more information about clones, see Enable Component Reuse by Using Clone Detection on page 3-33.

You can search for clones in a subsystem or anywhere across the model.

- Subsystem clones: Identifies clones only in a subsystems.
- Clones across model: Identifies clones anywhere across the model.

This example shows how to use the Clone Detector app and APIs to identify clones anywhere across the model, and then replace them with links to library blocks.

Identify Clones by Using the App

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

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


Copyright 2020-2021 The MathWorks, Inc.
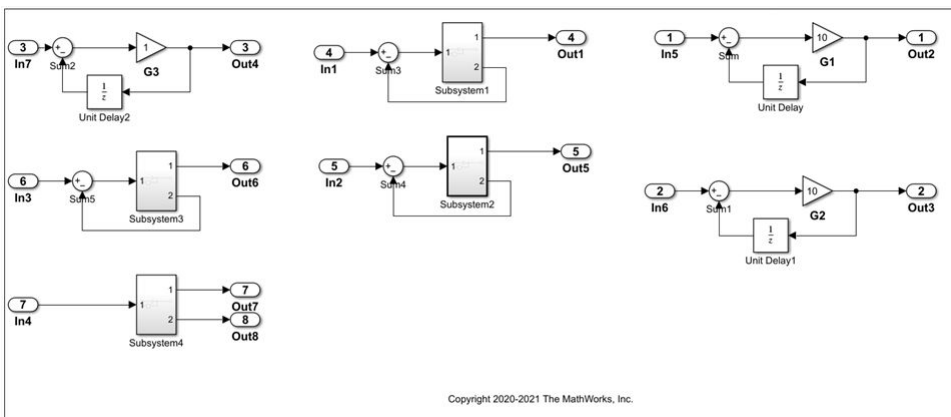
2   Save the model in the current working directory.

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

```
clonedetection("mClonesAnywhereSimilar")
```

4   To set up the parameters for clone detection, click **Settings**. Under **Clone anywhere settings**, click **Detect Clones Across Model**.
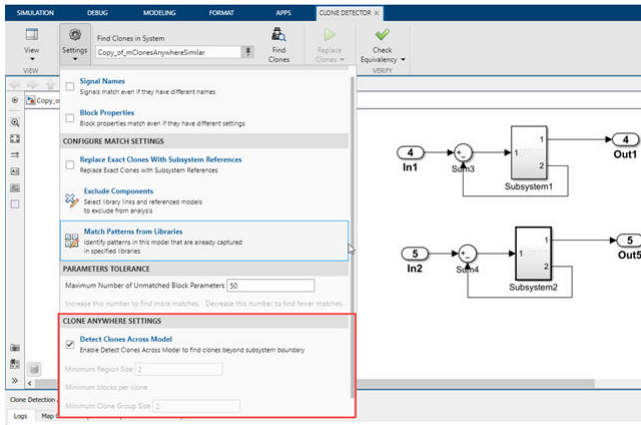
Minimum Region Size and **Minimum Clone Group Size** are set to 2 by default. The **Minimum Region Size** parameter represents the minimum blocks needed per clone region and the **Minimum Clone Group Size** parameter represents the minimum clone occurrences needed to define it as a clone group.

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

6   The app highlights the clones. Exact clones are highlighted in red and similar clones are highlighted in blue.
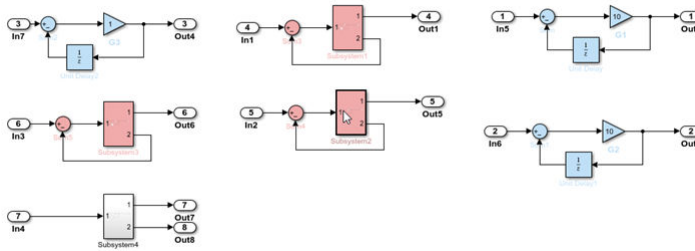
The clones highlighted in this example include clones identified in and outside of subsystems and a Simulink block connected to a subsystem.

7   Click **Replace Clones**.

The app refactors the model and replaces the clones with links to the `newLibraryFile` library file in your working directory. The app replaces the Simulink clone blocks outside of the subsystems with linked Subsystem blocks.

Identify Clones Programmatically

1   Use the `Simulink.CloneDetection.Settings` class to create an object.

```
cloneDetectionSettings = Simulink.CloneDetection.Settings()

cloneDetectionSettings =

                     IgnoreSignalName: 0
                   IgnoreBlockProperty: 0
                ExcludeModelReferences: 0
                   ExcludeLibraryLinks: 0
```

```
                   FindClonesRecursivelyInFolders: 1
                          ParamDifferenceThreshold: 50
    ReplaceExactClonesWithSubsystemReference: 0
                                       Libraries: {}
                                         Folders: {}
                       DetectClonesAcrossModel: 0
                        ExcludeInactiveRegions: 0
```

2 To search for clones across the model, set `DetectClonesAcrossModel` to 1.

```
cloneDetectionSettings.DetectClonesAcrossModel = 1
```

```
cloneDetectionSettings =
```

```
                              IgnoreSignalName: 0
                           IgnoreBlockProperty: 0
                        ExcludeModelReferences: 0
                           ExcludeLibraryLinks: 0
                         SelectedSystemBoundary: []
              FindClonesRecursivelyInFolders: 1
                      ParamDifferenceThreshold: 50
   ReplaceExactClonesWithSubsystemReference: 0
                                     Libraries: {}
                                       Folders: {}
                       DetectClonesAcrossModel: 1
                        ExcludeInactiveRegions: 0
                            MinimumRegionSize: 2
                         MinimumCloneGroupSize: 2
```

`MinimumRegionSize` and `MinimumCloneGroupSize` are set to 2 by default. You can change their values.

3 To find clones, execute the function Simulink.CloneDetection.findClones using the `cloneDetectionSettings` object.

```
cloneResults = Simulink.CloneDetection.findClones('mClonesAnywhereSimilar',cloneDetectionSettings)
```

```
cloneResults =
```

```
        Clones: [1×1 struct]
    ExceptionLog: ''
```

```
cloneResults.Clones =
```

```
  Results with properties:

        Summary: [1×1 struct]
    CloneGroups: [1×2 struct]
```

For more details on the clone detection APIs, see "Detect and Replace Subsystem Clones Programmatically" on page 3-90.

See Also

Related Examples

- Simulink.CloneDetection.findClones
- Simulink.CloneDetection.Settings

Detect Clones Programmatically on Multiple Models Across Different Folders

This example shows how to programmatically detect clones across multiple models located in different folders. For more information about the clone detection APIs, see "Detect and Replace Subsystem Clones Programmatically" on page 3-90.

This example demonstrates how to use the clone detection APIs to identify clones in six Simulink® models present in a folder. Replacement of clones is not possible using this example workflow. To replace clones programmatically in multiple models, see "Detect and Replace Clones Programmatically in a Loop on Multiple Models" on page 3-102.

1. At the MATLAB® command line, enter:

```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
ex_clone_detection_A
ex_clone_detection_B
ex_clone_detection_C
ex_clone_detection_D
ex_clone_detection_E
ex_clone_detection_F
```

Save the models to a writeable folder.

2. Use the `Simulink.CloneDetection.Settings` class to create an object.

```
cloneDetectionSettings = Simulink.CloneDetection.Settings();
```

3. Add the path of the folder with the models to the `cloneDetectionSettings` object.

```
cloneDetectionSettings.Folders = {'D:\models'}
```

```
cloneDetectionSettings =

  Settings with properties:

                          IgnoreSignalName: 0
                       IgnoreBlockProperty: 0
                    ExcludeModelReferences: 0
                       ExcludeLibraryLinks: 0
                   ParamDifferenceThreshold: 50
    ReplaceExactClonesWithSubsystemReference: 0
                                 Libraries: {}
                                   Folders: {'D:\models'}
                      DetectClonesAcrossModel: 0
                      ExcludeInactiveRegions: 0
                 FindClonesRecursivelyInFolders: 1
```

4. To find clones, execute the function `Simulink.CloneDetection.findClones` using the `cloneDetectionSettings` object.

```
cloneResults = Simulink.CloneDetection.findClones(cloneDetectionSettings)
```

```
cloneResults =
```

```
   Results with properties:

         Clones: [1×1 struct]
   ExceptionLog: {}
```

The `cloneResults` is an object of `Simulink.CloneDetection.Results` class which has two properties, `Clones` and `ExceptionLog`.

5. View the `Clones.Summary` field.

```
   cloneResults.Clones.Summary
```

```
ans =

  struct with fields:

                 CloneGroups: 4
               SimilarClones: 18
                 ExactClones: 8
                      Clones: 26
     PotentialReusePercentage: [1×1 struct]
```

In this example, the models have four different clone groups with matching subsystem patterns, eighteen similar clones, and eight exact clones, and the twenty-six subsystem clones.

5. View the details of first clone group.

```
   cloneResults.Clones.CloneGroups(1)
```

```
ans =

  struct with fields:

        Name: 'Exact Clone Group 1'
     Summary: [1×1 struct]
    CloneList: {5×1 cell}
```

6. View the summary of first clone group.

```
   cloneResults.Clones.CloneGroups(1).Summary
```

```
ans =

  struct with fields:

    ParameterDifferences: [1×1 struct]
                  Clones: 5
           BlocksPerClone: 5
                CloneType: 'Exact'
          BlockDifference: 0
```

7. View the clone list of the first clone group.

```
cloneResults.Clones.CloneGroups(1).CloneList
```

```
ans =

  5×1 cell array

    {'ex_clone_detection_A/Subsystem'}
    {'ex_clone_detection_B/Subsystem'}
    {'ex_clone_detection_C/Subsystem'}
    {'ex_clone_detection_D/Subsystem'}
    {'ex_clone_detection_F/Subsystem'}
```

Similarly, you can find the results of other clone groups using the above steps.

See Also

Related Examples

- Clone Detector
- "Enable Component Reuse by Using Clone Detection" on page 3-33
- "Detect and Replace Subsystem Clones Programmatically" on page 3-90

Detect and Replace Clones Programmatically in a Loop on Multiple Models

This example shows how to programmatically detect and replace clones on a multiple models in a loop by operating on models individually. For more information about Clone Detection APIs, see "Detect and Replace Subsystem Clones Programmatically" on page 3-90.

This example shows how to detect and replace clones programmatically for five Simulink® models using the library file `TestLib_1` as a subsystem reference to replace clones.

1. At the MATLAB® command line, enter:

```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
ex_clone_detection_A
ex_clone_detection_B
ex_clone_detection_C
ex_clone_detection_E
ex_clone_detection_F
TestLib_1
```
Save the models and library file in the current working directory.

2. Create an array to add the models to:

```
modelList = {};
```

3. Add the models to the `modelList` array:

```
modelList{end+1,1} = 'ex_clone_detection_A';
modelList{end+1,1} = 'ex_clone_detection_B';
modelList{end+1,1} = 'ex_clone_detection_C';
modelList{end+1,1} = 'ex_clone_detection_F';
```

4. Define containers to store `Results`, `ReplacementResults` and `equivalencyCheck` object for the models.

```
cloneResultsStorage = containers.Map();
cloneReplacementStorage = containers.Map();
equivalencyCheckStorage = containers.Map();
```

5. Add the library file to the `cloneDetectionSettings` object created from `Settings` class.

```
libName = 'TestLib_1';
cloneDetectionSettings = Simulink.CloneDetection.Settings();
cloneDetectionSettings = cloneDetectionSettings.addLibraries(libName);
```

6. Use a loop to cycle through the models using the `Simulink.CloneDetection.findClones`, `Simulink.CloneDetection.replaceClones`, and `Simulink.CloneDetection.checkEquivalency` functions.

```
for modelIndex = 1:length(modelList)
    modelName = modelList{modelIndex};
    try
        cloneResults = Simulink.CloneDetection.findClones(modelName, cloneDetectionSettings);
        cloneResultsStorage(modelName) = cloneResults;
        cloneReplacementResults = Simulink.CloneDetection.replaceClones(cloneResults);
        cloneReplacementStorage(modelName) = cloneReplacementResults;
```

```
        equivalencyCheckResults = Simulink.CloneDetection.checkEquivalency(cloneReplacement
        equivalencyCheckStorage(modelName) = equivalencyCheckResults;
    catch exception
    end
end
```

You can access the results of `cloneResultsStorage`, `cloneReplacementStorage`, and `equivalencyCheckStorage` objects for individual models. For more details, see "Detect and Replace Subsystem Clones Programmatically" on page 3-90.

See Also

Related Examples

- Clone Detector
- "Enable Component Reuse by Using Clone Detection" on page 3-33
- "Detect Clones Programmatically on Multiple Models Across Different Folders" on page 3-99

Running Clone Detection Custom Script in a Project

This example shows how to run a custom script for detecting clones on the set of Simulink® model files managed in a project. Creating a custom script would help to organize and automate the large modeling projects. For more information on setting up a custom task, see "Run Custom Tasks with a Project".

This example shows how to create and run a custom script to automate the clone detection across all the models in a Simulink project. For more information on clone detection, see "Enable Component Reuse by Using Clone Detection" on page 3-33.

This example uses the Airframe project to demonstrate the working of clone detection across multiple models.

1. Open the Airframe project and use `currentProject` to get a project object.

```
sldemo_slproject_airframe;
project = currentProject;
```

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

2. Use the `Simulink.CloneDetection.Settings` class to create an object for find clones operation. Add the path of the `RootFolder` to `cloneDetectionSettings` object.

```
cloneDetectionSettings = Simulink.CloneDetection.Settings();
cloneDetectionSettings.Folders = project.RootFolder;
```

3. To find clones, execute the function `Simulink.CloneDetection.findClones` using the `cloneDetectionSettings` object.

```
cloneResults = Simulink.CloneDetection.findClones(cloneDetectionSettings);
```

4. You can highlight the subsystem clone in a model using the function `Simulink.CloneDetection.highlightClone`.

```
Simulink.CloneDetection.highlightClone(cloneResults, 'AnalogControl/Saturation Detection');
```
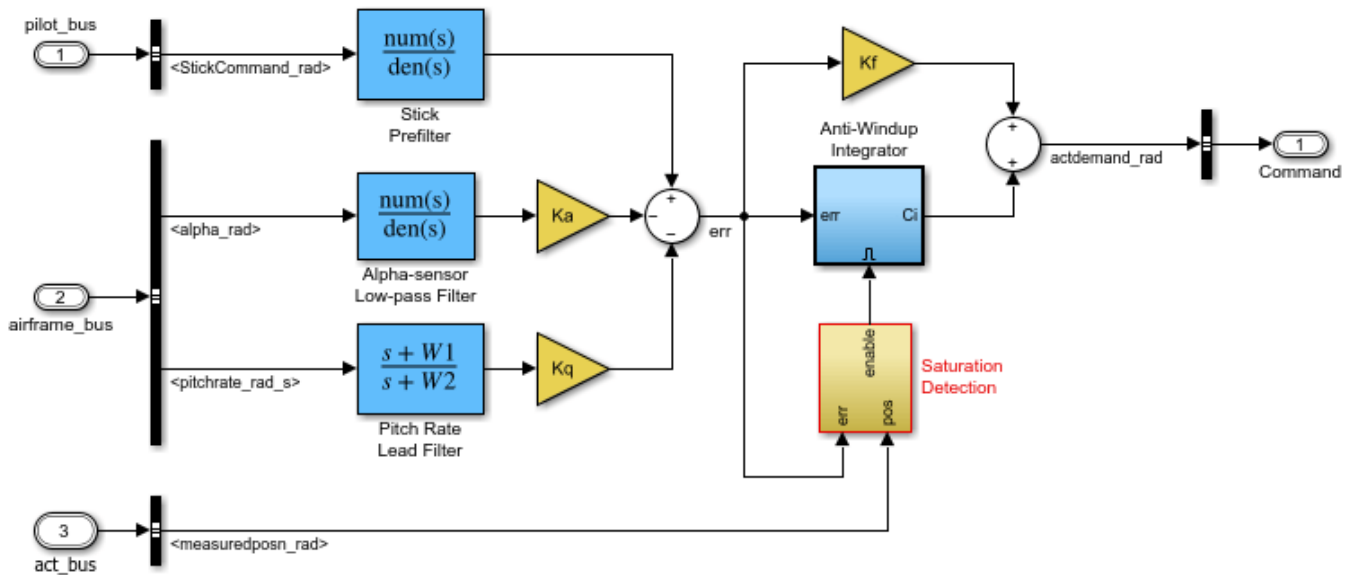
Copyright 1990-2018 The MathWorks, Inc.

The clone results is an object of `Simulink.CloneDetection.Results` class. For detailed information on how to see the clone results, see "Detect and Replace Subsystem Clones Programmatically" on page 3-90.

See Also

Related Examples
- Clone Detector
- "Enable Component Reuse by Using Clone Detection" on page 3-33
- "Detect Clones Programmatically on Multiple Models Across Different Folders" on page 3-99

Run Custom Model Advisor Checks on Architecture Models

Use Custom Model Advisor checks on Architecture models (System Composer).

Following is an example to identify the exceeded limit of Inports and Outports of a component in an Architecture model.

1. On the MATLAB command window, run `Advisor.Manager.refresh_customizations`. This step is necessary to publish the custom checks to the Model Advisor.

2. Open the architecture model.

```
open_system('slcheck_demo_systemcomposer.slx');
copyfile sl_customization_orig.m sl_customization.m f
```
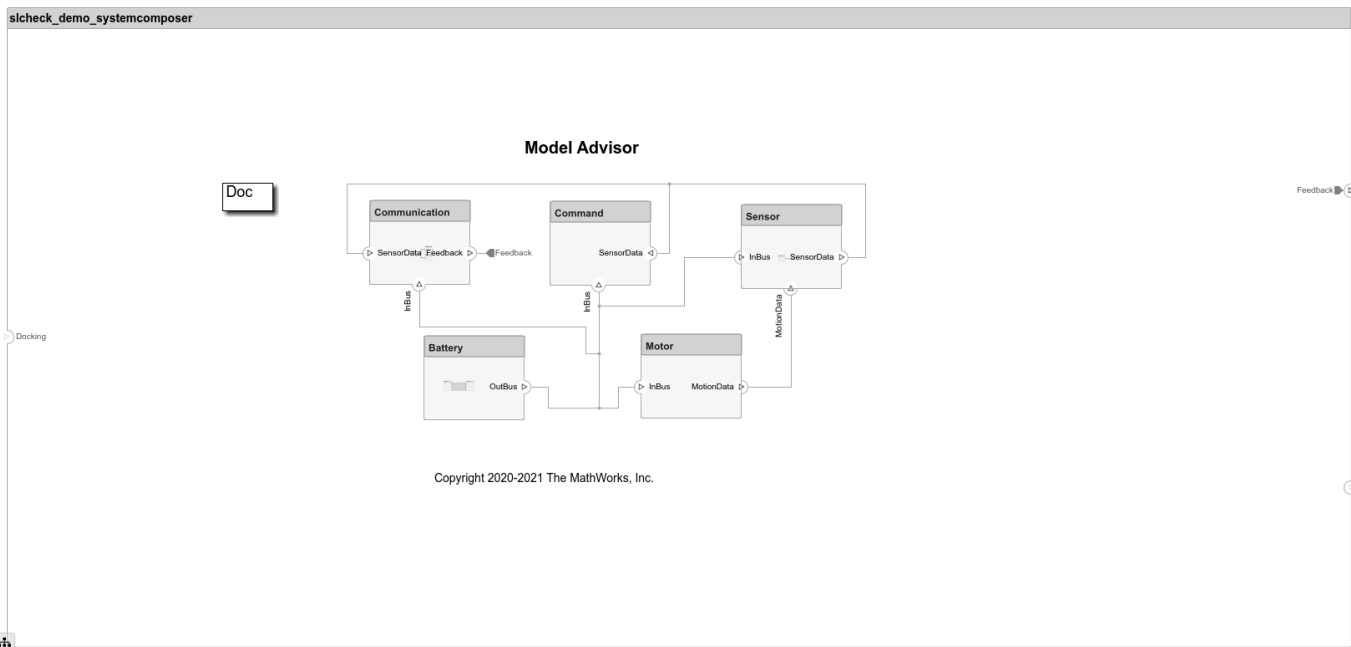

3. From the **Modeling** tab, open **Model Advisor**. You can also open the Model Advisor by typing this command at the MATLAB command prompt:

```
modeladvisor('slcheck_demo_systemcomposer.slx');

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

4. On the Model Advisor window, click **By Product > System Composer** and Select `Check number of Inports and Outports of a component` check.
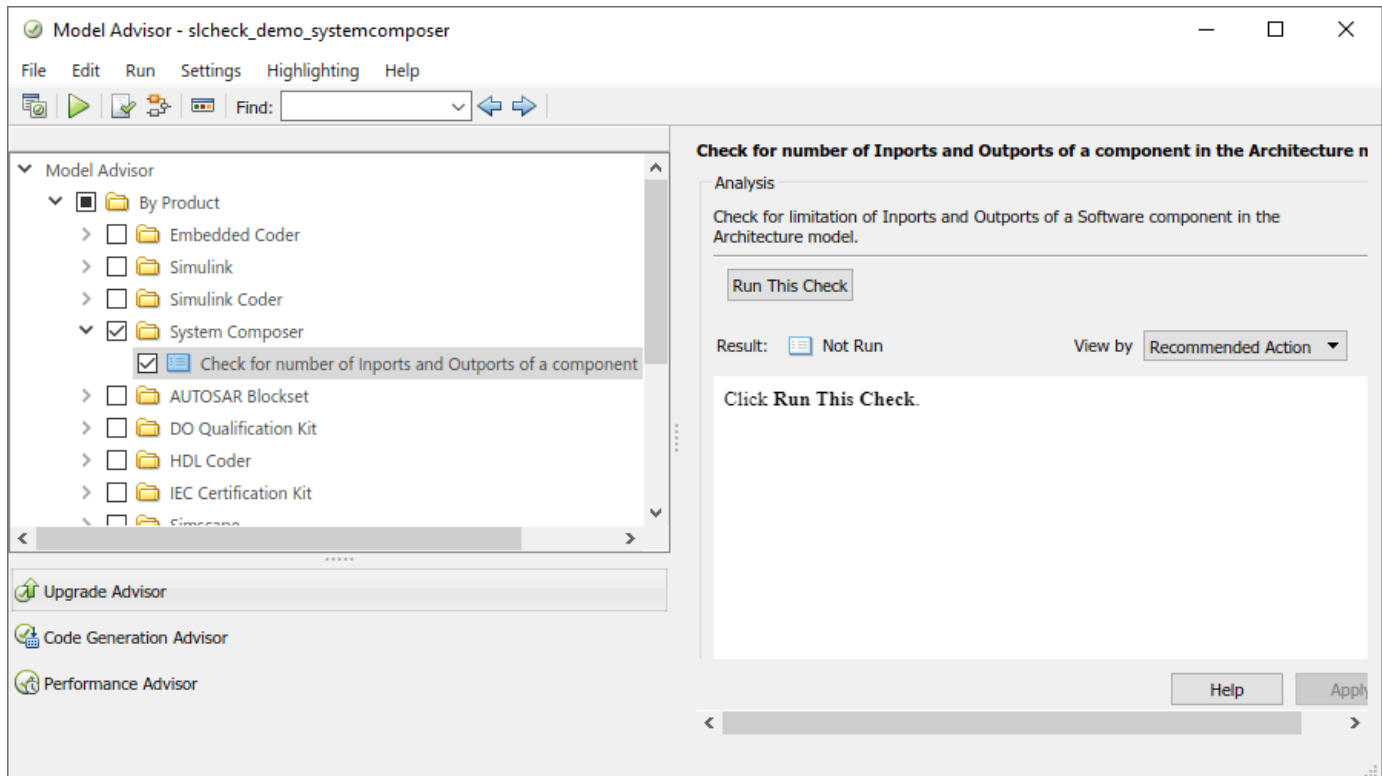
5. Right-click the check and select **Run This Check**. Model Advisor runs the selected check on the architecture model and displays the results.

6. Open the `sl_customization` file to view the demo source code.

```
edit sl_customization.m
```

7. Change the hardcoded threshold value for Inports and Outports to any desired values.

```
22    function sysCompPorts(system, checkObj)
23    Compwithmoreports = {};  result = []; allChecksPassed = true;
24    mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
25    Allcomponents = find_system(system,'LookUnderMasks','All','FollowLinks','On','BlockType','SubSystem');
26    for i = 1:length(Allcomponents)
27        Allports = get_param(Allcomponents{i},'PortHandles');
28        if length(Allports.Inport) > 5 || length(Allports.Outport) > 5
29            Compwithmoreports{end+1} = get_param(Allcomponents{i},'Handle'); %#ok<*SAGROW,*AGROW>
30            allChecksPassed = false;
31        end
32    end
```

8. Re-run the check to verify the updated results.

See Also

Related Examples

- "Define Custom Model Advisor Checks" on page 6-45
- "Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor" on page 6-9
- "Define Custom Edit-Time Checks that Fix Issues in Architecture Models" on page 6-17

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 4-6

More About

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

Create a Function to Check Multiple Systems

You can use the `ModelAdvisor.run` function to programmatically run checks on one or more models or subsystems. This example shows how to create a function that runs Model Advisor checks on multiple subsystems and then returns the number of failures and warnings.

This example also describes how you can modify the function to check multiple models or subsystems in parallel. If you have the Parallel Computing Toolbox™ license, you can run this function in parallel mode to reduce the processing time.

Write a Function to Run Checks and Return Results

1. In the MATLAB® window, select **New > Function**.

2. Save the file as `run_configuration.m`.

3. In the function, right-click on `untitled` and select **Replace function name by file name**. The function name is updated to `run_configuration`.

```
function [outputArg1,outputArg2] = run_configuration(inputArg1,inputArg2)
%UNTITLED Summary of this function goes here
%    Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

4. Delete the body of the function.

```
function [outputArg1,outputArg2] = run_configuration(inputArg1,inputArg2)
end
```

5. Replace the output arguments with `[fail,warn]` and the input argument with `SysList`.

```
function [fail,warn] = run_configuration(SysList)
end
```

6. Inside the function, specify the Model Advisor configuration file.

```
fileName = 'myCheckConfiguration.json';
```

7. The `SysList` input is a list of systems for the Model Advisor to run checks on. Call the `ModelAdvisor.run` function on `SysList`.

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

8. Determine the number of checks that return failures and warnings and output them to the `fail` and `warn` output arguments, respectively:

```
fail = 0;
warn = 0;

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

The `run_configuration` function now contains this content:

```
function [fail, warn] = run_configuration(SysList)

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

fileName = 'myCheckConfiguration.json';

% Run the Model Advisor.
SysResultObjArray = ModelAdvisor.run(SysList,'Configuration', fileName);

fail = 0;
warn = 0;

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

end
```

Test the Function

1. Save the `run_configuration` function.

2. Create `sl_customization` file that is necessary for the custom configuration of checks in this example. For more information about custom configuration files, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

```
copyfile customizationFile.m sl_customization.m f
```

3. Save the subsystems that you want to run Model Advisor checks on to a variable called `systems`.

```
systems = {'sldemo_auto_climatecontrol/Heater Control',...
    'sldemo_auto_climatecontrol/AC Control',...
    'sldemo_auto_climatecontrol/Interior Dynamics'};
```

4. Run the `run_configuration` function on `systems`.

```
[fail,warn] = run_configuration(systems);

        Running Model Advisor...
Updating Model Advisor cache...
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.Manager

        Systems passed: 0 of 3

        Systems with warnings: 3 of 3

        Systems failed: 0 of 3
        Summary Report
```

4. Review the results by using the Summary Report or the `disp` function:

- To view the Model Advisor reports for each system, click the Summary Report link. This opens the Model Advisor Command-Line Summary report.

- To view the number of failures and warnings returned by the `run_configuration` function, look at the `fail` and `warn` variables.

```
disp(['Number of checks that return failures: ', num2str(fail)]);

Number of checks that return failures: 0

disp(['Number of checks that return warnings: ', num2str(warn)]);

Number of checks that return warnings: 5
```

Checking Multiple Systems in Parallel

Checking multiple systems in parallel reduces the processing time required by the Model Advisor. If you have a Parallel Computing Toolbox™ license, you can check multiple systems in parallel on a multicore host machine.

To check multiple systems in parallel, call the `ModelAdvisor.run` function with `'ParallelMode'` set to `'On'`.

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

The Parallel Computing Toolbox does not support 32-bit Windows® machines.

Each parallel process runs checks on one model at a time. When the Model Advisor runs in parallel mode, it does not support model data in the base workspace. Model data must be defined in the model workspace or data dictionary.

See Also
`ModelAdvisor.run` | `ModelAdvisor.setDefaultConfiguration`

Related Examples
- "Checking Systems Programmatically" on page 4-2
- "Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration" on page 7-7

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, open the example model `sldemo_auto_climatecontrol`.

```
openExample('sldemo_auto_climatecontrol')
```

Run the Model Advisor as follows:

```
SysResultObjArray = ModelAdvisor.run('sldemo_auto_climatecontrol/Heater Control',...
'mathworks.maab.jc_0021','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: 10-Sep-2021 16:51:32
Configuration: None
System: sldemo_auto_climatecontrol/Heater Control
System version: 10.4
Created by: The MathWorks, Inc.
========================================================
(1) Warning: Check model diagnostic parameters [check ID: mathworks.maab.jc_0021]
--------------------------------------------------------
Summary:    Pass    Warning    Fail     Not Run
             0         1         0          0
========================================================


Systems passed: 0 of 1

Systems with warnings: 1 of 1

Systems failed: 0 of 1
Summary Report
```

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

View Results in Model Advisor Command-Line Summary Report

When you run the Model Advisor programmatically, a Summary Report link is displayed in the Command Window. Clicking this link opens the Model Advisor Command-Line Summary report.
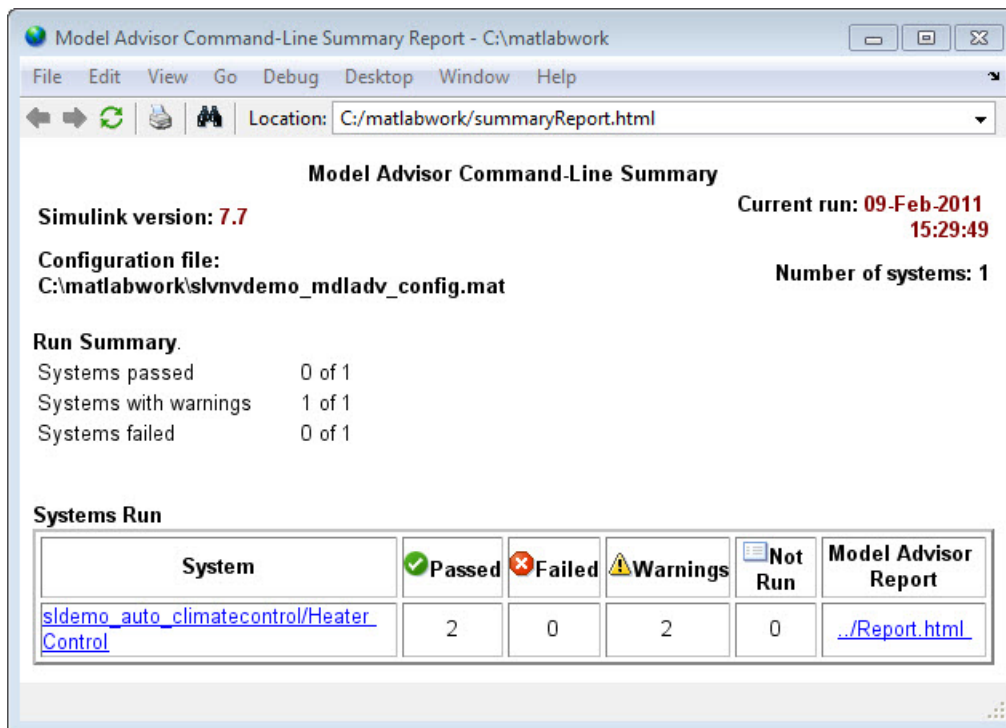
To view the Model Advisor Command-Line Summary report after loading an object, use the summaryReport function.

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

More About

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   Open the example model `sldemo_auto_climatecontrol`.

```
openExample('sldemo_auto_climatecontrol')
```

2   Call the `ModelAdvisor.run` function:

```
SysResultObjArray = ModelAdvisor.run('sldemo_auto_climatecontrol/Heater Control',...
'mathworks.maab.jc_0021');
```

3   Save the `SysResulObjArray` for use at a later time:

```
save my_model_advisor_run SysResultObjArray
```

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

```
clear
```

5   Load the results of the Model Advisor run:

```
load my_model_advisor_run SysResultObjArray
```

6   View the results in the Model Advisor:

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

See Also

`ModelAdvisor.run`

Related Examples

- "Archive and View Results" on page 4-6

Model Metrics

Collect and Explore Metric Data by Using the Metrics Dashboard

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

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

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

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

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

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

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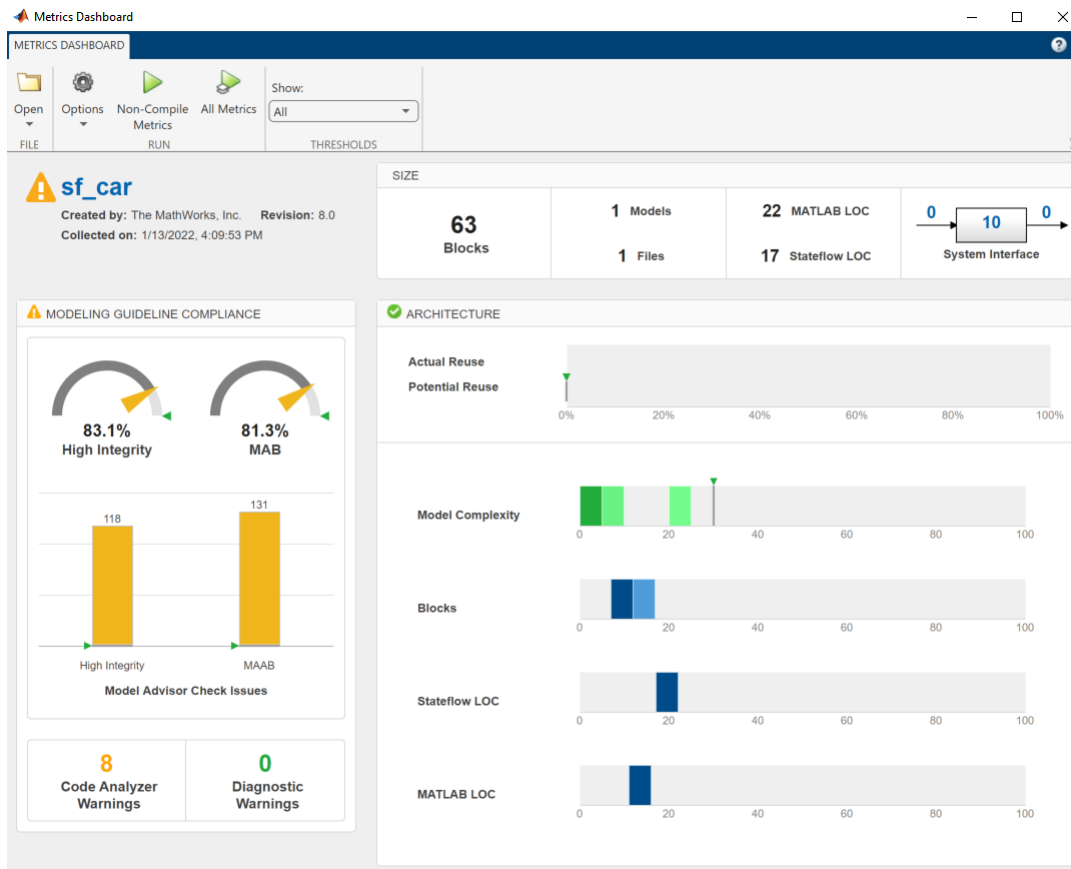
Metrics Dashboard Widgets

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

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

- You can view results in either a **Table** or **Tree** view. For the **High Integrity** and **MAB** compliance widgets, you can also choose a **Grid** view. To view highlighted results, in the grid view, click a cell.

- To sort the results by value or aggregated value, click the corresponding value column header.

- For metrics other than the **High Integrity** and **MAB** compliance widgets, you can filter results. To filter results, in the **Table** view, select the context menu on the right side of the **TYPE**, **COMPONENT**, and **PATH** column headers. From the **TYPE** menu, select applicable components. From the **COMPONENT** and **PATH** menus, type a component name or path in the search bar. The Metrics Dashboard saves the filters for a widget, so you can view metric details for other widgets and return to the filtered results.

- In the **Table** and **Tree** view, a value or aggregated value of n/a indicates that results are not available for that component. If the value and aggregated value are n/a, the **Table** view does not list the component. The **Tree** view does list such a component.

- 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 5-36.

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

Widget	Metric	Drill-In Data
Stateflow LOC	**Effective lines of code for Stateflow blocks** (mathworks.metrics.StateflowLOCCount)	Effective lines of code for Stateflow blocks by component
System Interface	• **Input and Output count** (mathworks.metrics.ExplicitIOCount) • **Parameter count** (mathworks.metrics.ParameterCount)	• Number of inputs and outputs by component (includes trigger ports) • Number of parameters by component

Modeling Guideline Compliance

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

Widget	Metric	Drill-In Data
High Integrity Compliance	**Model Advisor standards check compliance - High Integrity** (mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178)	For each component: • Percentage of checks passed • Status of each check Integration with the Model Advisor for more detailed results.
MAB Compliance	**Model Advisor standards check compliance - MAB** (mathworks.metrics.ModelAdvisorCheckCompliance.maab)	For each component: • Percentage of checks passed • Status of each check Integration with the Model Advisor for more detailed results.
High Integrity Check Issues	**Model Advisor standards issues - High Integrity** (mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178)	• Number of compliance check issues by component (see the following Note below). • Components without issues or aggregated issues are not listed.
MAB Check Issues	**Model Advisor standards issues - MAB** (mathworks.metrics.ModelAdvisorCheckIssues.maab)	• Number of compliance check issues by component (see the following Note below). • Components without issues or aggregated issues are not listed.
Code Analyzer Warnings	**Warnings from MATLAB Code Analyzer** (mathworks.metrics.MatlabCodeAnalyzerWarnings)	Number of Code Analyzer warnings by component.

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 5-36.

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

Widget	Metric	Drill-In Data
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 5-36.

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.

- The Metrics Dashboard does not count MAB checks that are not about blocks as issues. Examples include checks that warn about font formatting or file names. In the Model Advisor Check Issues widget, the tool might report zero MAB issues, but still report issues in the MAB Modeling Guideline Compliance widget. For more information about these issues, click the MAB Modeling Guideline Compliance widget.

See Also

More About

- "Collect Model Metrics Programmatically" on page 5-14
- "Model Metrics"
- "Collect Compliance Data and Explore Results in the Model Advisor" on page 5-24
- "Collect Metric Data Programmatically and View Data Through the Metrics Dashboard" on page 5-27

Collect Model Metrics Using the Model Advisor

You can use the Model Advisor metrics to analyze the size, complexity, and readability of your model.

The results of these metrics can help you verify compliance with industry standards and guidelines.

You can run model metrics in the Model Advisor **By Task > Model Metrics** subfolder.

This example uses the `sldemo_fuelsys` model to demonstrate how to collect these metrics.

1   Open the model `sldemo_fuelsys`. In the MATLAB Command Window, enter:

```
openExample('sldemo_fuelsys')
```

2   In the model window, open the **Modeling** tab and click **Model Advisor**. A System Selector dialog box opens. Click **OK**.

3   In the left pane of the Model Advisor, navigate to **By Task > Model Metrics**. Select the **Count Metrics**, **Complexity Metrics**, and **Readability Metrics** checks.
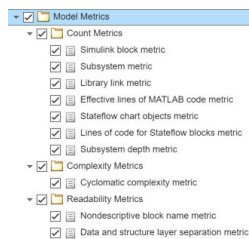
4   Right-click the **Model Metrics** folder and click **Run Selected Checks**.

5   After the Model Advisor runs the analysis, explore the metrics results by selecting a model metric in the Check Selector pane of the Model Advisor window. In the **Count Metrics** folder, select **Simulink block metric**. The **Report** tab shows a table for the number of blocks at the root

model level and subsystem level.

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

You can use the metric results to help compare your model to industry recommendations for model size, complexity, and readability.

See Also

More About

- "Model Metrics"
- "Model Metric Data Aggregation" on page 5-18
- "Collect Model Metrics Programmatically" on page 5-14
- "Create a Custom Model Metric for Nonvirtual Block Count" on page 5-11
- "Run Model Advisor Checks and Review Results" on page 3-4

Create a Custom Model Metric for Nonvirtual Block Count

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

Create Metric Class

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

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

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

2. Ensure that a custom `metricID` named `nonvirtualblockcount` is not already registered in the model metric repository.

```
slmetric.metric.unregisterMetric('nonvirtualblockcount');
slmetric.metric.refresh();
```

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. For this example, the file `nonvirtualblockcount_orig.m` contains the logic to create a metric that counts the nonvirtual blocks. Copy this file to the `nonvirtualblockcount.m` file.

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

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

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

Optionally, you can set these additional properties:

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

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

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

Collect Metric Data

To collect metric data on models, use instances of `slmetric.Engine`. Using the `getMetrics` method, specify the metrics you want to collect. For this example, specify the nonvirtual block count metric for the `sldemo_mdlref_conversion` model.

1. Load the `sldemo_mdlref_conversion` model.

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

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

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

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

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

Display and Export Results

To access the metrics for your model, use instance of `slmetric.metric.Result`. In this example, display the nonvirtual block count metrics for the `sldemo_mdlref_conversion` model. For each result, display the MetricID, ComponentPath, and Value.

```
for n=1:length(rc)
    if rc(n).Status == 0
        results = rc(n).Results;

        for m=1:length(results)
            disp(['MetricID: ',results(m).MetricID]);
            disp(['  ComponentPath: ', results(m).ComponentPath]);
            disp(['  Value: ', num2str(results(m).Value)]);
            disp(' ');
        end
    else
        disp(['No results for:',rc(n).MetricID]);
    end
    disp(' ');
end

MetricID: nonvirtualblockcount

  ComponentPath: sldemo_mdlref_conversion

  Value: 7


MetricID: nonvirtualblockcount

  ComponentPath: sldemo_mdlref_conversion/Bus Counter

  Value: 8
```

```
MetricID: nonvirtualblockcount

  ComponentPath: sldemo_mdlref_conversion/More Info

  Value: 0


MetricID: nonvirtualblockcount

  ComponentPath: sldemo_mdlref_conversion/SubSystem1

  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.createNewMetricClass` | `slmetric.metric.Result`

More About

- "Model Metrics"
- "Model Metric Data Aggregation" on page 5-18
- "Collect Model Metrics Programmatically" on page 5-14

Collect Model Metrics Programmatically

You can use the model metric API to programmatically collect model metrics that help you assess the architecture, complexity, and readability of your model. The results of these metrics can help you verify compliance with industry standards and guidelines.

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 the vdp model.

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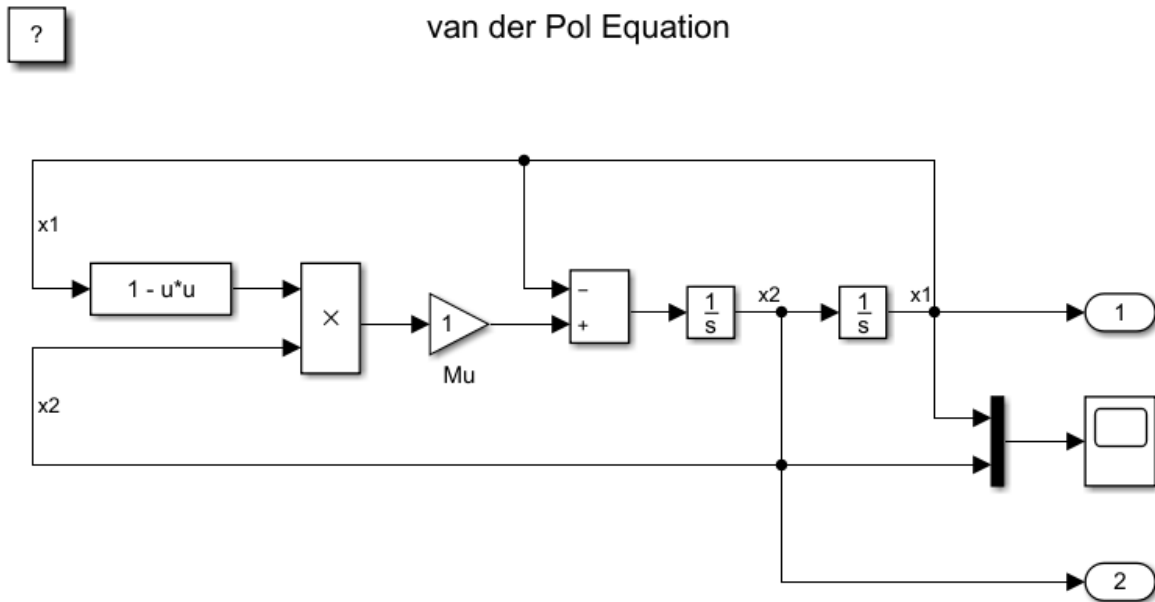

Collect Metrics

To collect metric data on a model, create an slmetric.Engine object and call execute.

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

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

Access Results

Use the `getMetrics` method to specify the metrics you want to collect. For this example, specify the block count and subsystem count metrics for the `vdp` model. `getMetrics` returns an array of `slmetric.metric.ResultCollection` objects.

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

Store and Display Results

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

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

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

```
MetricID: mathworks.metrics.SimulinkBlockCount

  ComponentPath: vdp

  Value: 13

MetricID: mathworks.metrics.SimulinkBlockCount

  ComponentPath: vdp/More Info

  Value: 1

MetricID: mathworks.metrics.SimulinkBlockCount

  ComponentPath: vdp/More Info/Model Info

  Value: 1

MetricID: mathworks.metrics.SimulinkBlockCount

  ComponentPath: vdp/More Info/Model Info/EmptySubsystem

  Value: 0
```

```
MetricID: mathworks.metrics.SubSystemCount

  ComponentPath: vdp

  Value: 1

MetricID: mathworks.metrics.SubSystemCount

  ComponentPath: vdp/More Info

  Value: 0

MetricID: mathworks.metrics.SubSystemCount

  ComponentPath: vdp/More Info/Model Info

  Value: 1

MetricID: mathworks.metrics.SubSystemCount

  ComponentPath: vdp/More Info/Model Info/EmptySubsystem

  Value: 0
```

Export Results

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

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

To export the metric results to an XML file, use the `exportMetrics` method. For each metric result, the XML file includes the `ComponentID`, `ComponentPath`, `MetricID`, `Value`, `AggregatedValue`, and `Measure`.

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

Close the `vdp` model.

```
bdclose(model);
```

Limitations

When you collect metric data, it is stored in a database file, `Metrics.db`, inside the simulation cache folder. You cannot collect metric data on one platform, move the database file to another platform, and then continue to collect additional metric data in that database file. For example, if you collect metric data on a Windows machine and then move the database file to a Linux machine, you cannot collect and store additional metric data in that database file. However, you are able to view that data in the Metrics Dashboard.

See Also
slmetric.Engine | slmetric.metric.Result | slmetric.metric.ResultCollection

More About

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

Model Metric Data Aggregation

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

How Model Metric Aggregation Works

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

- `Sum`: Returns the sum of the `Value` property and the `Value` properties of its children components across the component hierarchy. Returns the sum of the `Meaures` property and the `Measures` properties of its children components across the component hierarchy.

- `Max`: Returns the maximum of the `Value` property and the `Value` properties of its children components across the component hierarchy. Returns the maximum of the `Measures` property and the `Measures` properties of its children components across the component hierarchy.

- `None`: No aggregation of metric values.

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

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

- Model
- Subsystem block
- Chart
- MATLAB function block
- Protected model
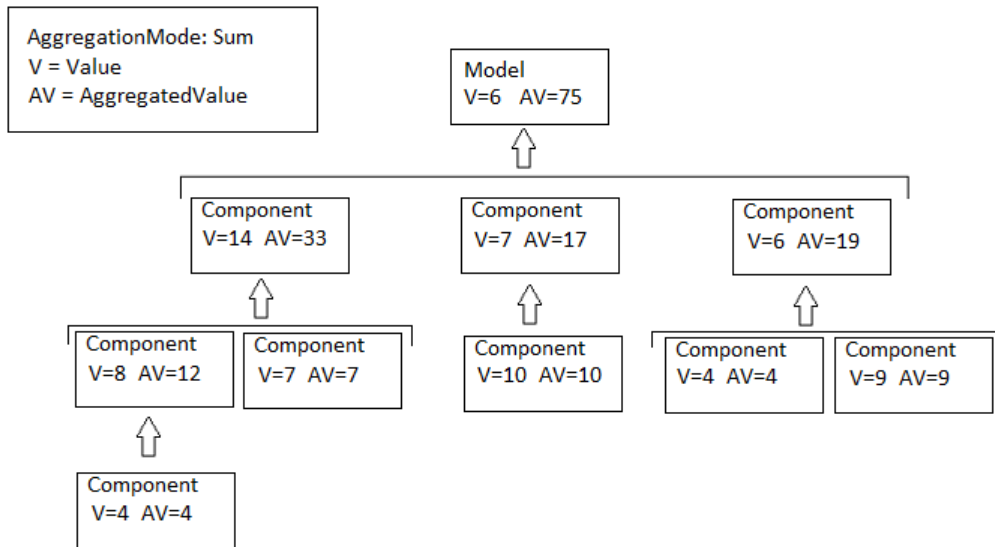
In the diagram, the `AggregationMode` is `Sum` and the model and components in the hierarchy each have a `Value` and an `AggregatedValue`. The `AggregatedValue` for a parent model or component is the sum of its `Value` and the `AggregatedValue` of each direct child component. For example, in this diagram, the `AggregatedValue` of the parent model is 75. The `AggregatedValue` of the parent model is calculated as the sum of the `Value` of the parent model, 6, plus the `AggregatedValue` of each direct child component, 33, 17, and 19.

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.

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

More About

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

Identify Modeling Clones with the Metrics Dashboard

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

Identify Clones

Open model ex_clone_detection.

open_system('ex_clone_detection.slx')
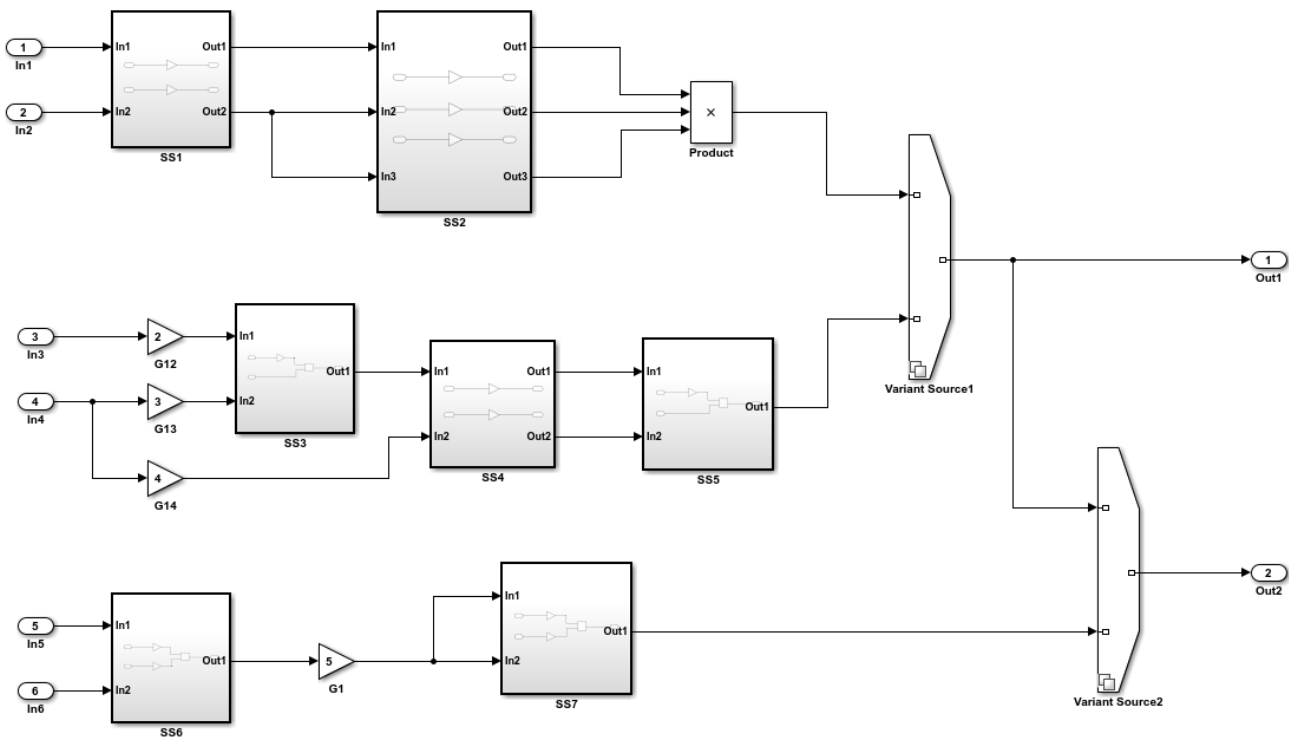
Copyright 2017 The MathWorks Inc.

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

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

Replace Clones with Links to Library Blocks

1   Open the Clone Detector app by clicking **Open Conversion Tool**. The Clone Detector app opens. For more information on the app, see "Enable Component Reuse by Using Clone Detection" on page 3-33.

2   Click **Replace Clones**. The Clone Detector app replaces the clones with links to library blocks. The library blocks are in the library specified by the **Library to place clones** parameter. This parameter is on the **Clone Results** tab. The library is on the MATLAB® path. It has a default name of `newLibraryFile`.

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

Run Model Metrics on the Refactored Model

1   Navigate to the **Metrics Dashboard**.

2   Click **All Metrics**.

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

See Also

More About

•   "Collect Model and Testing 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_system('sldemo_fuelsys');
```
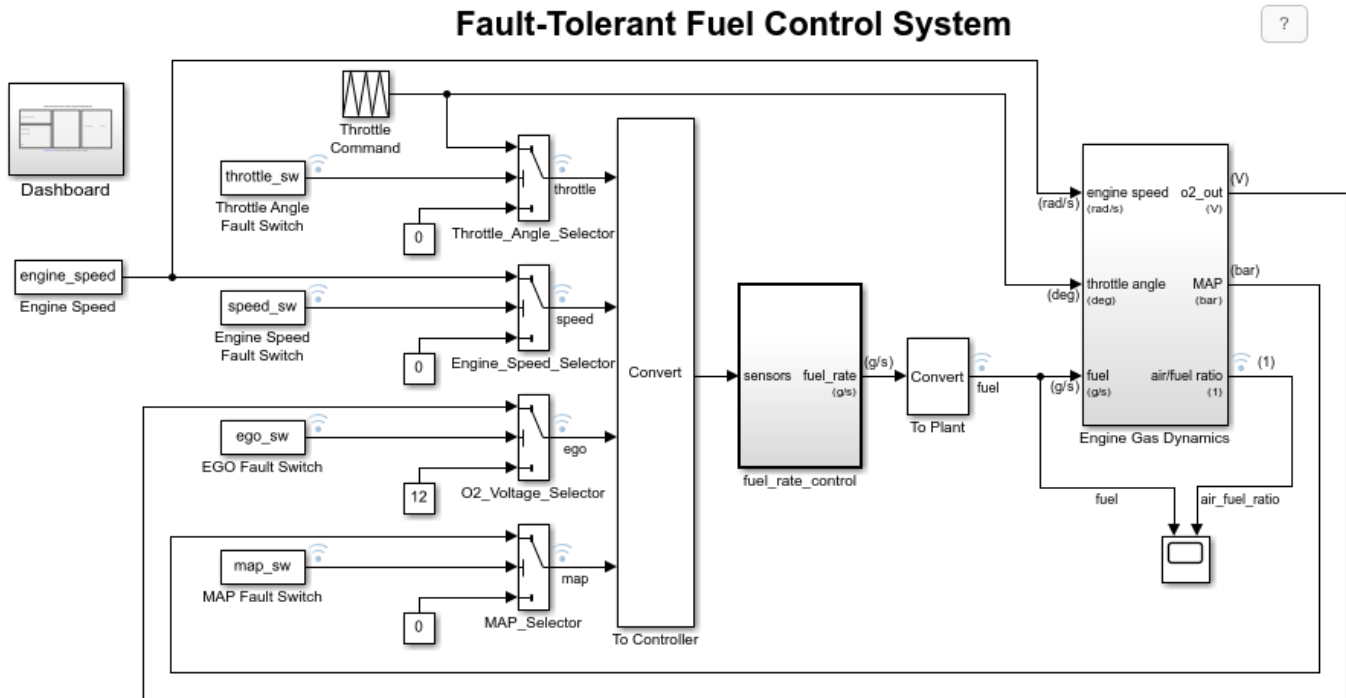

Fault-Tolerant Fuel Control System

Open the Dashboard subsystem to simulate any combination of sensor failures.   Copyright 1990-2017 The MathWorks, Inc.

Open the Metrics Dashboard

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

Collect Model Metrics

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

Explore Compliance Results

Locate the **MODELING GUIDELINE COMPLIANCE** section of the dashboard. This section displays the percentage of High Integrity and MAB compliance checks that passed on each system. The bar chart shows the number of issues reported by the checks in the corresponding check group.
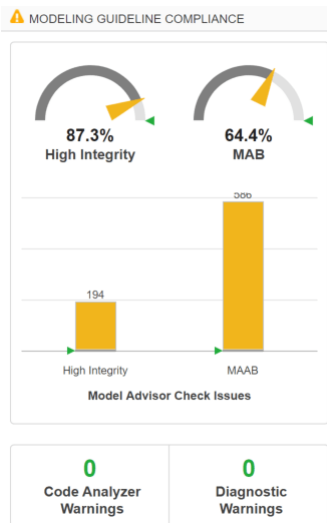
To see a table that details the number of compliance issues by component, click on the **High Integrity** bar in the bar chart. For compliance checks that analyze configuration settings, each check that does not pass adds one issue to the model on which it failed.

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.
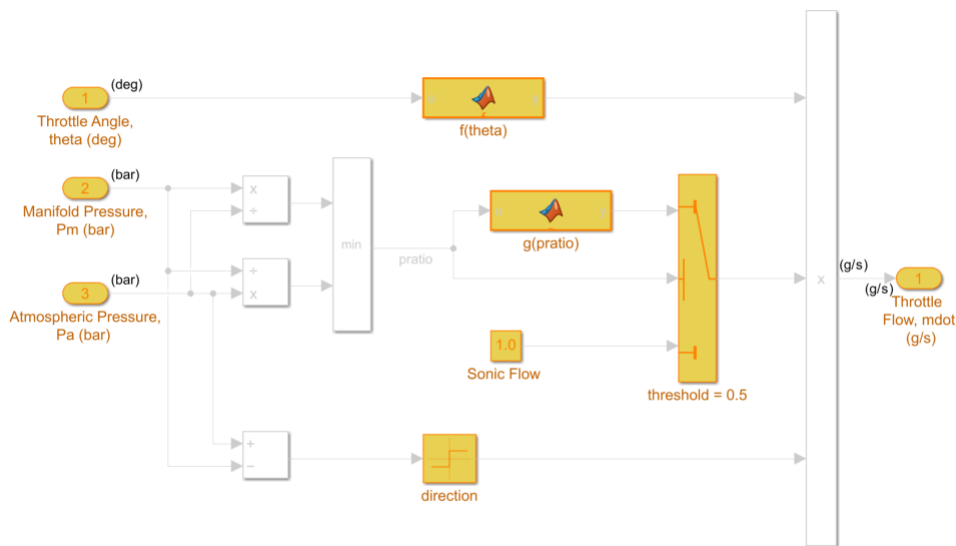
Throttle Flow vs. Valve Angle and Pressure

Explore Compliance Results in the Model Advisor

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

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

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

4   Expand the `sldemo_fuelsys` node.

5   To explore check results in more detail, click the **Check safety-related diagnostic settings for sample time** hyperlink.

6   In the Model Advisor Highlight dialog box, click **Check safety-related diagnostic settings for sample time** hyperlink.

Fix a Compliance Issue

1   In the Model Advisor Report, the check results show the Current Value and Recommended Value of diagnostic parameters.

2   To change the Current Value to the Recommended Value, click the parameter. The Model Configuration Parameters dialog box opens.

3   Change the parameter settings.

4   Save your changes and close the dialog box.

5   Save the changes to the model.

Recollect Metrics

1   Return to the Metrics Dashboard.

2   To recollect the model metrics, click the **All Metrics** icon.

3   To return to the main dashboard page, click the **Dashboard** icon.

4   Confirm that the number of **High Integrity** check issues is reduced.

See Also

More About

- "Collect and Explore Metric Data by Using the Metrics Dashboard" on page 5-2
- "Collect Model Metrics Programmatically" on page 5-14

Collect Metric Data Programmatically and View Data Through the Metrics Dashboard

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

Collect Metric Data Programmatically

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

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

Determine Model Compliance with MAB Guidelines

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

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

```
MAAB compliance: 64.4%
```

Open the Metrics Dashboard

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

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

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

Find the **MODELING GUIDELINE COMPLIANCE** section of the dashboard. For each category of compliance checks, the gauge indicates the percentage of compliance checks that passed.
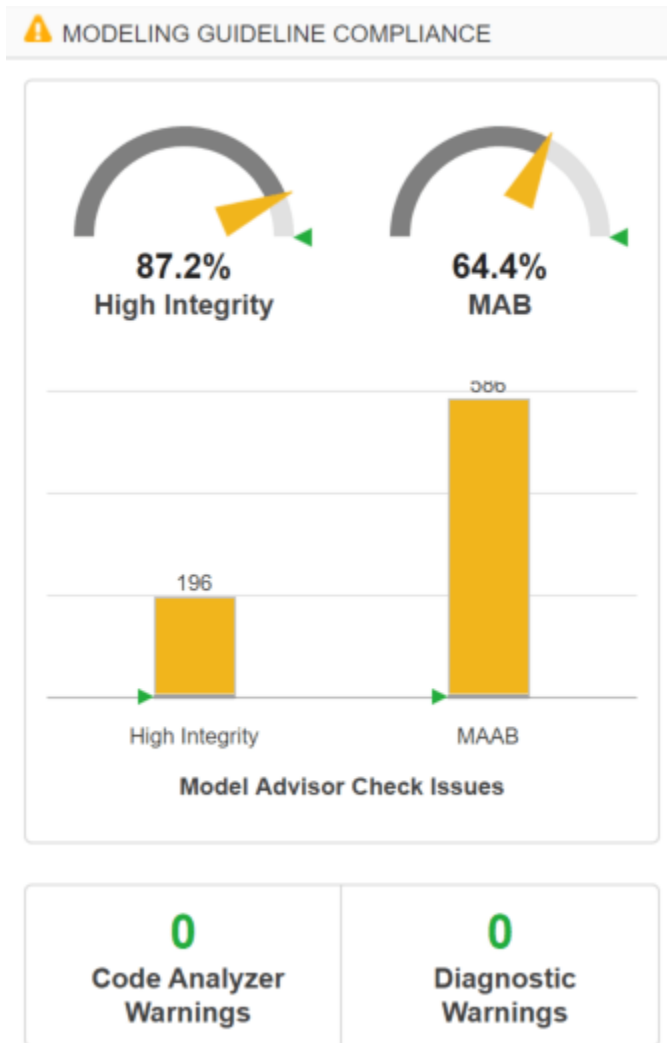
The dashboard reports the same MAB compliance percentage as the `slmetric.Engine` API reports.

Explore the MAB Compliance Results

Underneath the percentage gauges, the bar chart indicates the number of compliance check issues.

Click the MAAB bar in the bar chart to view a table of the Model Advisor Check Issues for MAB.

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

See Also

More About

- "Collect Model Metrics Programmatically" on page 5-14
- "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 files that you must provide:

- 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.

This 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

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

In addition to the files in the matlab:sldemo_slproject_airframe project, you must provide these additional files:

- 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".

- A MATLAB script that specifies metric thresholds and customizes the Metrics Dashboard. For more information on how to customize the Metrics Dashboard, see "Customize Metrics Dashboard Layout and Functionality" on page 5-36.

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

```
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 Shutdown**. For the **Startup files** field, specify the `setup.m` file.

- An `sl_customization.m` file that activates the Model Advisor configuration file to customize the Model Advisor checks. For more information on creating your own Model Advisor configuration, see Configure Compliance Metrics on page 5-0   .

- 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 Shutdown**. 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/index.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 main branch. For more information, see

   https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows.

3 On the left side bar, under **Settings** > **Repository**, protect the main branch by enforcing the use of merge requests when developers want to merge their changes into the main 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 TAP plugin to stream results to a `.tap` file. To enable communication of the 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 In the **Build Environment** section, select **Use MATLAB Version** and specify the **MATLAB root**, for example, `C:\Program Files\MATLAB\R2022a`.
6 In the **Build** section, 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)"
```

   For more information, see "Continuous Integration Using MATLAB Projects and Jenkins".

7 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 "Check Out from SVN Repository".

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" and "Pull, Push, and Fetch Files with Git".

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" and "Pull, Push, and Fetch Files with Git".

8 Push changes to the GitLab repository. See "Branch and Merge Files with Git" and "Pull, Push, and Fetch Files with Git".

9 In GitLab, create a merge request. Select the feature branch as source branch and the target branch as main. 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 **Create 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 main branch.

See Also

slmetric.config.setActiveConfiguration |
slmetric.dashboard.setActiveConfiguration

More About

- "Collect Model Metric Data by Using the Metrics Dashboard" on page 1-9
- "Collect and Explore Metric Data by Using the Metrics Dashboard" on page 5-2

External Websites

- https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html

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. For information on how to create a custom configuration file, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3. After you have set up your Model Advisor configuration, follow these steps to specify the check groups for which you want to obtain compliance and issues metric data:

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

```
vdp
```

2. Open the default configuration and save a corresponding `slmetric.config.Configuration` object to the base workspace.

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

3. Create a cell array, `values`, that specifies the Model Advisor **Check Group IDs** for MAAB, High-Integrity, and MISRA check groups.

- The value `maab` corresponds to a subset of the MAAB checks.
- The value `hisl_do178` corresponds to a subset of the High-Integrity System checks.
- The value `_SYSTEM_By Task_misra_c` is the **Check Group ID** for the MISRA check group `Modeling Standards for MISRA C:2012`.

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

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the folder that contains the desired group of checks. The **Check Group ID** is shown in the **Information** tab. For more information on the Model Advisor Configuration Editor, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

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. Obtain the compliance group from the layout.

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

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.
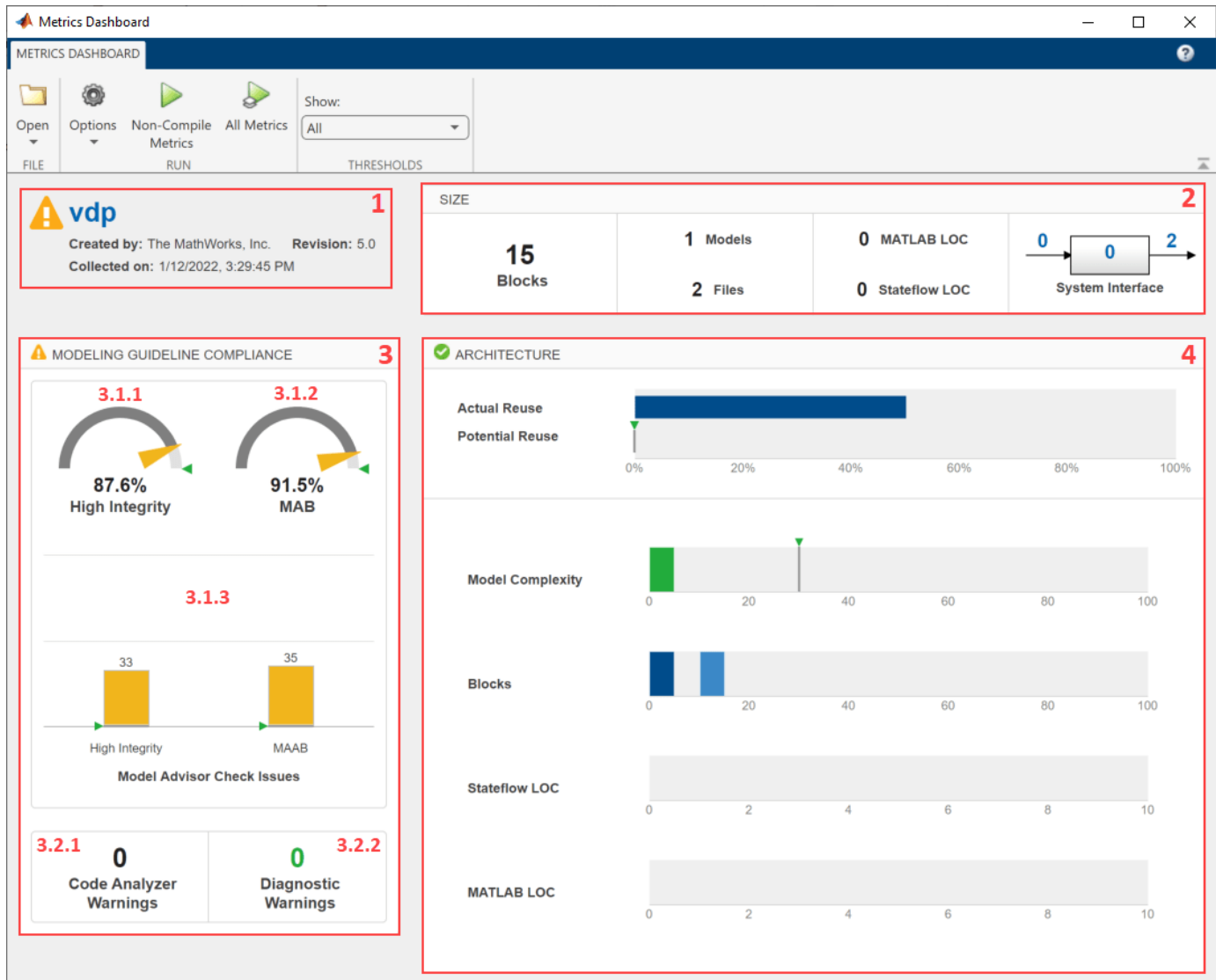
9. The modeling guideline compliance group contains two containers. The top container contains the **High Integrity** and **MAAB** compliance and check issues widgets. The red numbers 3.1.1, 3.1.2, and 3.1.3 indicate the order of the three widgets in the first container. The second container contains 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. The **Metric ID** for the configured MISRA check compliance metric is
`'mathworks.metrics.ModelAdvisorCheckCompliance._SYSTEM_By Task_misra_c'`.

```
misraComplianceMetricID = 'mathworks.metrics.ModelAdvisorCheckCompliance._SYSTEM_By Task_misra_c
```

The **Metric ID** for a configured check compliance metric is of the form **<Family ID>**.<Model Advisor **Check Group ID>**.

- Metrics configured for Model Advisor compliance use the **<Family ID>** `mathworks.metrics.ModelAdvisorCheckCompliance`. Configured check compliance metrics calculate the fraction of Model Advisor checks that pass for the selected Model Advisor **Check Group ID**.

- The Model Advisor **Check Group ID**, `_SYSTEM_By Task_misra_c`, is the **Check Group ID** for the MISRA check group `Modeling Standards for MISRA C:2012`.

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the folder that contains the desired group of checks. The **Check Group ID** is shown in the **Information** tab. For more information on the Model Advisor Configuration Editor, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

For more information on configured compliance metrics, see "Model Metrics".

11. Create a custom widget for visualizing MISRA check compliance metrics.

```
misraWidget = complianceContainers(1).addWidget('Custom', 1);
misraWidget.Title = ('MISRA');
misraWidget.VisualizationType = 'RadialGauge';
misraWidget.setMetricIDs(misraComplianceMetricID);
misraWidget.setWidths(slmetric.dashboard.Width.Medium);
```

12. The bar chart widget currently visualizes the High Integrity and MAAB check groups. Point this widget to the **Metric IDs** for the MISRA check issues and MAAB check issues.

```
misraIssuesMetricID = 'mathworks.metrics.ModelAdvisorCheckIssues._SYSTEM_By Task_misra_c';
maabIssuesMetricID = 'mathworks.metrics.ModelAdvisorCheckIssues.maab';

setMetricIDs(complianceContainerWidgets(3),...
({misraIssuesMetricID,maabIssuesMetricID}));
complianceContainerWidgets(3).Labels = {'MISRA', 'MAAB'};
```

The **Metric ID** for a configured check compliance metric is of the form **<Family ID>**.<Model Advisor **Check Group ID**>.

- Metrics configured for Model Advisor compliance issues use the **<Family ID>** `mathworks.metrics.ModelAdvisorCheckIssues`. Configured check compliance issues metrics calculate the number of issues reported by the selected Model Advisor **Check Group ID**.

- The Model Advisor **Check Group ID**, `_SYSTEM_By Task_misra_c`, is the **Check Group ID** for the MISRA check group `Modeling Standards for MISRA C:2012`. `maab` is a **Check Group ID** that corresponds to a subset of MAAB checks.

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the folder that contains the desired group of checks. The **Check Group ID** is shown in the **Information** tab. For more information on the Model Advisor Configuration Editor, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

For more information on configured compliance metrics, see "Model Metrics".

13. To run and view the Metrics Dashboard at this point in the example, enter the following lines of code in the MATLAB Command Window. The `save` commands serialize the API information to XML files. The `slmetric.config.setActiveConfiguration` and `slmetric.dashboard.setActiveConfiguration` commands set the active configuration objects.

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

14. To open the Metrics Dashboard, enter the following code in the MATLAB Command Window.

```
metricsdashboard vdp
```

15. Click the **All Metrics** button to run each of the metrics. The Metrics Dashboard displays results for the MISRA checks instead of the High Integrity checks.
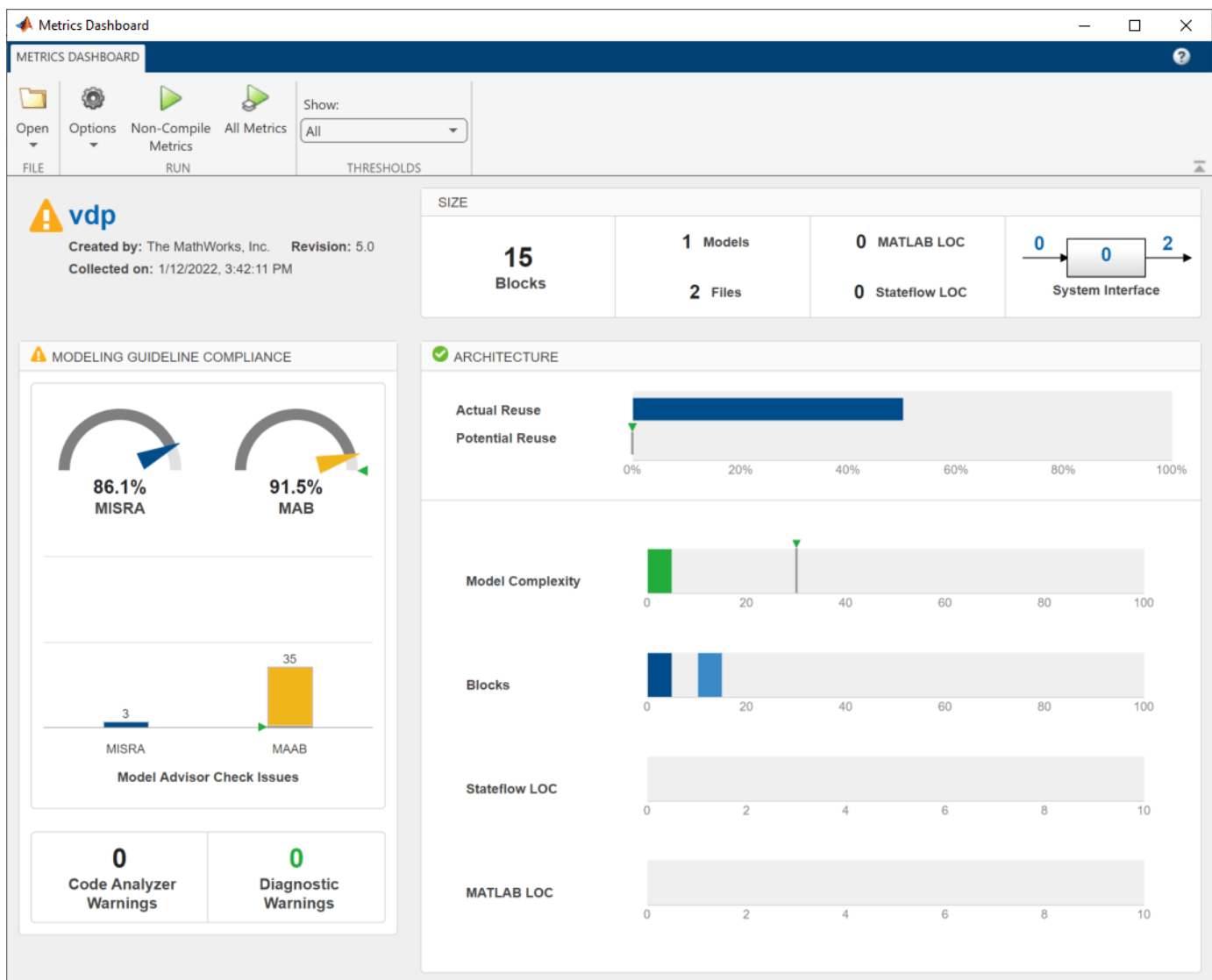
16. Close the Metrics Dashboard.

Add a Custom Metric to Dashboard

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

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

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

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

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

3. Register the new metric in the metric repository.

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

The new nonvirtual block count metric has the metric ID `nonvirtualblockcount`.

To view the available metrics for your metric engine, use `slmetric.metric.getAvailableMetrics`.

```
availableMetricIDs = slmetric.metric.getAvailableMetrics
```

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

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

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

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

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

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

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

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

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

8. To open the Metrics Dashboard, enter the following code in the MATLAB Command Window.

```
metricsdashboard vdp
```

9. Click the **All Metrics** button to run each of the metrics. The Metrics Dashboard displays results for the nonvirtual block count metric instead of the Simulink block count metric.
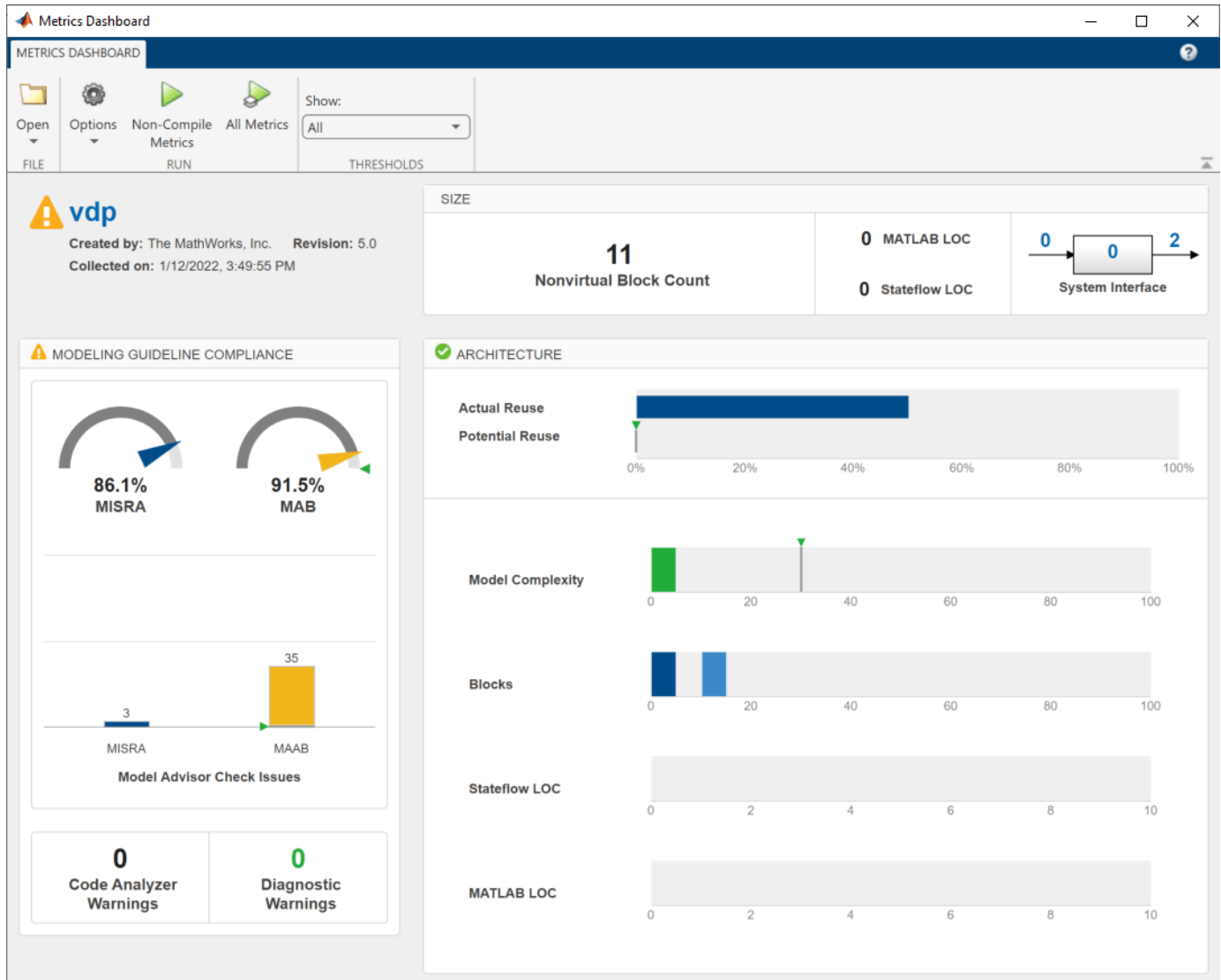
10. Close the Metrics Dashboard.

Add Metric Thresholds

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

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

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

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

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

```
T1 = addThreshold(TC, misraIssuesMetricID,'AggregatedValue');
C = getClassifications(T1);
C.Range.Start = -inf;
C.Range.End = 0;
C.Range.IncludeStart = 0;
C.Range.IncludeEnd = 1;

T2 = addThreshold(TC,misraComplianceMetricID,'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 model has MISRA check issues, the model status is warning. If the model does not have MISRA check 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 vdp
```
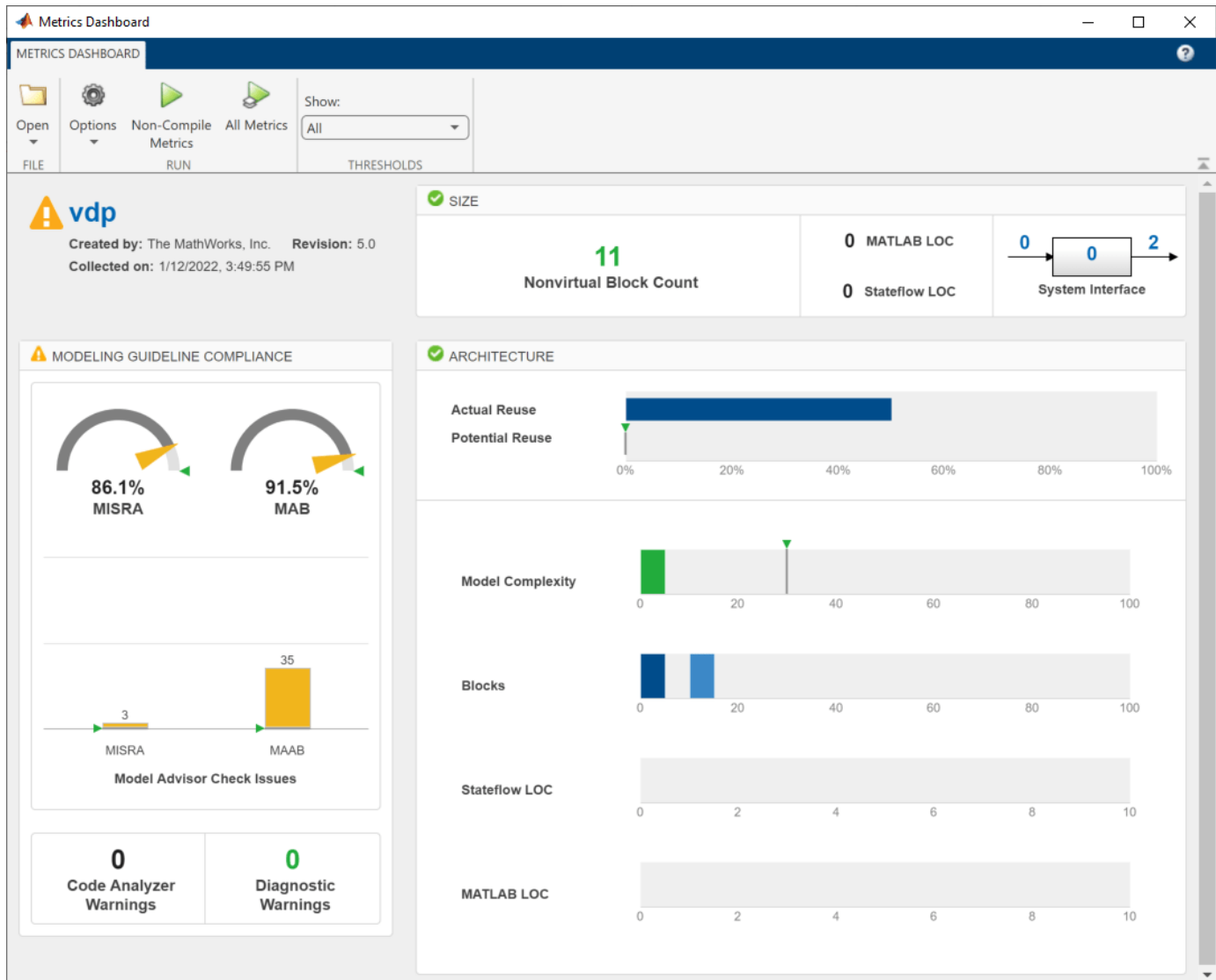
For the MISRA check compliance issues, the gauge is yellow because 86.1% of the checks pass. A percentage less than 100% generates a warning. The bar chart also displays a yellow because the model contains three MISRA check issues. A number greater than zero generates a warning.

The **Nonvirtual Block Count** widget is in the compliant range because there are 11 nonvirtual blocks.

8. To reset the configuration and unregister the metric, enter the following lines of code in the MATLAB Command Window.

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

See Also

slmetric.dashboard.Configuration | slmetric.config.Configuration

More About

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

Compare Model Complexity and Code Complexity Metrics

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

Metric Threshold Values

Code Complexity Threshold

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

Model Complexity Threshold

When you use the model-based design workflow to model an algorithm and generate code, you can assess the readability of the system by using the cyclomatic complexity metric of the model instead of measuring the cyclomatic complexity of the generated code. The graphical modeling of Simulink allows you to manage complex algorithms better than traditional hand code does. To account for this, the default cyclomatic complexity metric threshold for the model is 30, which is higher than the standard code complexity threshold of 10. To change the model metric threshold value, see "Customize Metrics Dashboard Layout and Functionality" on page 5-36.

Comparing Code and Model Complexity Metric Results

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

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

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

1 Open the project.

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

2 From the project, open the model folder and open `simulinkCruiseErrorAndStandardsExample`.
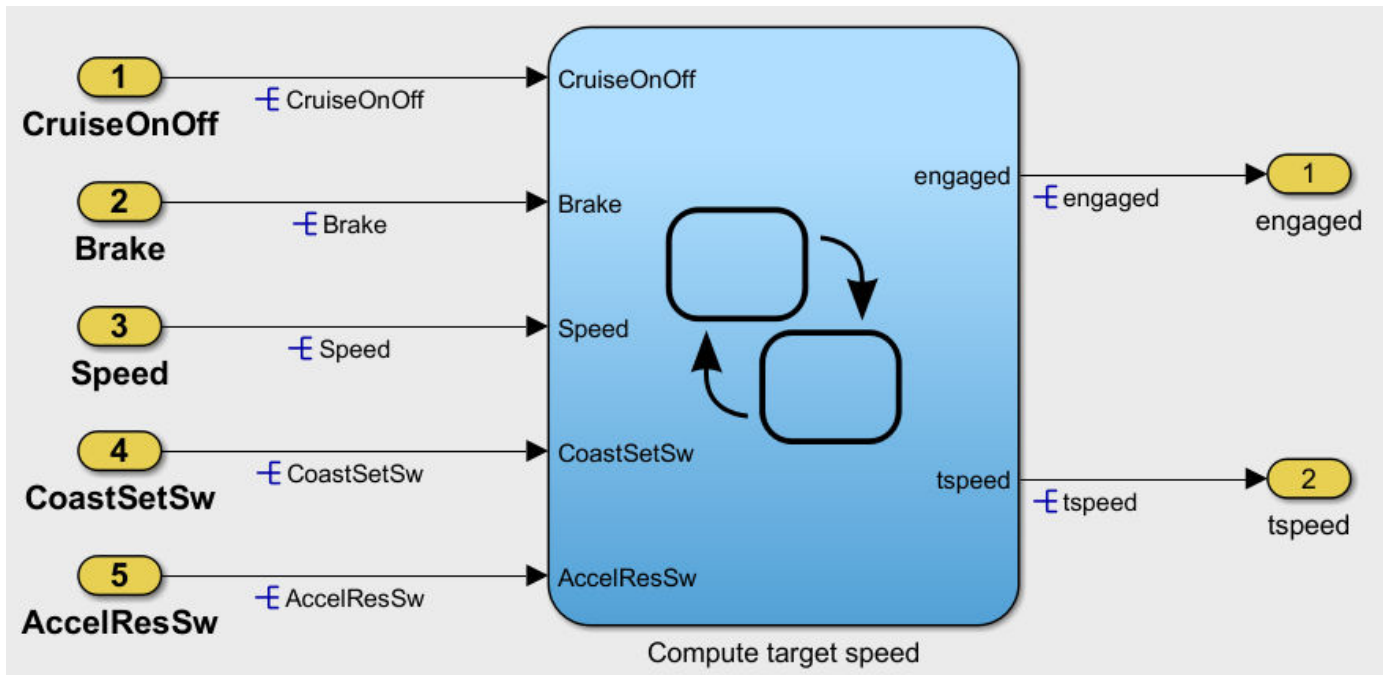
The model contains the chart Compute target speed. To generate and analyze code for the chart by using Polyspace, see "Analyze Code and Test Software-in-the-Loop" on page 2-12.
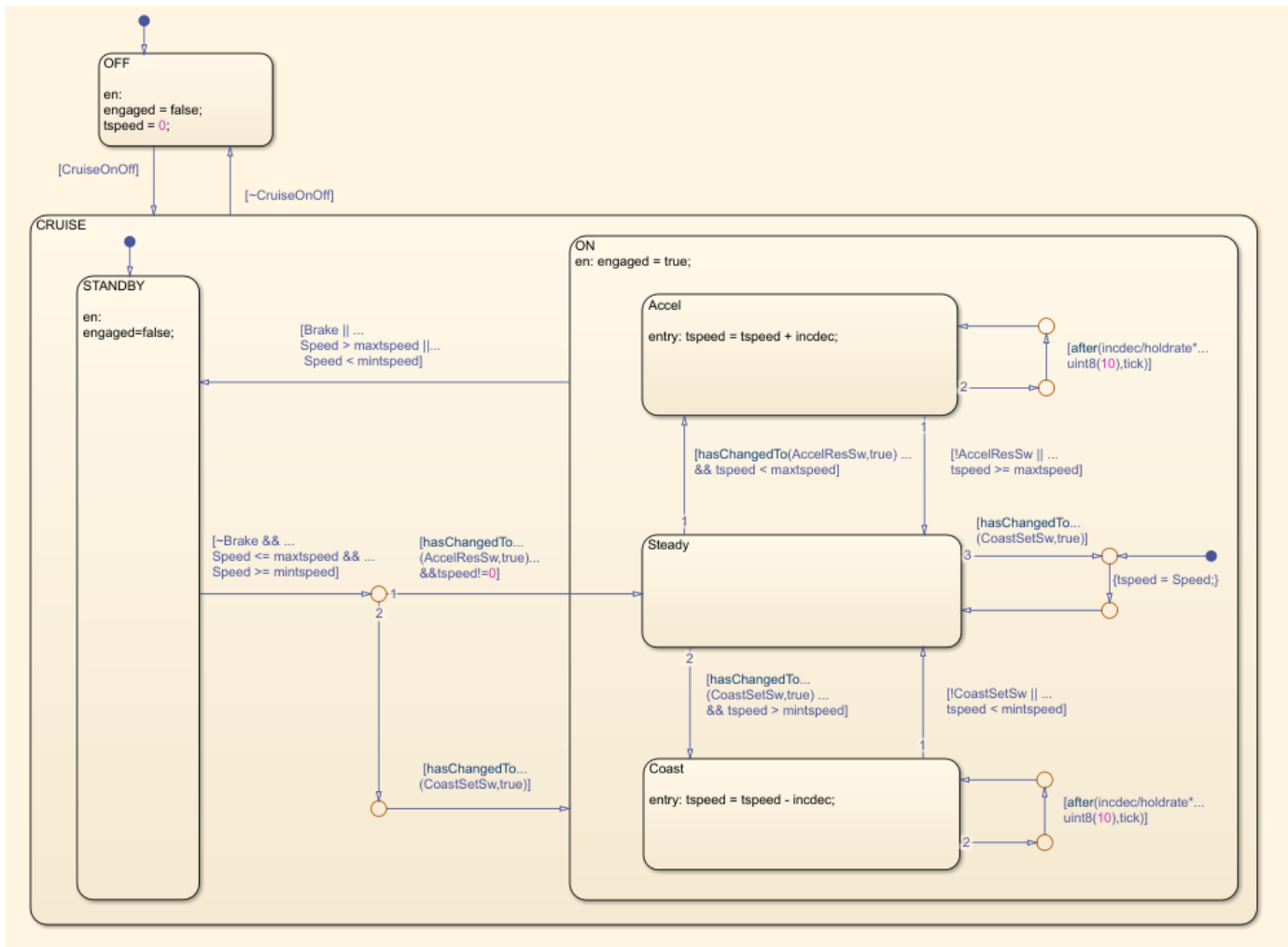
The reports that Polyspace generates for the code include code metrics such as cyclomatic complexity. The generated step function for the chart has a cyclomatic complexity of 20.

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

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

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

See Also
"Cyclomatic complexity metric"

More About

- *"Analyze Code and Test Software-in-the-Loop"* on page 2-12

Explore Status and Quality of Testing Activities Using the Model Testing Dashboard

The Model Testing Dashboard collects metric data from the model design and testing artifacts in a project to help you assess the status and quality of your requirements-based model testing.

The dashboard analyzes the artifacts in a project, such as requirements, models, and test results. Each metric in the dashboard measures a different aspect of the quality of the testing of your model and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C.

This example shows how to assess the testing status of a unit by using the Model Testing Dashboard. If the requirements, models, or tests in your project change, use the dashboard to assess the impact on testing and update the artifacts to achieve your testing goals.

Explore Testing Artifacts and Metrics for a Project

Open the project that contains the models and testing artifacts. For this example, in the MATLAB® Command Window, enter:

```
dashboardCCProjectStart('incomplete')
```

Open the Model Testing Dashboard by using one of these approaches:

- On the **Project** tab, click **Model Testing Dashboard**.
- In the Command Window, enter:

```
modelTestingDashboard
```

The first time that you open the dashboard for a project, the dashboard must identify the artifacts in the project and collect traceability information.
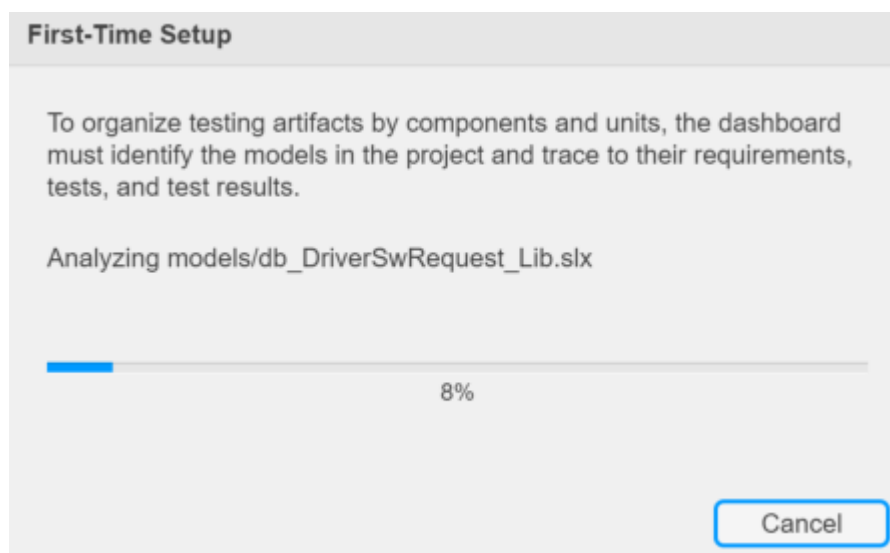
The dashboard displays metric results for the unit you select in the **Artifacts** panel.

Click the unit **db_DriverSwRequest** to view its metric results. When you initially select a unit in the **Artifacts** panel, the dashboard automatically collects the metric results for the unit. If you want to collect metrics for each of the units in the project, click **Collect > Collect All**.

If metric data was previously collected for a unit, the dashboard populates with the existing data. Collecting data for a metric requires a license for the product that supports the underlying artifacts, such as Requirements Toolbox™, Simulink® Test™, or Simulink Coverage™. Once metric results have been collected, viewing the results requires only a Simulink® Check™ license. For more information, see "Model Testing Metrics".
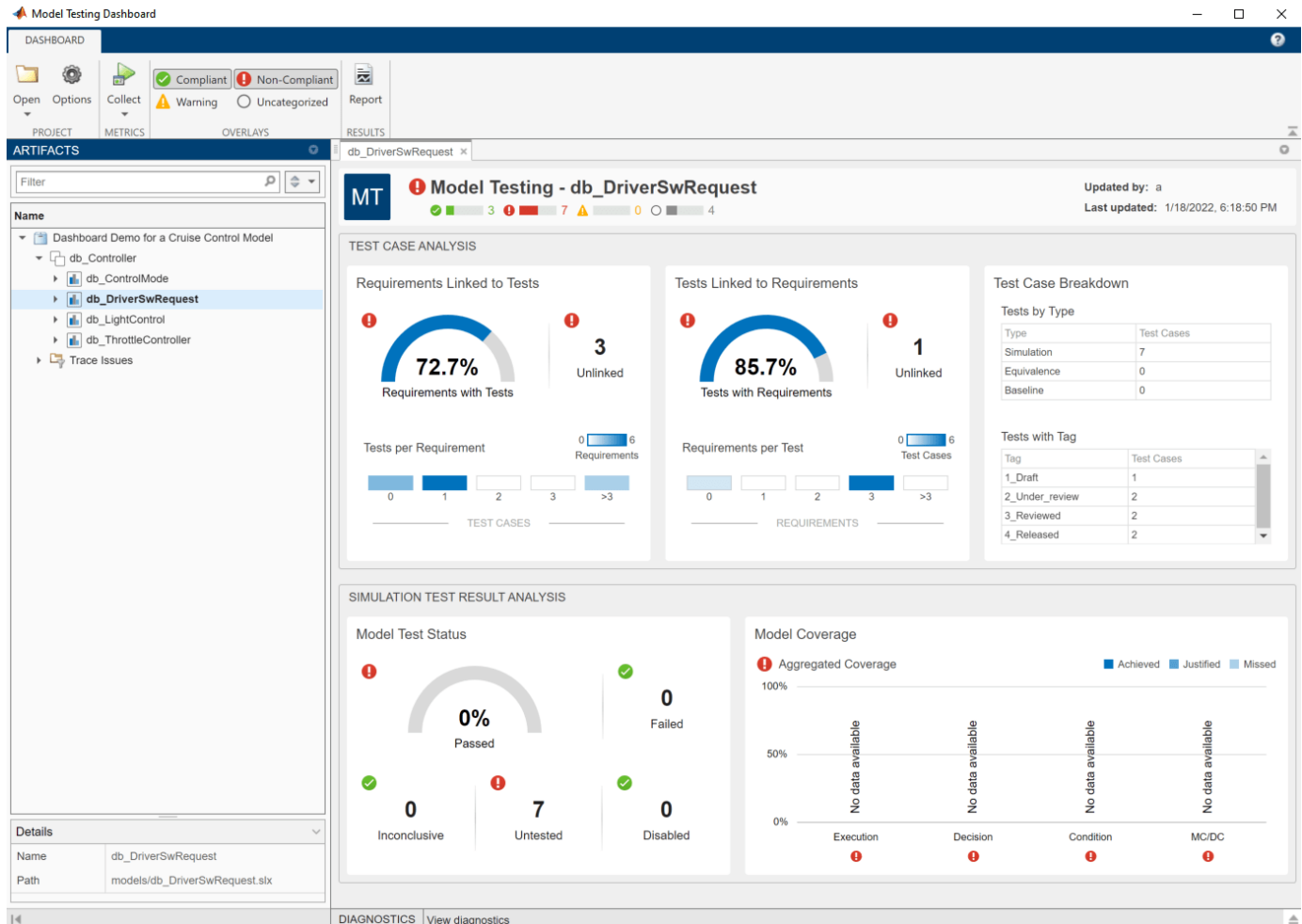
View Traceability of Design and Testing Artifacts

The **Artifacts** panel organizes the units in the project under the component models according to the model reference hierarchy. Expand a unit to see the artifacts in the project that trace to it.

For this example, in the **Artifacts** panel, expand the folder for the unit db_DriverSwRequest and its subfolders.
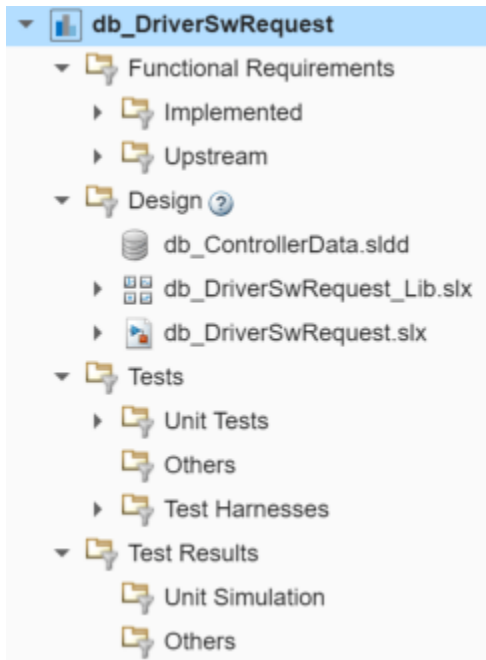
For each unit in the project, the traced artifacts include:

Functional Requirements

Requirements of **Type** `Functional` that are either implemented by or upstream of the unit. Use the Requirements Toolbox to create or import the requirements in a requirements file (`.slreqx`).

- **Implemented** — Functional requirements that are directly linked to the unit with a link **Type** of `Implements`. The dashboard uses these requirements in the metrics for this unit.
- **Upstream** — Functional requirements that are indirectly or transitively linked to the implemented requirements. The dashboard does not use these requirements in the metrics for this unit.

Design

The model file that contain the unit that you test and the libraries and data dictionaries that the model uses.

Tests

Test cases and test harnesses that trace to the unit. Create the test cases in a test suite file by using Simulink Test.

- **Unit Tests** — Test cases that the dashboard considers as unit tests. A unit test directly tests either the entire unit model or the model subsystems. The dashboard uses these tests in the metrics for this unit.
- **Others** — Test cases that trace to the unit but the dashboard does not consider as unit tests. For example, the dashboard does not consider tests on a library to be unit tests. The dashboard does not use these tests in the metrics for this unit.
- **Test Harnesses** — External test harnesses that trace to the unit or unit subsystems. Double-click a test harness to open it.

Test Results

Results of the test cases for the unit. To use the results in the dashboard, run the unit tests, export the results, and save them as a results file. The dashboard shows the latest saved results from the test cases.

- **Unit Simulation** — Simulation results from unit tests. The dashboard uses these results in the metrics for this unit.

- **Others** — Results that are not simulation results, are not from unit tests, or are only reports. For example, SIL results are not simulation results. The dashboard does not use these results in the metrics for this unit.

If there are changes to artifact files in the project, the dashboard detects the changes and automatically re-traces the artifacts to update the artifact traceability information shown in the **Artifacts** panel. You can turn the automatic re-tracing on or off by clicking **Collect > Auto Trace**.

An artifact appears under the folder **Trace Issues** if there are unexpected requirement links, requirement links which are broken or not supported by the dashboard, or artifacts that the dashboard cannot trace to a unit. The folder includes artifacts that are missing traceability and artifacts that the dashboard is unable to trace. If an artifact generates an error during traceability analysis, it appears under the **Errors** folder. For more information about artifact tracing issues and errors, see "Trace Artifacts to Units for Model Testing Analysis" on page 5-70.

Navigate to the requirement artifact for **Cancel Switch Detection**. Expand **db_DriverSwRequest > Functional Requirements > Implemented > db_SoftwareReqs.slreqx** and select the requirement **Cancel Switch Detection**.

At the bottom of the **Artifacts** panel, the **Details** pane displays the name of the artifact and the path to the artifact from the project root. You can scroll to a unit, collapse or expand the artifact list, or open a unit dashboard by right-clicking an artifact and selecting an action. You can also use the menu button ⬍ ▾ to the right of the search bar to perform these actions, restore the default view of the artifacts list, or view a legend of the dashboard icons.

View Metric Results for a Unit

You can collect and view metric results for each unit in the **Artifacts** panel. To view the results for the unit db_DriverSwRequest, in the **Artifacts** panel, click **db_DriverSwRequest**. When you click on a unit, the dashboard shows the **Model Testing** information for that unit. The top of the dashboard shows the name of the unit, the data collection timestamp, and the user name that collected the data. To open the results for multiple units at the same time, right-click a unit and click **Open unit dashboard in new tab**.

If artifacts in the project change after the results are collected, the dashboard detects this and shows a warning banner at the top of the dashboard to indicate that the metric results are stale.

⚠ Metric results are stale.   ↪ Collect   ⑦

The **Stale** icon ▢STALE appears on dashboard widgets that might show stale data which does not include the changes. If you see the warning banner, click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current

artifacts. You can also find the **Collect** button on the dashboard toolstrip in the **Metrics** section. For the unit in this example, the metric results in the dashboard are not stale.

The dashboard widgets summarize the metric data results and show testing issues you can address, such as:

- Missing traceability between requirements and tests
- Tests or requirements with a disproportionate number of links between requirements and tests
- Failed or disabled tests
- Missing model coverage

You can use the overlays in the Model Testing Dashboard to see if the metric results for a widget are compliant, non-compliant, or generate a warning that the metric results should be reviewed. Results are compliant if they show full traceability, test completion, or model coverage. In the **Overlays** section of the toolstrip, check that the **Compliant** and **Non-Compliant** buttons are selected. The overlay appears on the widgets that have results in that category. You can see the total number of widgets in each compliance category in the top-right corner of the dashboard.

To see the compliance thresholds for a metric, point to the overlay icon.
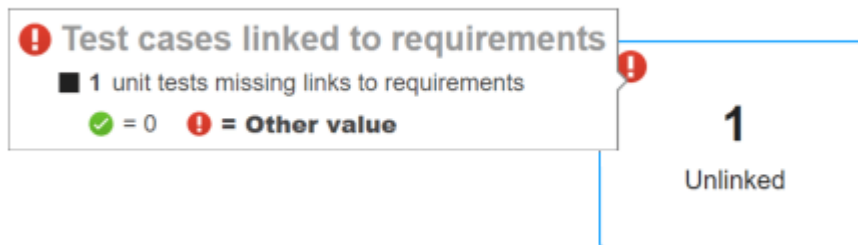
You can hide the overlay icons by clicking a selected category in the **Overlays** section of the toolstrip. For more information on the compliance thresholds for each metric, see "Model Testing Metrics".

To explore the data in more detail, click an individual metric widget to open the **Metric Details**. For the selected metric, a table displays a metric value for each artifact. The table provides hyperlinks to open the artifacts so that you can get detailed results and fix the artifacts that have issues. When exploring the tables, note that:

- You can filter the results by the value returned for each artifact. To filter the results, click the filter

  icon in the table header.

- By default, some widgets apply a filter to the table. For example, for the **Requirements Linked to Tests** section, the table for the **Unlinked** widget is filtered to only show requirements that are missing linked test cases. Tables that have filters show a check mark in the bottom right corner of

  the filter icon .

- To sort the results by artifact, source file, or value, click the corresponding column header.

Evaluate Testing and Traceability of Requirements

A standard measure of testing quality is the traceability between individual requirements and the test cases that verify them. To assess the traceability of your tests and requirements, use the metric data in the **Test Case Analysis** section of the dashboard. You can quickly find issues in the requirements and tests by using the data summarized in the widgets. Click a widget to view a table with detailed results and links to open the artifacts.

Requirements Missing Tests

In the **Requirements Linked to Tests** section, the **Unlinked** widget indicates how many requirements are missing links to test cases. To address unlinked requirements, create test cases that verify each requirement and link those test cases to the requirement. The **Requirements with Tests** gauge widget shows the linking progress as the percentage of requirements that have tests.

Requirements Linked to Tests

72.7%
Requirements with Tests

3
Unlinked
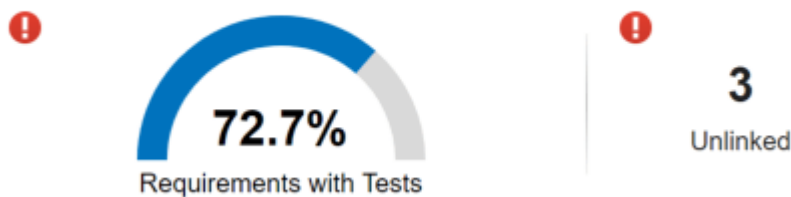
Click any widget in the section to see the detailed results in the **Requirement linked to test cases** table. For each requirement artifact, the table shows the source file that contains the requirement and whether the requirement is linked to at least one test case. When you click the **Unlinked** widget, the table is filtered to show only requirements that are missing links to test cases.

Requirements with Disproportionate Numbers of Tests

The **Tests per Requirement** section summarizes the distribution of the number tests linked to each requirement. For each value, a colored bin indicates the number of requirements that are linked to that number of tests. Darker colors indicate more requirements. If a requirement has a too many tests, the requirement might be too broad, and you may want to break it down into multiple more granular requirements and link each of those requirements to the respective test cases. If a requirement has too few tests, consider adding more test cases and linking them to the requirement.

Tests per Requirement

0   6
Requirements

0   1   2   3   >3

TEST CASES

To see the requirements that have a certain number of test cases, click the corresponding number to open a filtered **Test cases per requirement** table. For each requirement artifact, the table shows the source file that contains the requirement and the number of linked test cases. To see the results for each of the requirements, in the **Linked Test Cases** column, click the filter icon   , then select **Clear Filters**.

Tests Missing Requirements

In the **Tests Linked to Requirements** section, the **Unlinked** widget indicates how many tests are not linked to requirements. To address unlinked tests, add links from these test cases to the requirements they verify. The **Tests with Requirements** gauge widget shows the linking progress as the percentage of tests that link to requirements.

Tests Linked to Requirements

85.7%
Tests with Requirements

1
Unlinked
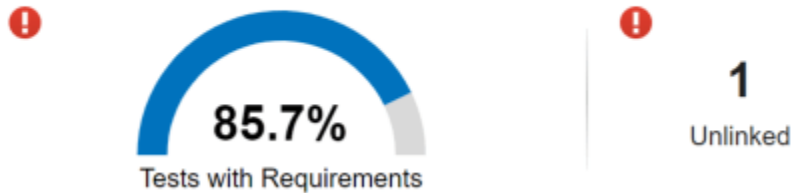
Click any widget in the section to see detailed results in the **Test case linked to requirements** table. For each test case artifact, the table shows the source file that contains the test and whether the test case is linked to at least one requirement. When you click the **Unlinked** widget, the table is filtered to show only test cases that are missing links to requirements.

Tests with Disproportionate Numbers of Requirements

The **Requirements per Test** widget summarizes the distribution of the number of requirements linked to each test. For each value, a colored bin indicates the number of requirements that are linked to that number of tests. Darker colors indicate more tests. If a test has too many or too few requirements, it might be more difficult to investigate failures for that test, and you may want to change the test or requirements so that they are easier to track. For example, if a test verifies many more requirements than the other tests, consider breaking it down into multiple smaller tests and linking them to the requirements.

Requirements per Test

0 [   ] 6
Test Cases

0   1   2   3   >3
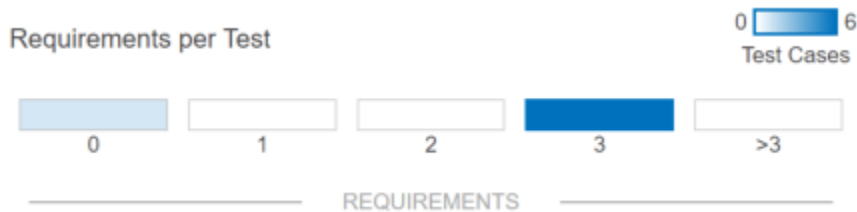
REQUIREMENTS

To see the test cases that have a certain number of requirements, click the corresponding bin to open the **Requirements per test case** table. For each test case artifact, the table shows the source file that contains the test and the number of linked requirements. To see the results for each of the test

cases, in the **Linked Requirements** column, click the filter icon ▼⊘ , then select **Clear Filters**.

Disproportionate Number of Tests of One Type

The **Tests by Type** and **Tests with Tags** widgets show how many tests the unit has of each type and with each custom tag. In industry standards, tests are often categorized as normal tests or robustness tests. You can tag test cases with `Normal` or `Robustness` and see the total count for each tag by using the **Tests with Tag** widget. Use the **Test Case Breakdown** to decide if you want to add tests of a certain type, or with a certain tag, to your project.

Test Case Breakdown

Tests by Type

Type	Test Cases
Simulation	7
Equivalence	0
Baseline	0

Tests with Tag

Tag	Test Cases	
1_Draft	1	
2_Under_review	2	
3_Reviewed	2	
4_Released	2	

To see the test cases of one type, click the corresponding row in the **Tests by Type** table to open the **Test case type** table. For each test case artifact, the table shows the source file that contains the test and the test type. To see results for each of the test cases, in the **Type** column, click the filter icon

, then select **Clear Filters**.

To see the test cases that have a tag, click the corresponding row in the **Tests with Tag** table to open the **Test case tags** table. For each test case artifact, the table shows the source file that contains the test and the tags on the test case. To see results for each of the test cases, in the **Tags** column, click

the filter icon , then select **Clear Filters**.

Analyze Test Results and Coverage

To see a summary of the test results and coverage measurements, use the widgets in the **Simulation Test Result Analysis** section of the dashboard. Find issues in the tests and in the model by using the test result metrics. Find coverage gaps by using the coverage metrics and add tests to address missing coverage. Run the tests for the model and collect the dashboard metrics to check for model testing issues.

Tests That Have Not Passed

In the **Model Test Status** section, the **Untested** and **Disabled** widgets indicate how many tests for the unit have not been run. Run the tests by using the Simulink Test Manager and export the new results.

The **Failed** widget indicates how many tests failed. Click on the **Failed** widget to view a table of the test cases that failed. Click the hyperlink for each failed test case artifact to open it in the Test Manager and investigate the artifacts that caused the failure. Fix the artifacts, re-run the tests, and export the results.

The **Inconclusive** widget indicates how many tests do not have pass/fail criteria such as verify statements, custom criteria, baseline criteria, and logical or temporal assessments. If a test does not contain pass/fail criteria, then it does not verify the functionality of the linked requirement. Add one or more of these pass/fail criteria to your test cases to help verify the functionality of your model.

Model Test Status
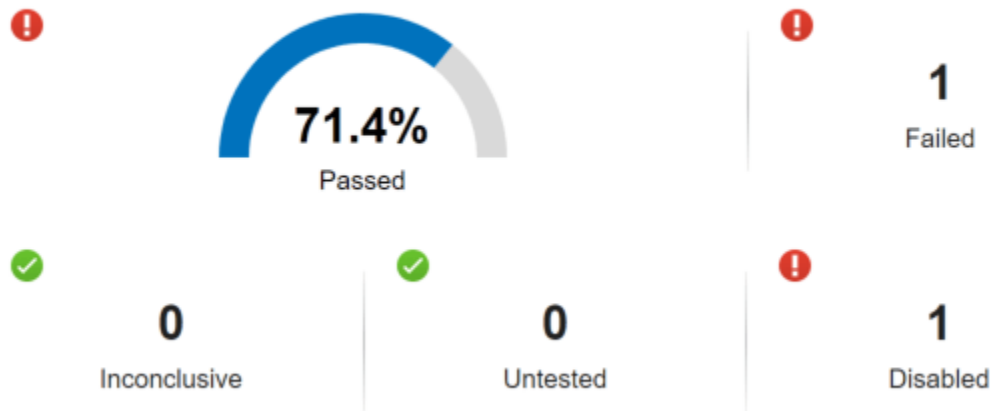
Click any widget in the section to open the **Test case status** table. For each test case artifact, the table shows the source file that contains the test and the status of the test result. When you click the **Failed**, **Disabled**, **Untested**, or **Inconclusive** widgets, the table is filtered to show only tests for those test case result statuses. The dashboard analyzes only the latest test result that it traces to each test case.

Missing Coverage

The **Model Coverage** widget shows whether there are model elements that are not covered by the tests. If one of the coverage types shows less than 100% coverage, you can click the dashboard widgets to investigate the coverage gaps. Add tests to cover the gaps or justify points that do not need to be covered. Then run the tests again and export the results. For more information on coverage justification, see "Fix Requirements-Based Testing Issues" on page 5-61.

Model Coverage
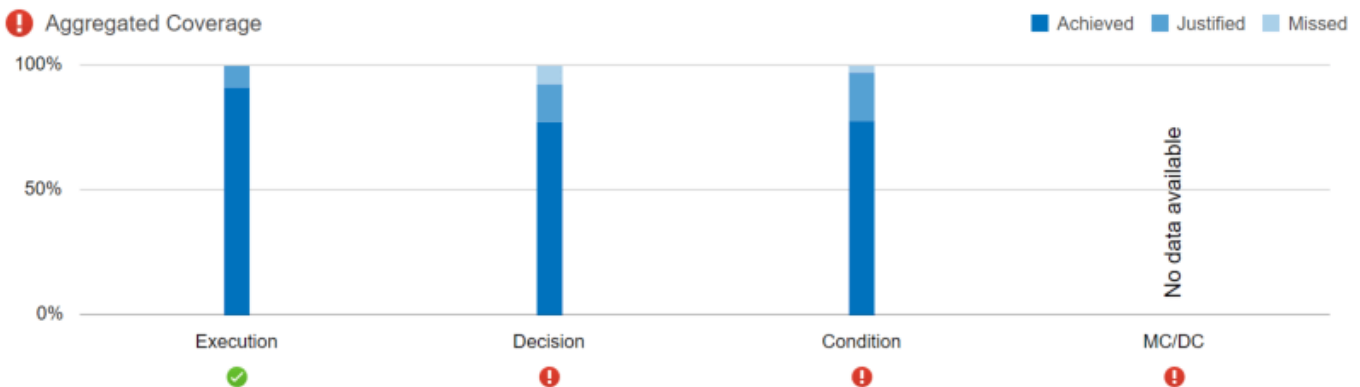
To see the detailed results for one type of coverage, click the corresponding bar. For the model and test case artifacts, the table shows the source file and the achieved and justified coverage.

See Also

"Model Testing Metrics"

More About

- "Fix Requirements-Based Testing Issues" on page 5-61
- "Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard" on page 5-96

Fix Requirements-Based Testing Issues

This example shows how to address common traceability issues in model requirements and tests by using the Model Testing Dashboard. The dashboard analyzes the testing artifacts in a project and reports metric data on quality and completeness measurements such as traceability and coverage, which reflect guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. The dashboard widgets summarize the data so that you can track your requirements-based testing progress and fix the gaps that the dashboard highlights. You can click the widgets to open tables with detailed information, where you can find and fix the testing artifacts that do not meet the corresponding standards.

Collect Metrics for the Testing Artifacts in a Project

The dashboard displays testing data for a model and the artifacts that the unit traces to within a project. For this example, open the project and collect metric data for the artifacts.

1   Open the project that contains the models and testing artifacts. For this example, in the MATLAB® Command Window, enter `dashboardCCProjectStart('incomplete')`.

2   Open the dashboard. To open the Model Testing Dashboard: on the **Project** tab, click **Model Testing Dashboard** or enter `modelTestingDashboard` at the command line.

3   In the **Artifacts** panel, the dashboard organizes unit models under the component models that contain them in the model hierarchy. Artifacts such as requirements, test cases, and test results appear under the units that they trace to. View the metric results for the unit `db_DriverSwRequest`. In the **Artifacts** panel, click the name of the unit, **db_DriverSwRequest**. When you initially select `db_DriverSwRequest`, the dashboard collects the metric results for uncollected metrics and populates the widgets with the data for the unit.
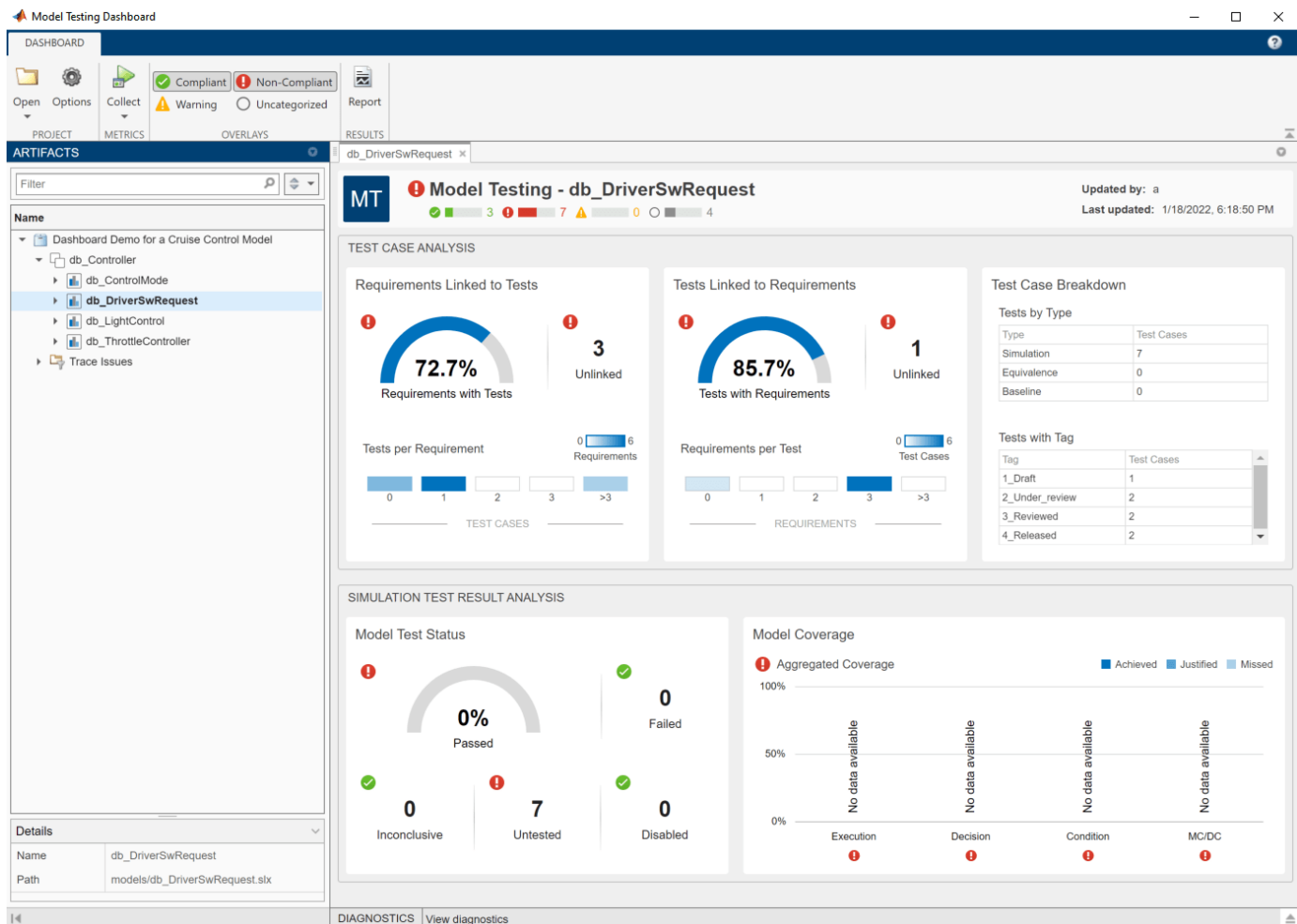
Link a Requirement to its Implementation in a Model

On the **Artifacts** panel, the **Trace Issues** folder shows artifacts that do not trace to unit models in the project. The **Trace Issues** folder contains subfolders for:

- **Unexpected Implementation Links** — Requirement links of **Type** `Implements` for a requirement of **Type** `Container` or **Type** `Informational`. The dashboard does not expect these links to be of **Type** `Implements` because container requirements and informational requirements do not contribute to the Implementation and Verification status of the requirement set that they are in. If a requirement is not meant to be implemented, you can change the link type. For example, you can change a requirement of **Type** `Informational` to have a link of **Type** `Related to`.

- **Unresolved and Unsupported Links** — Requirement links which are broken or not supported by the dashboard. For example, if a model block implements a requirement, but you delete the model block, the requirement link is now unresolved. If a requirement links to or from a data dictionary, the link is not supported and the Model Testing Dashboard does not trace the link. The Model Testing Dashboard does not support traceability analysis for some artifacts and some links. If you expect a link to trace to a unit and it does not, see the troubleshooting solutions in "Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard" on page 5-96.

- **Untraced Tests** — Tests that execute on models or subsystems that are not on the project path.

- **Untraced Results** — Results that the dashboard can no longer trace to a test case. For example, if a test case produces results, but you delete the test case, the results can no longer be traced to the test case.

Address Testing Traceability Issues

Open the dashboard for the unit **db_DriverSwRequest** by clicking the name of the unit in the **Artifacts** panel. The widgets in the **Test Case Analysis** section of the dashboard show data about the unit requirements, test cases for the unit, and links between them. The widgets indicate if there are gaps in testing and traceability for the implemented requirements.

Link Requirements and Test Cases

For the unit db_DriverSwRequest, the **Tests Linked to Requirements** section shows that some of the test cases are missing links to requirements in the model.

To see detailed information about the missing links, in the **Tests Linked to Requirements** section, click the widget **Unlinked**. The dashboard opens the **Metric Details** for the widget with a table of metric values and hyperlinks to each related artifact. The table shows the test cases that are implemented in the unit, but do not have links to requirements. The table is filtered to show only test cases that are missing links to requirements.

The test case Detect long decrement is missing linked requirements.

1. Open the test case in the Test Manager. In the **Artifact** column of the table, click **Detect long decrement**. For this example, the test case needs to link to three requirements that already exist in the project. If there were not already requirements, you could add a requirement by using the Requirements Editor.

2. Open the software requirements in the Requirements Editor. In the **Artifacts** panel, expand the folder **db_DriverSwRequest > Functional Requirements > Implemented** and double-click the requirement file **db_SoftwareReqs.slreqx**.

3. View the software requirements in the container with the summary **Driver Switch Request Handling**. Expand **db_SoftwareReqs > Driver Switch Request Handling**.

4. Select multiple software requirements. Hold down the **Ctrl** key as you click **Output request mode**, **Avoid repeating commands**, and **Long Increment/Decrement Switch recognition**. Keep these requirements selected in the Requirements Editor.

5. In the Test Manager, expand the **Requirements** section for the test case Detect long decrement. Click the arrow next to the **Add** button and select **Link to Selected Requirement**. The traceability link indicates that the test case Detect long decrement verifies the three requirements Output request mode, Avoid repeating commands, and Long Increment/Decrement Switch recognition.

6. The metric results in the dashboard reflect only the saved artifact files. To save the test suite db_DriverSwRequest_Tests.mldatx, in the **Test Browser**, right-click **db_DriverSwRequest_Tests** and click **Save**.

Refresh Metric Results in the Dashboard

The dashboard detects that the metric results are now stale and shows a warning banner at the top of the dashboard.

⚠ Metric results are stale.   📥 Collect   ⓘ

1. Click the **Collect** button on the warning banner to re-collect the metric data so that the dashboard reflects the traceability link between the test case and requirements.

2. View the updated dashboard widgets by returning to the **Model Testing** results. At the top of the dashboard, there is a breadcrumb trail from the **Metric Details** back to the **Model Testing** results. Click the breadcrumb button for **db_DriverSwRequest** to return to the **Model Testing** results for the unit.
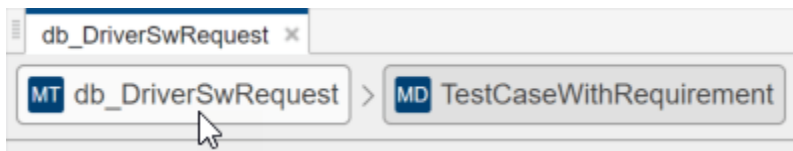
The **Tests Linked to Requirements** section shows that there are no unlinked tests. The **Requirements Linked to Tests** section shows that there are 3 unlinked requirements. Typically, before running the tests, you investigate and address these testing traceability issues by adding tests and linking them to the requirements. For this example, leave the unlinked artifacts and continue to the next step of running the tests.

Test the Model and Analyze Failures and Gaps

After you create and link unit tests that verify the requirements, run the tests to check that the functionality of the model meets the requirements. To see a summary of the test results and coverage measurements, use the widgets in the **Simulation Test Result Analysis** section of the dashboard. The widgets help show testing failures and gaps. Use the metric results to analyze the underlying artifacts and to address the issues.

Perform Unit Testing

Run the test cases for the model by using the Test Manager. Save the test results in your project and review them in the Model Testing Dashboard.

1. Open the unit tests for the model in the Test Manager. In the Model Testing Dashboard, in the **Artifacts** panel, expand the unit **db_DriverSwRequest**. Expand the **Tests > Unit Tests** folder and double-click the test file **db_DriverSwRequest_Tests.mldatx**.

2. In the Test Manager, click **Run**.

3. Save the test results as a file in the project. On the **Tests** tab, in the **Results** section, click **Export**. Name the results file `Results1.mldatx` and save the file under the project root folder.
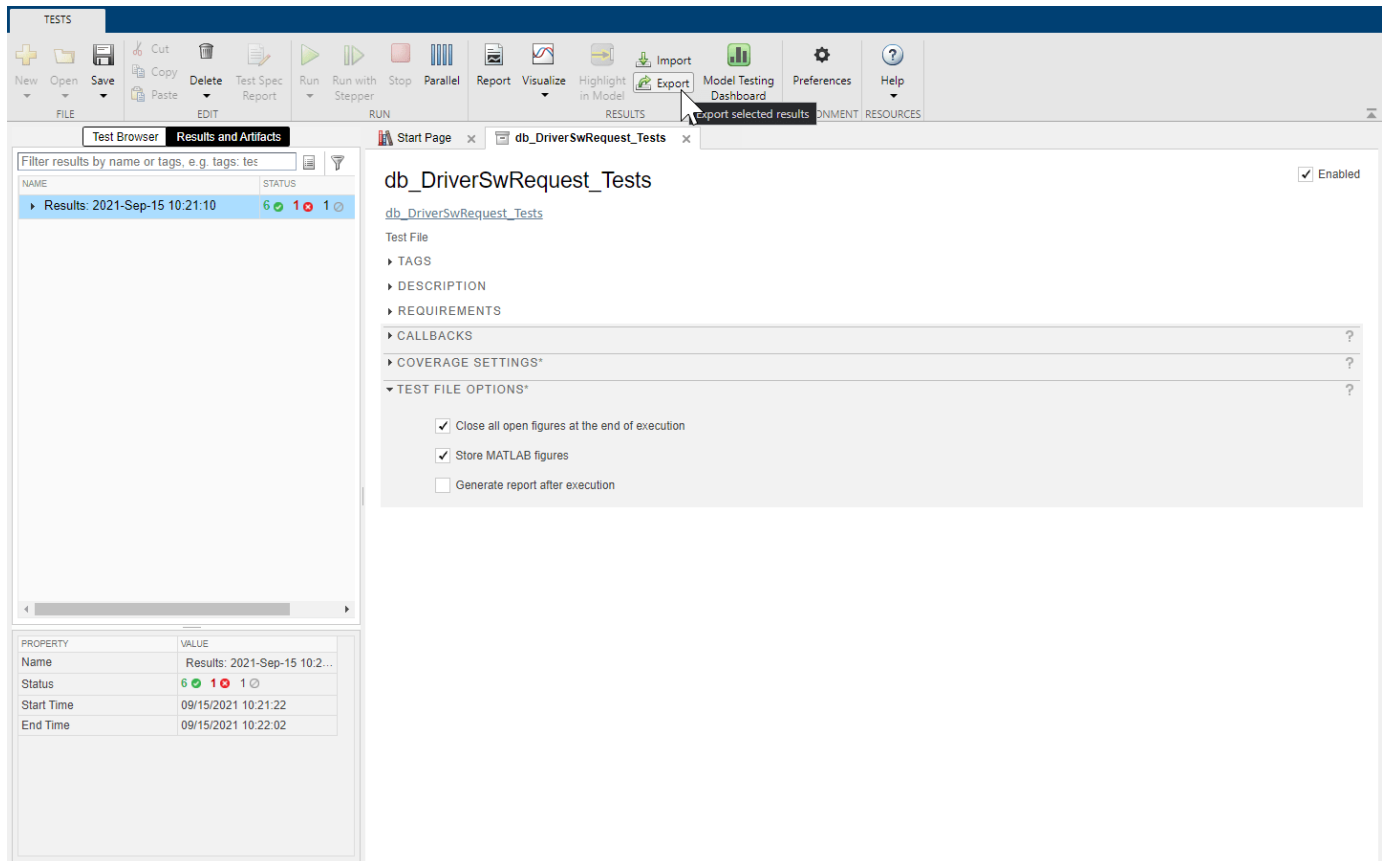
The Model Testing Dashboard detects that you exported the results and automatically updates the **Artifacts** panel to include the new test results for the unit in the subfolder **Test Results > Unit Simulation**.

The dashboard also detects that the metric results are now stale and shows a warning banner at the top of the dashboard.
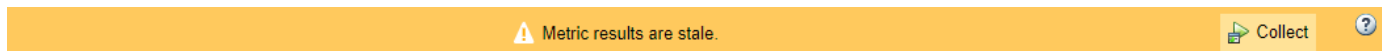
The **Stale** icon appears on the widgets in the **Simulation Test Result Analysis** section to indicate that they are showing stale data that does not include the changes.

Click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current artifacts. If you want to collect metrics for each of the units and components in the project, click **Collect > Collect All**.

Address Testing Failures and Gaps

For the unit db_DriverSwRequest, the **Model Test Status** section of the dashboard indicates that one test failed and one test was disabled during the latest test run.

1   To view the disabled test, in the dashboard, click the **Disabled** widget. The table shows the disabled test cases for the model.

2 Open the disabled test in the Test Manager. In the table, click the test artifact **Detect long decrement**.

3 Enable the test. In the **Test Browser**, right-click the test case and click **Enabled**.

4 Re-run the test. In the **Test Browser**, right-click the test case and click **Run** and save the test suite file.

5 View the updated number of disabled tests. In the dashboard, click the **Collect** button on the warning banner. Note that there are now zero disabled tests reported in the **Model Test Status** section of the dashboard.

6 View the failed test in the dashboard. Click the breadcrumb button for **db_DriverSwRequest** to return to the **Model Testing** results and click the **Failed** widget.

7 Open the failed test in the Test Manager. In the table, click the test artifact **Detect set**.

8 Examine the test failure in the Test Manager. You can determine if you need to update the test or the model by using the test results and links to the model. For this example, instead of fixing the failure, use the breadcrumbs to return to the **Model Testing** results and continue on to examine test coverage.

Check if the tests that you ran fully exercised the model design by using the coverage metrics. For this example, the **Model Coverage** section of the dashboard indicates that some conditions in the model were not covered. Place your cursor over the **Decision** bar in the widget to see what percent of condition coverage was achieved.

1 View details about the decision coverage by clicking one of the **Decision** bars. For this example, click the **Decision** bar for **Achieved** coverage.

2 In the table, expand the model artifact. The table shows the test case results for the model and the results files that contains them. For this example, click on the hyperlink to the source file **Results1.mldatx** to open the results file in the Test Manager.

3 To see detailed coverage results, use the Test Manager to open the model in the Coverage perspective. In the Test Manager, in the **Aggregated Coverage Results** section, in the **Analyzed Model** column, click **db_DriverSwRequest**.

4 Coverage highlighting on the model shows the points that were not covered by the test cases. For this example, do not fix the missing coverage. For a point that is not covered in your project, you can add a test to cover it. You can find the requirement that is implemented by the model element or, if there is none, add a requirement for it. Then you can link the new test case to the requirement. If the point should not be covered, you can justify the missing coverage by using a filter.

Once you have updated the unit tests to address failures and gaps in your project, run the tests and save the results. Then examine the results by collecting the metrics in the dashboard.

Iterative Requirements-Based Testing with the Model Testing Dashboard

In a project with many artifacts and traceability connections, you can monitor the status of the design and testing artifacts whenever there is a change to a file in the project. After you change an artifact, use the dashboard to check if there are downstream testing impacts by updating the tracing data and metric results. Use the **Metric Details** tables to find and fix the affected artifacts. Track your progress by updating the dashboard widgets until they show that the model testing quality meets the standards for the project.

See Also
"Model Testing Metrics"

More About

- "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51
- "Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard" on page 5-96

Manage Requirements-Based Testing Artifacts for Analysis in the Model Testing Dashboard

When you develop and test software units using Model-Based Design, use the Model Testing Dashboard to assess the status and quality of your unit model testing activities. Requirements-based testing is a central element of model verification. By establishing traceability links between your requirements, model design elements, and test cases, you can measure the extent to which the requirements are implemented and verified. The Model Testing Dashboard analyzes this traceability information and provides detailed metric measurements on the traceability, status, and results of these testing artifacts.
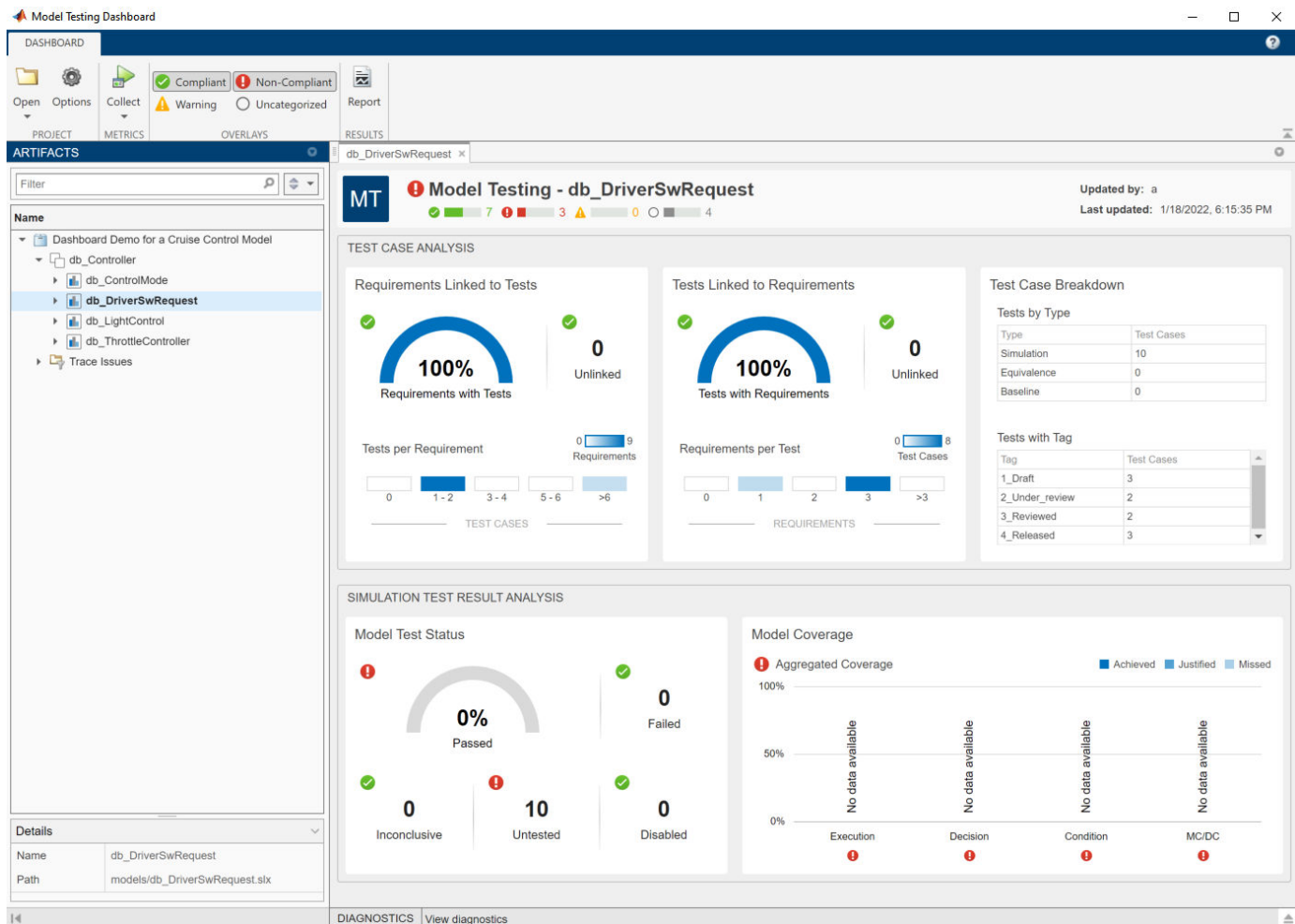
Each metric in the dashboard measures a different aspect of the quality of your unit testing and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. To monitor the requirements-based testing quality of your models in the Model Testing Dashboard, maintain your artifacts in a project and follow these considerations. For more information on using the Model Testing Dashboard, see "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51.

Manage Artifact Files in a Project

To analyze your requirements-based testing activities in the Model Testing Dashboard, store your design and testing artifacts in a MATLAB project. The artifacts that the testing metrics analyze include:

- Models
- Libraries that the models use
- Requirements that you create in Requirements Toolbox
- Test cases that you create in Simulink Test
- Test results from the executed test cases

When your project contains many models and model reference hierarchies, you can track your unit testing activities by configuring the dashboard to recognize the different testing levels of your models. You can specify which entities in your software architecture are units or higher-level components by labeling them in your project and configuring the Model Testing Dashboard to recognize the labels. The dashboard organizes your models in the **Artifacts** panel according to their testing levels and the model reference hierarchy. For more information, see "Categorize Models in a Hierarchy as Components or Units" on page 5-90.

Trace Dependencies Between Project Files and Identify Outdated Metric Results

When you use the Model Testing Dashboard, the dashboard creates a *digital thread* to capture the attributes and unique identifiers of the artifacts in your project. The digital thread is a set of metadata information about the artifacts in a project, the artifact structure, and the traceability relationships between artifacts.

The Model Testing Dashboard monitors and analyzes the digital thread to:

- Detect when project files move and maintain the same universal unique identifiers (UUIDs) for the artifact files and the elements inside the artifact files
- Capture traceability and listen to tools, such as the Test Manager in Simulink Test, to detect new tool outputs and the dependencies of the tool operations
- Identify outdated tool outputs by analyzing the traceability and checksums of inputs to the tool operations
- Create an index of your project and store a representation of each artifact, their inner structure, and their relationships with other artifacts
- Provide a holistic analysis of your project artifacts to help you maintain traceability and up-to-date information on the requirements, units, test cases, and results impacting your design

The dashboard can store the results of the digital thread analysis and then perform traceability analysis across domains, tools, and artifacts, without needing to locally load or access the project artifacts.

As you modify your models and testing artifacts, check that you save the changes to the artifacts files and store the files that you want to analyze in your project.

Trace Artifacts to Units for Model Testing Analysis

To determine which artifacts are in the scope of a unit, the Model Testing Dashboard analyzes the traceability links between the artifacts and the software unit models in the project. The **Artifacts** panel lists the units, organized by the components that reference them. Under each unit, the panel shows these artifacts that trace to the unit:

- Functional Requirements
- Design Artifacts
- Tests
- Test Results
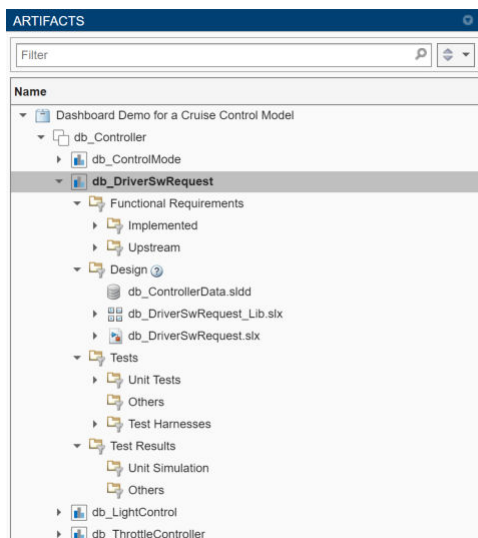
To see the traceability path that the dashboard found from an artifact to its unit, right-click the artifact and click **View trace to unit**. A traceability graph opens in a new tab in the Model Testing Dashboard. The graph shows the connections and intermediate artifacts that the dashboard traced from the unit to the artifact. To see the type of traceability that connects two artifacts, place your cursor over the arrow that connects the artifacts. The traceability relationship is either one artifact containing the other or one artifact tracing to the other. For example, for the unit **db_DriverSwRequest**, expand **Functional Requirements > Upstream > db_SystemReqs.slreqx**. Right-click the requirement for **Target speed increment** and click **View trace to unit**. The trace view shows that the unit `db_DriverSwRequest` traces to the implemented functional requirement `Switch precedence`, which traces to the upstream functional requirement `Target speed increment`.
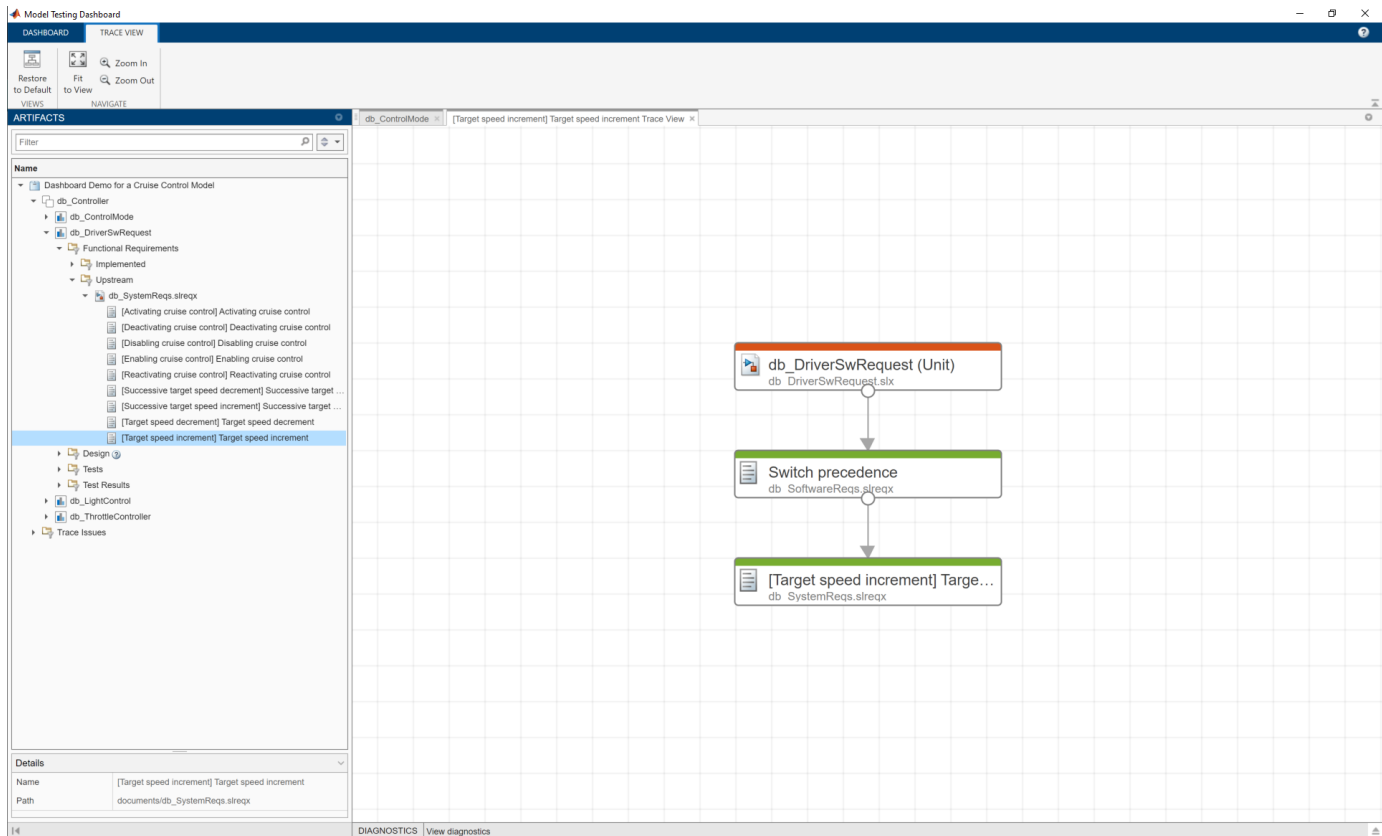
Under the list of components is the folder **Trace Issues** which contains unexpected requirement links, requirements links which are broken or not supported by the dashboard, and artifacts that the dashboard cannot trace to a unit. To help identify the type of tracing issue, the folder **Trace Issues** contains subfolders for **Unexpected Implementation Links**, **Unresolved and Unsupported Links**, **Untraced Tests**, and **Untraced Results**. For more information, see "Fix Requirements-Based Testing Issues" on page 5-61.

If an artifact returns an error during traceability analysis, the panel includes the artifact in an **Errors** folder. Use the traceability information in these sections and in the units to check if the testing artifacts trace to the units that you expect. To see details about the warnings and errors that the dashboard finds during artifact analysis, at the bottom of the Model Testing Dashboard dialog, click **Diagnostics**.

As you edit and save the artifacts in your project, the dashboard tracks your changes and indicates if the traceability data in the **Artifacts** panel might be stale by showing a warning banner. To update the traceability data, click the **Trace Artifacts** button on the warning banner.

Functional Requirements

The folder **Functional Requirements** shows requirements of **Type** `Functional` that are either implemented by or upstream of the unit.

When you collect metric results for a unit, the dashboard analyzes only functional requirements that the unit directly implements. The folder **Functional Requirements** contains two subfolders to help identify which requirements are implemented by the unit or are upstream of the unit:

- **Implemented** — Functional requirements that are directly linked to the unit with a link **Type** of `Implements`. The dashboard uses these requirements in the metrics for the unit.
- **Upstream** — Functional requirements that are indirectly or transitively linked to the implemented requirements. The dashboard does not use these requirements in the metrics for the unit.

If a requirement does not trace to a unit, it appears in the "Trace Issues" folder. If a requirement does not appear in the **Artifacts** panel when you expect it to, see "Requirement Missing from Artifacts Panel" on page 5-97.

Use the Requirements Toolbox to create or import the requirements in a requirements file (`.slreqx`).

Design Artifacts

The folder **Design** shows:

- The model file that contains the block diagram for the unit.
- Models that the unit references.
- Libraries that are partially or fully used by the model.
- Data dictionaries that are linked to the model.

Tests

The folder **Tests** shows test cases and test harness that trace to the unit. This includes test cases that run on the unit and test cases that run on subsystems in the unit model by using test harnesses.

When you collect metric results for a unit, the dashboard analyzes only test cases that run on the unit model or unit model subsystems. The folder **Tests** contains subfolders to help identify which test cases are testing the unit and which test harnesses trace to the unit:

- **Unit Tests** — Test cases that the dashboard considers as unit tests. A unit test directly tests either the entire unit model or the model subsystems. The dashboard uses these tests in the metrics for the unit.
- **Others** — Test cases that trace to the unit but that the dashboard does not consider as unit tests. For example, the dashboard does not consider tests on a library to be unit tests. The dashboard does not use these tests in the metrics for the unit.
- **Test Harnesses** — External test harnesses that trace to the unit or unit subsystems. Double-click a test harness to open it.

If a test case does not trace to a unit, it appears in the "Trace Issues" folder. If a test case does not appear in the **Artifacts** panel when you expect it to, see "Test Case Missing from Artifacts Panel" on page 5-98. For troubleshooting test cases in metric results, see "Fix a test case that does not produce metric results" on page 5-101.

Create test cases in a test suite file by using Simulink Test.

Test Results

When you collect metric results for a unit, the dashboard analyzes only the test results from unit tests. The folder **Test Results** contains two subfolders to help identify which test results are from unit tests:

- **Unit Simulation** — Simulation results from unit tests. The dashboard uses these results in the metrics for the unit.

The following types of test results are shown:

- ⬚ Saved test results — results that you have collected in the Test Manager and have exported to a results file.

- ⬚Temporary test results — results that you have collected in the Test Manager but have not exported to a results file. When you export the results from the Test Manager the dashboard analyzes the saved results instead of the temporary results. Additionally, the dashboard stops recognizing the temporary results when you close the project or close the result set in the Simulink Test Result Explorer. If you want to analyze the results in a subsequent test session or project session, export the results to a results file.
- **Others** — Results that are not simulation results, are not from unit tests, or are only reports. For example, SIL results are not simulation results. The dashboard does not use these results in the metrics for the unit.

If a test result does not trace to a unit, it appears in the "Trace Issues" folder. If a test result does not appear in the **Artifacts** panel when you expect it to, see "Test Result Missing from Artifacts Panel" on page 5-98. For troubleshooting test results in dashboard metric results, see "Fix a test result that does not produce metric results" on page 5-101.

Trace Issues

The folder **Trace Issues** shows artifacts that the dashboard has not traced to units. Use the folder **Trace Issues** to check if artifacts are missing traceability to the units. The folder **Trace Issues** contains subfolders to help identify the type of tracing issue:

- **Unexpected Implementation Links** — Requirement links of **Type** `Implements` for a requirement of **Type** `Container` or **Type** `Informational`. The dashboard does not expect these links to be of **Type** `Implements` because container requirements and informational requirements do not contribute to the Implementation and Verification status of the requirement set that they are in. If a requirement is not meant to be implemented, you can change the link type. For example, you can change a requirement of **Type** `Informational` to have a link of **Type** `Related to`.
- **Unresolved and Unsupported Links** — Requirements links that are either broken in the project or not supported by the dashboard. For example:

  - If a model block implements a requirement, but you delete the model block, the requirement link is now unresolved.
  - If a requirement links to or from a data dictionary, the link is not supported and the Model Testing Dashboard does not trace the link.

    The Model Testing Dashboard does not support traceability analysis for some artifacts and some links. If you expect a link to trace to a unit and it does not, see the troubleshooting solutions in "Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard" on page 5-96.
- **Untraced Tests** — Tests that execute on models or subsystems that are not on the project path.
- **Untraced Results** — Results that the dashboard cannot trace to a test case. For example, if a test case produces a result, but you delete the test case, the dashboard cannot trace the results to the test case.

When you add traceability to an artifact, the dashboard detects this and shows a warning banner at the top of the dashboard to indicate that the artifact traceability shown in the **Artifacts** panel is

outdated. Click the **Trace Artifacts** button on the warning banner to refresh the data in the **Artifacts** panel.

The Model Testing Dashboard does not support traceability analysis for some artifacts and some links. If an artifact is untraced when you expect it to trace to a unit, see the troubleshooting solutions in "Trace Issues" on page 5-99.

Artifact Errors

The folder **Errors** appears if artifacts returned errors when the dashboard performed artifact analysis. These are some errors that artifacts might return during traceability analysis:

- An artifact returns an error if it has unsaved changes when traceability analysis starts.
- A test results file returns an error if it was saved in a previous version of Simulink.
- A model returns an error if it is not on the search path.

Open these artifacts and fix the errors. The dashboard detects changes to the artifacts and shows a warning banner at the top of the dashboard to indicate that the artifact traceability shown in the **Artifacts** panel is outdated. Click the **Trace Artifacts** button on the warning banner to refresh the data in the **Artifacts** panel.

Diagnostics

To see details about artifacts that cause errors, warnings, and informational messages during analysis, at the bottom of the Model Testing Dashboard dialog, click **Diagnostics**. You can filter the diagnostic messages by their type: **Error**, **Warning**, and **Info**. You can also clear the messages from the viewer.

The diagnostic messages show:

- Modeling constructs that the dashboard does not support
- Links that the dashboard does not trace
- Test harnesses or cases that the dashboard does not support
- Test results missing coverage or simulation results
- Artifacts that return errors when the dashboard loads them
- Information about model callbacks that the dashboard deactivates
- Files that have file shadowing or path traceability issues
- Artifacts that are not on the path and are not considered during tracing

Collect Metric Results

The Model Testing Dashboard collects metric results for each unit listed in the **Artifacts** panel. Each metric in the dashboard measures a different aspect of the quality of your model testing and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178. For more information about the available metrics and the results that they return, see "Model Testing Metrics".

As you edit and save the artifacts in your project, the dashboard detects changes to the artifacts and shows a warning banner at the top of the dashboard to indicate that the artifact traceability shown in the **Artifacts** panel is outdated. Click the **Trace Artifacts** button on the warning banner to refresh the data in the **Artifacts** panel.

After you update the traceability information, if the metric results might be affected by your artifact changes, the dashboard shows a warning banner at the top of the dashboard to indicate that the

metric results are stale. Affected widgets have a gray staleness icon **STALE**. To update the results, click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current artifacts. If you want to collect metrics for each of the units and components in the project, click **Collect > Collect All**.

The dashboard does not indicate stale metric data for these changes:

- After you run a test case and analyze the results in the dashboard, if you make changes to the test case, the dashboard indicates that test case metrics are stale but does not indicate that the results metrics are stale.
- When you change a coverage filter file that your test results use, the coverage metrics in the dashboard do not indicate stale data or include the changes. After you save the changes to the filter file, re-run the tests and use the filter file for the new results.

See Also
"Model Testing Metrics"

Related Examples
- "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51
- "Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard" on page 5-96

Assess the Completeness of Requirements-Based Testing in Accordance with ISO 26262

You can use the Model Testing Dashboard to assess the quality and completeness of your requirements-based testing activities in accordance with ISO 26262-6:2018. The dashboard facilitates this activity by monitoring the traceability between requirements, tests, and test results and by providing a summary of testing completeness and structural coverage. The dashboard analyzes the implementation and verification artifacts in a project and provides:

- Completeness and quality metrics for the requirements-based test cases in accordance with ISO 26262-6:2018, Clause 9.4.3
- Completeness and quality metrics for the requirements-based test results in accordance with ISO 26262-6:2018, Clause 9.4.4
- A list of artifacts in the project, organized by the units

To assess the completeness of your requirements-based testing activities, follow these automated and manual review steps using the Model Testing Dashboard.

Open the Model Testing Dashboard and Collect Metric Results

To analyze testing artifacts using the Model Testing Dashboard:

1 Open a project that contains your models and testing artifacts. Or to load an example project for the dashboard, in the MATLAB Command Window, enter `dashboardCCProjectStart('incomplete')`.

2 Open the dashboard. To open the Model Testing Dashboard, use one of these approaches:

- On the **Project** tab, click **Model Testing Dashboard**.
- At the MATLAB command line, enter `modelTestingDashboard`.

If you have not previously opened the dashboard for the project, the dashboard performs a first-time setup, identifying the artifacts in the project and collecting traceability information.

3 In the **Artifacts** panel, the dashboard organizes artifacts such as requirements, test cases, and test results under the models that they trace to. To view the metric results for the unit `db_DriverSwRequest` in the example project, in the **Artifacts** panel, click `db_DriverSwRequest`. The dashboard collects metric results and populates the widgets with the metric data for the unit. The dashboard automatically collects metric results when you click a unit. To turn the automatic metric collection off, click **Collect > Auto Collect**.

Note If you do not specify the models that are considered units, then the Model Testing Dashboard considers a model to be a unit if it does not reference other models. You can control which models appear as units and components by labeling them in your project and configuring the Model Testing Dashboard to recognize the labels. For more information, see "Specify Models as Components and Units" on page 5-91.
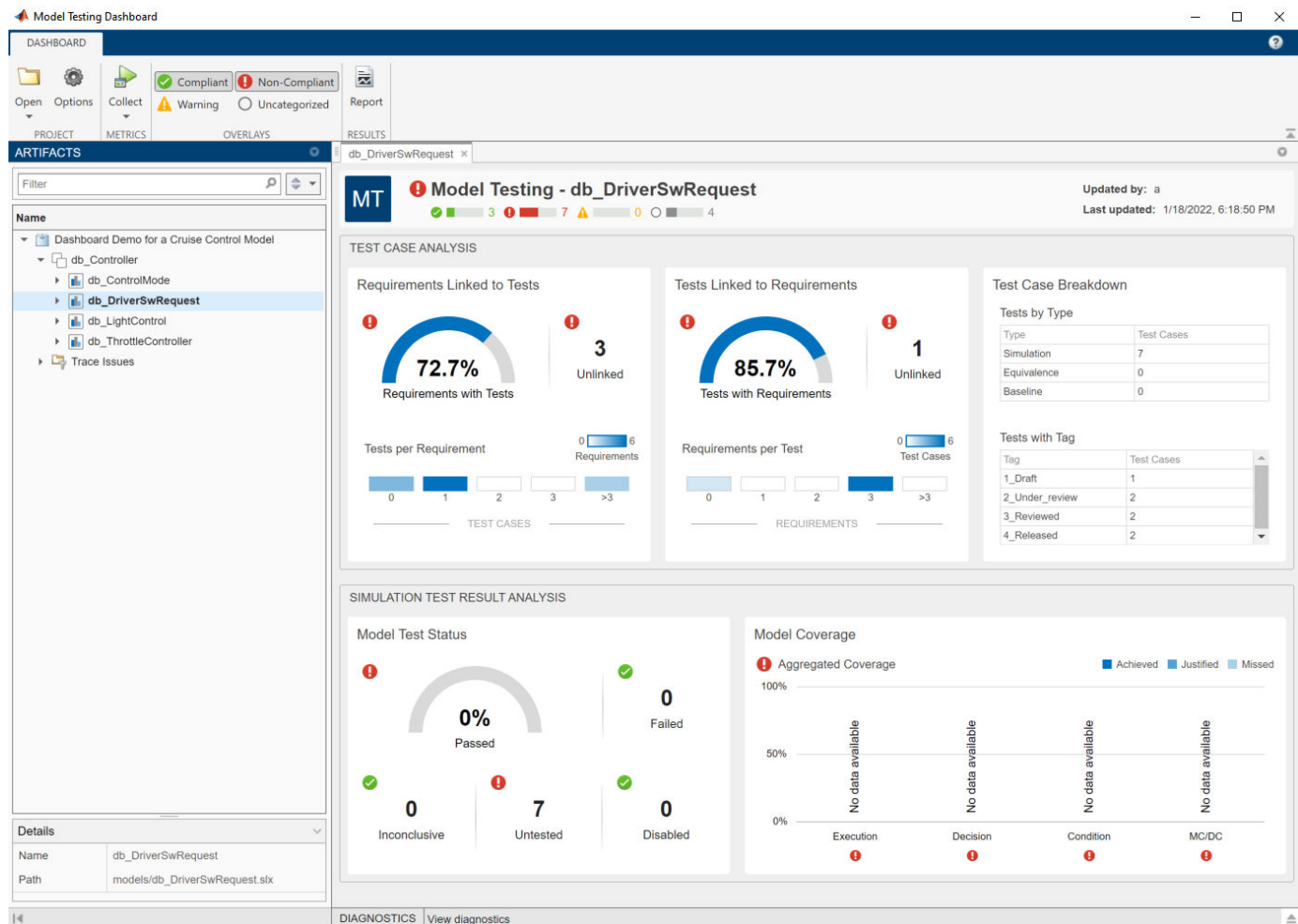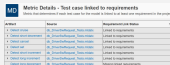
The dashboard widgets summarize the traceability and completeness measurements for the testing artifacts for each unit. The metric results displayed with the red **Non-Compliant** overlay icon ⊘ indicate issues that you may need to address to complete requirements-based testing for the unit. Results are compliant if they show full traceability, test completion, or model coverage. To see the compliance thresholds for a metric, point to the overlay icon. To explore the data in more detail, click an individual metric widget. For the selected metric, a table displays the artifacts and the metric value for each artifact. The table provides hyperlinks to open the artifacts so that you can get detailed metric results and fix the artifacts that have issues. For more information about using the Model Testing Dashboard, see "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51.

Test Case Review

To verify that a unit satisfies its requirements, you create test cases for the unit based on the requirements. ISO 26262-6, Clause 9.4.3 requires that test cases for a unit are derived from the requirements. When you create a test case for a requirement, you add a traceability link between the test case and the requirement, as described in "Link Requirements to Tests" (Requirements Toolbox) and in "Establish Requirements Traceability for Testing" (Simulink Test). Traceability allows you to track which requirements have been verified by your tests and identify requirements that the model does not satisfy. Clause 9.4.3 requires traceability between requirements and test cases, and review

of the correctness and completeness of the test cases. To assess the correctness and completeness of the test cases for a unit, use the metrics in the **Test Case Analysis** section of the Model Testing Dashboard.

The following is an example checklist provided to aid in reviewing test case correctness and completeness with respect to ISO 26262-6. For each question, perform the review activity using the corresponding dashboard metric and apply the corresponding fix. This checklist is provided as an example and should be reviewed and modified to meet your application needs.

Checklist Item	Review Activity	Dashboard Metric	Fix
1 — Does each test case trace to a requirement?	Check that 100% of the test cases for the unit are linked to requirements by viewing the widgets in the **Tests Linked to Requirements** section.	**Tests Linked to Requirements** Metric ID — `TestCaseWithRequirementPercentage` For more information, see Test linked to requirement percentage.	For each unlinked test case, add a link to the requirement that the test case verifies, as described in "Fix Requirements-Based Testing Issues" on page 5-61.
2 — Does each test case trace to the correct requirements?	For each test case, manually verify that the requirement it is linked to is correct. Click the **Tests with Requirements** widget to view a table of the test cases. To see the requirements that a test case traces to, in the **Artifacts** column, click the arrow to the left of the test case name.	**Tests Linked to Requirements** Metric ID — `TestCaseWithRequirement` For more information, see Test linked to requirements.	For each link to an incorrect requirement, remove the link. If the test case is missing a link to the correct requirement, add the correct link.
3 — Do the test cases cover all requirements?	Check that 100% of the requirements for the unit are linked to test cases by viewing the widgets in the **Requirements Linked to Tests** section.	**Requirements Linked to Tests** Metric ID — `RequirementWithTestCasePercentage` For more information, see Percentage requirements with test cases.	For each unlinked requirement, add a link to the test case that verifies it, as described in "Fix Requirements-Based Testing Issues" on page 5-61.
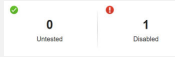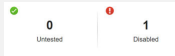
Checklist Item	Review Activity	Dashboard Metric	Fix
4 — Do the test cases define the expected results including pass/fail criteria?	Manually review the test cases of each type. Click the widgets in the **Tests by Type** section to view a table of the test cases for each type: **Simulation**, **Equivalence**, and **Baseline**. Open each test case in the Test Manager by using the hyperlinks in the **Artifact** column. Baseline test cases must define baseline criteria. For simulation test cases, review that each test case defines pass/fail criteria by using assessments, as described in "Assess Simulation and Compare Output Data" (Simulink Test).	**Tests by Type** Metric ID — `TestCaseType` For more information, see Test case type.	For each test case that does not define expected results, click the hyperlink in the **Artifact** column to open the test case in the Test Manager, then add the expected test definition and pass/fail criteria.
5 — Does each test case properly test the requirement that it traces to?	Manually review the requirement links and content for each test case. Click the **Tests with Requirements** widget to view a table of the test cases. To see the requirements that a test case traces to, in the **Artifact** column, click the arrow to the left of the test case name. Use the hyperlinks to open the test case and requirement and review that the test case properly tests the requirement.	**Tests Linked to Requirements** Metric ID — `TestCaseWithRequirement` For more information, see Test linked to requirements.	For each test case that does not properly test the requirement it traces to, click the hyperlink in the **Artifact** column to open the test case in the Test Manager, then update the test case. Alternatively, add test cases that further test the requirement.
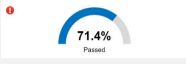
Test Results Review

After you run tests on a unit, you must review the results to check that the tests executed, passed, and sufficiently tested the unit. Clause 9.4.4 in ISO 26262-6:2018 requires that you analyze the coverage of requirements for each unit. Check that each of the test cases tested the intended model and passed. Additionally, measure the coverage of the unit by collecting model coverage results in the

tests. To assess the testing coverage of the requirements for the unit, use the metrics in the
Simulation Test Result Analysis section of the Model Testing Dashboard.

The following checklist is provided to facilitate test results analysis and review using the dashboard.
For each question, perform the review activity using the corresponding dashboard metric and apply
the corresponding fix. This checklist is provided as an example and should be reviewed and modified
to meet your application needs.

Checklist Item	Review Activity	Dashboard Metric	Fix
1 — Does each test result trace to a test case?	Use only test results that appear in the dashboard. Test results that do not trace to a test case do not appear in the dashboard. Click a widget in the **Model Test Status** section to view a table of the test cases and the results that trace to them.	**Model Test Status** Metric ID — `TestCaseStatusDistribution` For more information, see Test case status distribution.	Open the metric details and click the hyperlink in the **Artifacts** column to open the test case in the Test Manager. Re-run the tests that the results should trace to and export the new results.
2 — Does each test case trace to a test result?	Check that zero test cases are untested and zero test cases are disabled.	**Model Test Status** Metric ID — `TestCaseStatusDistribution` For more information, see Test case status distribution.	For each disabled or untested test case, in the Test Manager, enable and run the test.
3 — Have all test cases been executed?	Check that zero test cases are untested and zero test cases are disabled.	**Model Test Status** Metric ID — `TestCaseStatusDistribution` For more information, see Test case status distribution.	For each disabled or untested test case, in the Test Manager, enable, and run the test.

Checklist Item	Review Activity	Dashboard Metric	Fix
4 — Do all test cases pass?	Check that 100% of the test cases for the unit passed.	**Model Test Status > Passed** Metric ID — `TestCaseStatusPercentage` For more information, see Test case status percentage.	For each test failure, review the failure in the Test Manager and fix the corresponding test case or design element in the model.
5 — Do all test results include coverage results?	Manually review each test result in the Test Manager to check that it includes coverage results.	Not applicable	For each test result that does not include coverage, open the test case in the Test Manager, and then enable coverage collection. Run the test case again.

Checklist Item	Review Activity	Dashboard Metric	Fix
6 — Were the required structural coverage objectives achieved for each unit?	Check that the tests achieved 100% model coverage for the coverage types that your unit testing requires. To determine the required coverage types, consider the safety level of your software unit and use table 9 in clause 9.4.4 of ISO 26262-6:2018.	**Model Coverage** Metric ID — `ExecutionCoverageBreakdown` Metric ID — `ConditionCoverageBreakdown` Metric ID — `DecisionCoverageBreakdown` Metric ID — `MCDCCoverageBreakdown` For more information, see: • Justified and achieved execution coverage • Justified and achieved condition coverage • Justified and achieved decision coverage • Justified and achieved MC/DC coverage	For each design element that is not covered, analyze to determine the cause of the missed coverage. Analysis can reveal shortcomings in tests, requirements, or implementation. If appropriate, add tests to cover the element. Alternatively, add a justification filter that justifies the missed coverage, as described in "Create, Edit, and View Coverage Filter Rules" (Simulink Coverage).

Checklist Item	Review Activity	Dashboard Metric	Fix
7 — Have shortcomings been acceptably justified?	Manually review coverage justifications. Click a bar in the **Model Coverage** section to view a table of the results for the corresponding coverage type. To open a test result in the Test Manager for further review, click the hyperlink in the **Artifacts** column.	**Model Coverage** Metric ID — `ExecutionCoverageBreakdown` Metric ID — `ConditionCoverageBreakdown` Metric ID — `DecisionCoverageBreakdown` Metric ID — `MCDCCoverageBreakdown` For more information, see: • Justified and achieved execution coverage • Justified and achieved condition coverage • Justified and achieved decision coverage • Justified and achieved MC/DC coverage	For each coverage gap that is not acceptably justified, update the justification of missing coverage. Alternatively, add test cases to cover the gap.

Unit Verification in Accordance with ISO 26262

The Model Testing Dashboard provides information about the quality and completeness of your unit requirements-based testing activities. To comply with ISO 26262-6:2018, you must also test your software at other architectural levels. ISO 26262-6:2018 describes compliance requirements for these testing levels:

• Software unit testing in Table 7, method 1j
• Software integration testing in Table 10, method 1a
• Embedded software testing in Table 14, method 1a

The generic verification process detailed in ISO 26262-8:2018, clause 9 includes additional information on how you can systematically achieve testing for each of these levels by using planning, specification, execution, evaluation, and documentation of tests. This table shows how the Model

Testing Dashboard applies to the requirements in ISO 26262-8:2018, clause 9 for the unit testing level, and complementary activities required to perform to show compliance.

Requirement	Compliance Argument	Complementary Activities
9.4.1 — Scope of verification activity	The Model Testing Dashboard applies to all safety-related and non-safety-related software units.	Not applicable
9.4.2 — Verification methods	The Model Testing Dashboard provides a summary on the completion of requirements-based testing (Table 7, method 1j) including a view on test results.	Where applicable, apply one or more of these other verification methods: • Manual review and analysis check list • Applying other tools, such as static code analysis, control flow analysis, and data flow analysis • Developing extra tests, such as interface tests, fault injection tests, and back-to-back comparisons
9.4.3 — Methods for deriving test cases	The Model Testing Dashboard provides several ways to traverse the software unit requirements and the relevant tests, which helps you to derive test cases from the requirements.	You can also derive test cases by using other tools, such as Simulink Design Verifier.
9.4.4 — Requirement and structural coverage	The Model Testing Dashboard aids in showing: • Completeness of requirement coverage • Branch/statement and MCDC model coverage achieved by testing • A rationale for the sufficiency of achieved coverage	The dashboard provides structural coverage only at the model level. You can use other tools to track the structural coverage at the code level.
9.4.5 — Test environment	The Model Testing Dashboard aids in requirements-based testing at the model level.	Apply back-to-back comparison tests to verify that the behavior of the model is equivalent to the generated code.

References:

- ISO 26262-4:2018(en)Road vehicles — Functional safety — Part 4: Product development at the system level, International Standardization Organization

ISO 26262-6:2018(en)Road vehicles — Functional safety — Part 6: Product development at the software level, International Standardization Organization

ISO 26262-8:2018(en)Road vehicles — Functional safety — Part 8: Supporting processes, International Standardization Organization

See Also
"Model Testing Metrics"

Related Examples

- "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51
- "Fix Requirements-Based Testing Issues" on page 5-61

Collect Metrics on Model Testing Artifacts Programmatically

This example shows how to programmatically assess the status and quality of requirements-based testing activities in a project. When you develop software units by using Model-Based Design, you use requirements-based testing to verify your models. You can assess the testing status of one unit model by using the metric API to collect metric data on the traceability between requirements and test cases and on the status of test results. The metrics measure characteristics of completeness and quality of requirements-based testing that reflect industry standards such as ISO 26262 and DO-178. After collecting metric results, you can access the results or export them to a file. By running a script that collects these metrics, you can automatically analyze the testing status of your project to, for example, design a continuous integration system. Use the results to monitor testing completeness or to detect downstream testing impacts when you make changes to artifacts in the project.

Open the Project

Open a project that contains models and testing artifacts. For this example, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart('incomplete')
```

The project contains models and requirements and test cases for the models. Some of the requirements have traceability links to the models and test cases, which help to verify that a model's functionality meets the requirements.

Collect Metric Results

Create a `metric.Engine` object for the current project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect pending artifact changes and to track the test results.

```
updateArtifacts(metric_engine);
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of the metric identifiers used in the requirements-based version of the Model Testing Dashboard app. For more information, see `getAvailableMetricIds`.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
'App','DashboardApp',...
'Dashboard','RequirementsBasedModelUnitTesting');
```

For a list of model testing metrics and their identifiers, see "Model Testing Metrics".

When you collect metric results, you can collect results for one unit at a time or for each unit in the project.

Collect Results for One Unit

When you collect and view results for a unit, the metrics return data for the artifacts that trace to the model.

Collect the metric results for the `db_DriverSwRequest`.

Create an array that identifies the path to the model file in the project and the name of the model.

```
unit = {fullfile(pwd,'models','db_DriverSwRequest.slx'),'db_DriverSwRequest'};
```

Execute the engine and use `'ArtifactScope'` to specify the unit for which you want to collect results. The engine runs the metrics on only the artifacts that trace to the model that you specify. Collecting results for these metrics requires a Simulink Test license, a Requirements Toolbox license, and a Simulink Coverage license.

```
execute(metric_engine, metric_Ids, 'ArtifactScope', unit)
```

Collect Results for Each Unit in the Project

To collect the results for each unit in the project, execute the engine without the argument for `'ArtifactScope'`.

```
execute(metric_engine, metric_Ids)
```

For more information on collecting metric results, see the function `execute`.

Access Results

Generate a report file that contains the results for all units in the project. For this example, specify the HTML file format, use `pwd` to provide the path to the current folder, and name the report `'MetricResultsReport.html'`.

```
reportLocation = fullfile(pwd, 'MetricResultsReport.html');
generateReport(metric_engine,'Type','html-file','Location',reportLocation);
```

Open the HTML report. The report is in the current folder, at the root of the project.

```
web('MetricResultsReport.html')
```

To open the table of contents and navigate to results for each unit, click the menu icon in the top-left corner of the report. For each unit in the report, there is an artifact summary table that displays the size and structure of that unit.

1.1. Artifact Summary

Artifact Group	Artifact Type	Number of Artifacts
Requirements	Functional requirement	41
Design	Block diagram	3
	Model reference	0
	Subsystem	17
	Stateflow chart	0
	MATLAB function	0
	Data dictionary file	1
Tests	Test case	37
	Test case result	0

Saving the metric results in a report file allows you to access the results without opening the project and the dashboard. Alternatively, you can open the Model Testing Dashboard to see the results and explore the artifacts.

```
modelTestingDashboard
```

To access the results programmatically, use the `getMetrics` function. The function returns the `metric.Result` objects that contain the result data for the specified unit and metrics. For this

example, store the results for the metrics `TestCaseStatus` and `TestCasesPerRequirementDistribution` in corresponding arrays.

```
results_TestCasesPerReqDist = getMetrics(metric_engine, 'TestCasesPerRequirementDistribution');
results_TestStatus = getMetrics(metric_engine, 'TestCaseStatus');
```

View Distribution of Test Case Links per Requirement

The metric `TestCasesPerRequirementDistribution` returns a distribution of the number of test cases linked to each functional requirement for the unit. Use the `disp` function to display the bin edges and bin counts of the distribution, which are fields in the `Value` field of the `metric.Result` object. The left edge of each bin shows the number of test case links and the bin count shows the number of requirements that are linked to that number of test cases. The sixth bin edge is 18446744073709551615, which is the upper limit of the count of test cases per requirement, which shows that the fifth bin contains requirements that have four or more test cases.

```
disp(['Unit:  ', results_TestCasesPerReqDist(1).Scope(1).Name])
disp(['  Tests per Requirement:  ', num2str(results_TestCasesPerReqDist(1).Value.BinEdges)])
disp(['  Requirements:  ', num2str(results_TestCasesPerReqDist(1).Value.BinCounts)])

Unit:  db_DriverSwRequest
  Tests per Requirement:  0  1  2  3  4  18446744073709551615
  Requirements:  3  6  0  0  2
```

This result shows that for the unit `db_DriverSwRequest` there are 3 requirements that are not linked to test cases, 6 requirements that are linked to one test case, and 2 requirements that are linked to four or more test cases. Each requirement should be linked to at least one test case that verifies that the model meets the requirement. The distribution also allows you to check if a requirement has many more test cases than the other requirements, which might indicate that the requirement is too general and that you should break it into more granular requirements.

View Test Case Status Results

The metric `TestCaseStatus` assesses the testing status of each test case for the unit and returns one of these numeric results:

- 0 — Failed
- 1 — Passed
- 2 — Disabled
- 3 — Untested

Display the name and status of each test case.

```
for n=1:length(results_TestStatus)

    disp(['Test Case: ', results_TestStatus(n).Artifacts(1).Name])
    disp([' Status: ', num2str(results_TestStatus(n).Value)])

end
```

For this example, the tests have not been run, so each test case returns a status of 3.

See Also

"Model Testing Metrics" | `metric.Engine` | `execute` | `generateReport` | `getAvailableMetricIds` | `updateArtifacts`

Related Examples

- "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51
- "Test Model Against Requirements and Report Results" (Requirements Toolbox)
- "Perform Functional Testing and Analyze Test Coverage" (Simulink Test)

Categorize Models in a Hierarchy as Components or Units

When testing your model-based software architecture, there are different testing requirements for different levels of the architecture. The Model Testing Dashboard helps you to focus on the models that require unit testing so you can assess their testing quality. You can use labels to classify the models in your projects as units or components, then use the Model Testing Dashboard to see the hierarchy and analyze the testing requirements for the units. For more information, see "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51.

Units in the Model Testing Dashboard

A *unit* is a functional entity in your software architecture that you can execute and test independently or as part of larger system tests. Software development standards, such as ISO 26262-6, define objectives for unit testing. Unit tests typically must cover each of the requirements for the unit and must demonstrate traceability between the requirements, the test cases, and the unit. Unit tests must also meet certain coverage objectives for the unit, such as modified condition/decision coverage.

You can label models as units in the Model Testing Dashboard. The dashboard then provides metric results for each unit. If you do not specify the models that are considered units, then the dashboard considers a model to be a unit if it does not reference other models.

In the Model Testing Dashboard, in the **Artifacts** panel, the unit dashboard icon ![icon] indicates a unit. If a unit is referenced by a component, it appears under the component. If a unit references one or more other models, those models are part of the unit. The referenced models appear in the **Design** folder under the unit and contribute to the metric results for the unit.
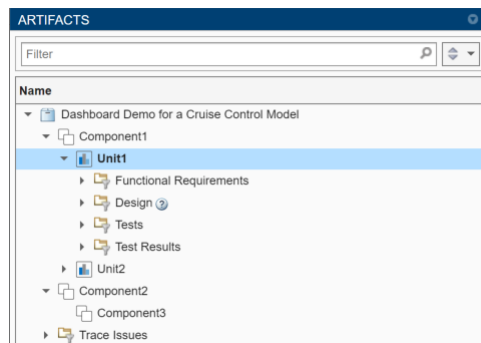
To specify which models are units, label them in your project and configure the dashboard to recognize the label, as shown in "Specify Models as Components and Units" on page 5-91.

Components in the Model Testing Dashboard

A *component* is an entity that integrates multiple testable units together. For example:

- A model that references multiple unit models could be a component model.
- A System Composer™ architecture model could be a component. Supported architectures include System Composer architecture models, System Composer software architecture models, and AUTOSAR architectures.
- A component could also integrate other components.

The Model Testing Dashboard organizes components and units under the components that reference them in the **Artifacts** panel. The dashboard does not provide metric results for components because components typically must meet different testing objectives than units.

If you do not specify the models that are considered components, then the dashboard considers a model to be a component if it references one or more other models.

In the Model Testing Dashboard, in the **Artifacts** panel, the component icon ⬜ indicates a component. To see the units under a component, expand the component node by clicking the arrow next to the component icon.

To specify the models that are considered components, label them in your project and configure the dashboard to recognize the label, as shown in "Specify Models as Components and Units" on page 5-91.

Specify Models as Components and Units

You can control which models appear as units and components by labeling them in your project and configuring the Model Testing Dashboard to recognize the labels.

1   Open your project. For example, at the command line, type `dashboardCCProjectStart`. This example project already has component and unit models configured.

2   In MATLAB, at the bottom left of the **Project** window, right-click in the **Labels** pane and click **Create New Category**. Type a name for the category that will contain your testing architecture labels, for example, `Testing Interface` and then click **Create**.

3   Create a label for the units. On the **Labels** pane, right-click the category that you created and click **Create New Label**. Type the label name `Software Unit` and click **OK**.

4   Create another label for component models and name the label `Software Component`.
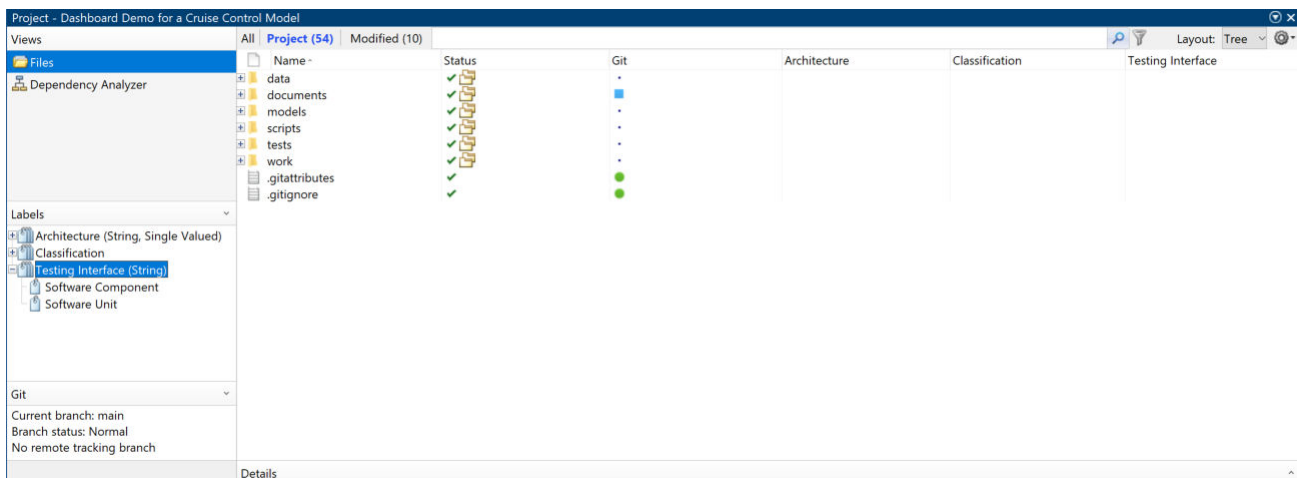
The unit and component labels appear under the category in the **Labels** pane.

5   Label the models in the project as components and units. In the project pane, right-click a model and click **Add label**. In the dialog box, select the label and click **OK**. For this example, apply these labels:

- `db_Controller` — Software Component
- `db_ControlMode` — Software Unit
- `db_DriverSwRequest` — Software Unit
- `db_LightControl` — Software Unit
- `db_TargetSpeedThrottle` — Software Unit

6 To open the Model Testing Dashboard, use one of these approaches:

- On the **Project** tab, in the **Tools** section, click **Model Testing Dashboard**.
- At the MATLAB command line, enter `modelTestingDashboard`.

7 In the **Dashboard** tab, click **Options**.

8 In the Project Options dialog box, in the **Classification** section, specify the category and labels that you created for the components and units. For the component interface, set **Category** to `Testing Interface` and **Label** to `Software Component`. For the unit interface, set **Category** to `Testing Interface` and **Label** to `Software Unit`.
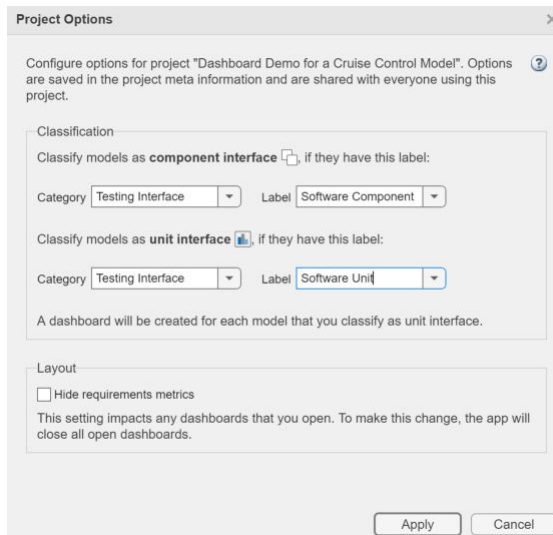
9 Click **Apply**. The dashboard updates the traceability information in the **Artifacts** panel and organizes the models under the component models that reference them. If a model is not referenced by a component, it appears at the top level with the components.

For each unit, the dashboard shows the artifacts that trace to the unit. To view the metric results for a unit, click the unit name in the **Artifacts** panel. The dashboard shows the model testing metric results for the unit that you select.

See Also

Related Examples
- "Create Labels"
- "Add Labels to Files"

Include Subsystem-Level Test Results in the Model Testing Dashboard

You can run subsystem-level tests and analyze the aggregated model coverage in the Model Testing Dashboard. The dashboard metrics include test cases that you created on:

- Atomic subsystems
- Atomic subsystem references
- Atomic Stateflow® charts
- Atomic MATLAB® Function blocks
- Referenced models

This example shows how you can address gaps in your achieved coverage by using subsystem-level tests.

Open the Dashboard for the Project

1. Open a project that contains models and testing artifacts. For this example, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart
```

2. Open the Model Testing Dashboard by using one of these approaches:

- On the **Project** tab, click **Model Testing Dashboard**.
- At the command line, enter:

```
modelTestingDashboard
```

The dashboard opens for the unit db_ControlMode.

Run Tests for the Unit and Identify Gaps in Model Coverage

Run tests on the whole unit and use the Model Testing Dashboard to collect the metric results for model coverage.

1. Open the test cases associated with the unit db_ControlMode. In the Model Testing Dashboard, in the **Artifacts** panel, expand the unit **db_ControlMode**. Expand the **Tests > Unit Tests** folder and double-click the test file **db_ControlMode_Tests.mldatx**.

2. Expand the test file **db_ControlMode_Tests.mldatx**. The test file contains three test suites:

- Control Mode Unit Tests — Tests for the subsystem Control_Mode_StateMachine
- Target Speed Unit Tests — Tests for the subsystem Target_Speed_Calculator
- Combined Tests — Test for the unit db_ControlMode

3. Run the test suite Combined Tests. Right-click **Combined Tests** and click **Run**. The test suite contains the test case Idle. Idle is a top-level test for the unit db_ControlMode. A *top-level test* is a model test that tests the whole unit.

4. View the model coverage. In the dashboard, click the **Collect** button on the warning banner or toolstrip to refresh the model coverage widgets.

5. View the achieved execution coverage for the unit. In the **Model Coverage** section, point to the achieved coverage in the **Execution** bar.

Based on the results from the top-level test, the achieved execution coverage is 38.2%.
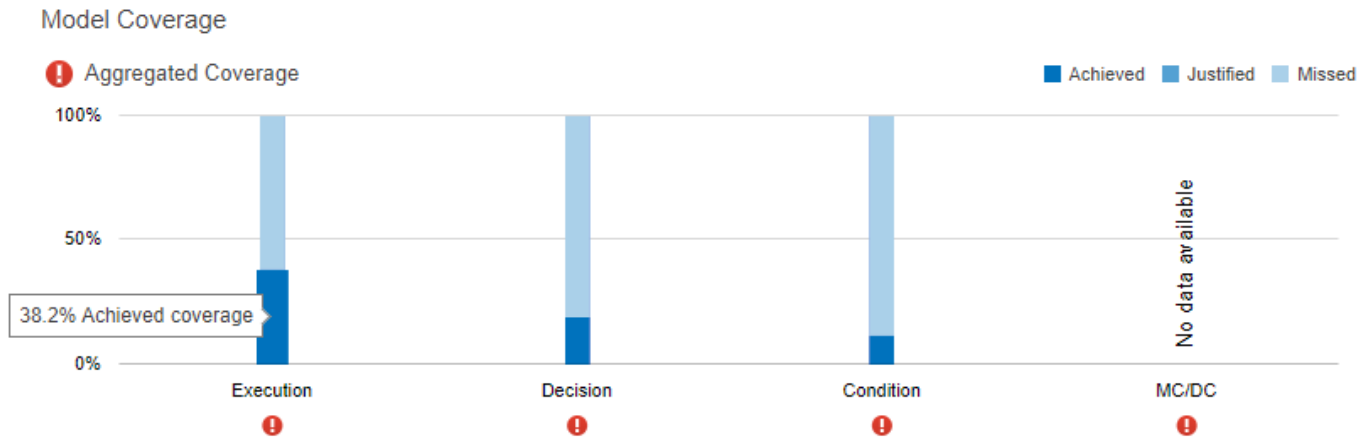
Use Subsystem-Level Testing to Address Gaps in Coverage

Run tests at the subsystem-level and use the Model Testing Dashboard to observe the increase in model coverage.

1. In the Test Manager, click **Test Browser**.

2. Run the test suite `Target Speed Unit Tests`. Right-click **Target Speed Unit Tests** and click **Run**.

3. In the Model Testing Dashboard, click the **Collect** button on the warning banner or toolstrip to refresh the model coverage widgets. In the **Model Coverage** section, the aggregated model coverage includes the results from the subsystem-level tests in `Target Speed Unit Tests`.

Note that you can only collect aggregated coverage for subsystem-level tests if you have previously run a top-level test executed on the model. The dashboard cannot calculate aggregated coverage for units that only have test results at the subsystem level because subsystem-level tests do not test the whole unit. The top-level test must be executed on the model, not by SIL or PIL simulations on the model code.

For this example, you could not calculate aggregated coverage without the test results from the test suite `Combined Tests`. The test suites `Control Mode Unit Tests` and `Target Speed Unit Tests` only contain subsystem-level tests and therefore do not test the whole unit.

4. View the achieved execution coverage for the unit. In the **Model Coverage** section, point to the achieved coverage in the **Execution** bar.

Based on the results from the top-level and subsystem-level tests, the achieved execution coverage is now 47.1%. The results from the subsystem-level tests in `Target Speed Unit Tests` increased the achieved model coverage.
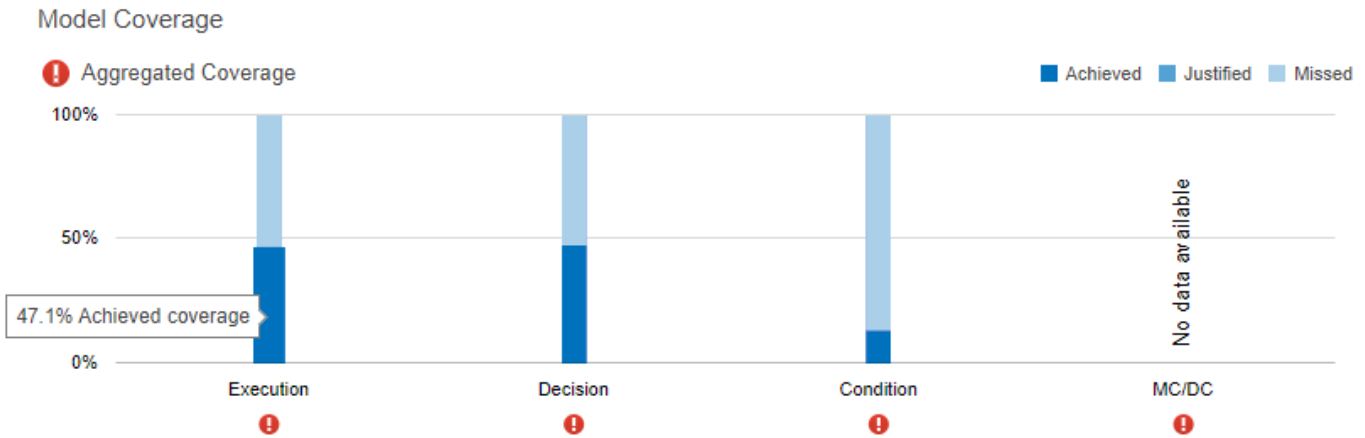
The subsystem-level tests in test suites `Control Mode Unit Tests` and `Target Speed Unit Tests` allow you to address the gaps in model coverage from the top-level tests.

If you run the test suite `Control Mode Unit Tests`, the unit `db_ControlMode` is able to achieve 100% execution, decision, and condition coverage.
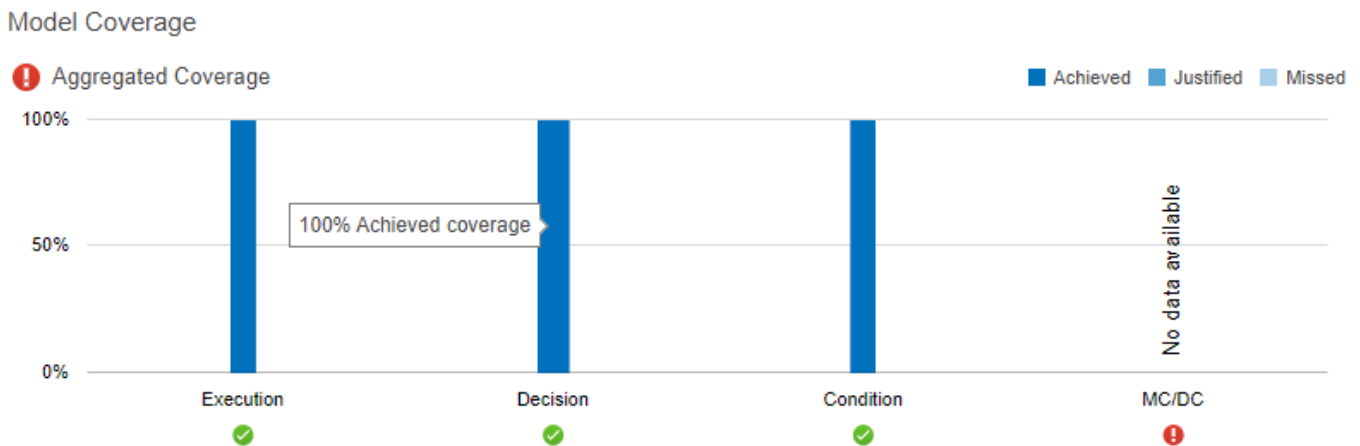
See Also

Related Examples

- "Assess the Completeness of Requirements-Based Testing in Accordance with ISO 26262" on page 5-76
- "Fix Requirements-Based Testing Issues" on page 5-61
- "Model Testing Metrics"

Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard

Issue

The Model Testing Dashboard analyzes artifacts—models, requirements, tests, and results—that are part of the requirements-based testing workflow for software unit models. If an artifact or a link between artifacts is not part of the requirements-based testing workflow, it might not appear in the Model Testing Dashboard or contribute to the analysis results. Additionally, some artifacts and links are not supported by the Model Testing Dashboard. If you expect a link or artifact to appear in the dashboard and it does not, try one of these solutions.

Possible Solutions

Try these solutions when you begin troubleshooting artifacts in the Model Testing Dashboard:

- Save changes to your artifact files.
- Check that your artifacts are saved in the project. The Model Testing Dashboard does not analyze files that are not saved in the project.
- Check that your artifacts are not in a referenced project. The Model Testing Dashboard does not analyze files in referenced projects.
- Check that your artifacts are on the MATLAB search path before you open the dashboard. When you change the MATLAB search path, the traceability information in the **Artifacts** panel is not updated. Do not change the search path while the dashboard is open.
- Open the "Diagnostics" on page 5-74 pane and address errors or warnings.
- Use the dashboard to re-trace the artifacts and re-collect metric results.

Depending on the type of artifact or analysis issue that you are troubleshooting, try one of these solutions.

Enable Artifact Tracing for the Project

As you edit and save the artifacts in your project, the dashboard needs to track these changes to enable artifact tracing and to detect stale results.

By default, the Model Testing Dashboard requests that you enable artifact tracing the first time you open a project in the dashboard. Click **Enable and Continue** to allow the Model Testing Dashboard to track tool outputs to detect outdated metric results.

The dashboard needs to track tool outputs, such as test results from Simulink Test, to detect outdated metric results.

You can also enable artifact tracing from the **Startup and Shutdown** settings of the project. In the **Startup and Shutdown** settings for your project, select **Track tool outputs to detect outdated results**. For more information on the tool outputs and outdated metric results, see "Trace Dependencies Between Project Files and Identify Outdated Metric Results" on page 5-69.

Project Requires Analysis by the Dashboard

The first time that you open the dashboard for the project, the dashboard identifies the artifacts in the project and collects traceability information. The dashboard must perform this first-time setup to

establish the traceability data before it can monitor the artifacts. If you cancel the first-time setup, the artifacts in the project appear in the **Unanalyzed** folder in the **Artifacts** panel. To trace the unanalyzed artifacts, click **Collect > Trace Artifacts**.

Incorrect List of Models in Artifacts Panel

The **Artifacts** panel shows the models in your project that are either unit models or component models. Models are organized under the components that reference them, according to the model reference hierarchy. If the list of unit and component models does not show the expected hierarchy of your models, try one of these solutions.

Check that your unit and component models are labeled

Label the unit and component models in your project and configure the Model Testing Dashboard to recognize the labeled models. Note that if a unit model references one or more other models, the referenced models appear in the **Design** folder under the unit model. For more information about labeling models and configuring the dashboard, see "Categorize Models in a Hierarchy as Components or Units" on page 5-90. Check that if you have Observer models, they are not labeled as units. The dashboard includes Observer models as units if they match the label requirements.

Check that your model was saved in a supported release

Check that your model was saved in R2012b or later. The Model Testing Dashboard does not support models that were saved before R2012b.

Block Skipped During Artifact Analysis

If a block has a mask and the mask hides the content of the block, the dashboard excludes the block from artifact analysis.

Check that your custom libraries do not contain blocks with self-modifiable masks. The Model Testing Dashboard does not analyze blocks that contain self-modifiable masks. Self-modifiable masks can change the structural content of a block, which is incompatible with artifact traceability analysis.

Library Missing from Artifacts Panel

Check that the library does not use a library forwarding table. The Model Testing Dashboard does not support library forwarding tables.

Requirement Missing from Artifacts Panel

If a requirement is missing from the **Artifacts** panel, try one of these solutions.

Check that the requirement is a functional requirement

Verify that the requirement is configured as a functional requirement. In the Requirement Editor, on the left pane, click the requirement. On the right pane, in the **Properties** section, set **Type** to `Functional`. Because the Model Testing Dashboard reports on requirements-based unit testing, only functional requirements appear in the **Artifacts** panel and are analyzed by the dashboard.

Check that the requirement is saved in a supported requirements file

Verify that the requirement is saved in a requirements file that has the `.slreqx` extension.

Test Case Missing from Artifacts Panel

Check that the test case is supported by the Model Testing Dashboard. The Model Testing Dashboard does not support MATLAB-based Simulink tests.

Test Harness Missing from Artifacts Panel

Check that the test harness is an external test harness. The Model Testing Dashboard does not show internal test harnesses in the **Artifacts** panel. Note that the Model Testing Dashboard shows the test cases from both internal and external test harnesses in the **Tests** folder.

Check that the test harness is not on a subsystem inside a library block instance. If a test harness is on a subsystem inside a library block inside a model, the dashboard cannot perform artifact traceability analysis on the test harness. The relationship between a model and a test harness on a subsystem inside a library block instance is incompatible with artifact traceability analysis. To enable artifact traceability analysis, move the test harness to the library.

Test Result Missing from Artifacts Panel

Check that either:

- The result is saved in a test results file. Save test results by exporting them from the Test Manager.
- You collected the results during the current project session and have not closed them. When you collect test results and do not export them, the dashboard recognizes the temporary results in the Test Manager, denoted by the icon. The dashboard stops recognizing the temporary results when you close the project, close the test results set, or export the test results to a results file.

Artifact Returns a Warning

Check the details of the warning by opening the **Diagnostics** pane.

Artifact Returns an Error

Check the details of the error by opening the **Diagnostics** pane.

If the dashboard returns an error in the **Diagnostics** pane, the metric data shown by the dashboard widgets may be incomplete. Errors indicates that the dashboard may not have been able to properly trace artifacts, analyze artifacts, or collect metrics.

Before using the metrics results shown in the dashboard, resolve any reported errors and retrace the artifacts.

Fix ambiguous links

Check that the links in your project define unambiguous relationships between project artifacts.

In requirements-based testing, projects often contain links between software requirements and:

- the design artifacts that implement the requirements
- the test cases that test the implemented requirements
- the higher-level system requirements

The links in your project help to define the relationships between artifacts. The Model Testing Dashboard uses a digital thread to capture the traceability relationships between the artifacts in your

project. To maintain the traceability relationships, the dashboard returns an error when the links to project artifacts are ambiguous. Ambiguous links are not supported in the dashboard.

If one of these conditions is met, the dashboard cannot establish unambiguous traceability:

- A link set shadows another loaded link set of the same name.
- A requirement set shadows another loaded requirement set of the same name.
- A link is not on the project path or is only temporarily on the project path.
- A link is not portable.

  To avoid links that are not portable:

  - Do not set the preference for a link path format to be an absolute path. Absolute paths are not portable. For information on how to set the preference for the path format of links, see `rmipref` and "Document Path Storage" (Requirements Toolbox).
  - When you identify the source artifact of a link set, use the default link file name and location. Link source remapping persists in the MATLAB preferences directory and is not portable. For more information, see "Requirements Link Storage" (Requirements Toolbox).

Use the details and suggested actions in the dashboard error messages to fix the ambiguous links.

For more information on traceability relationships and the digital thread, see "Trace Dependencies Between Project Files and Identify Outdated Metric Results" on page 5-69.

Trace Issues

If an artifact appears in the **Trace Issues** folder when you expect it to trace to a unit model, depending on the type of artifact that is untraced, try one of these solutions.

Fix an untraced requirement

Check that the requirement traces to the unit model using an implementation link. The requirement and its links must meet one of these criteria:

- The requirement is linked to the model or to a library subsystem used by the model with a link where the **Type** is set to `Implements`.
- The requirement is the child of a container requirement that is linked to the model or to a library subsystem used by the model with a link where the **Type** is set to `Implements`.
- The requirement traces to the model through a combination of the previous two criteria. For example, a requirement that is under a container requirement that links to another requirement, which links to the model.

Requirements-based testing verifies that your model fulfills the functional requirements that it implements. Because the Model Testing Dashboard reports on requirements-based testing quality, it analyzes only requirements that are specified as functional requirements and implemented in the unit. For each unit, the dashboard shows the functional requirements that are implemented in the unit in the folder **Functional Requirements > Implemented**.

Check that the requirement does not use an unsupported link. The Model Testing Dashboard does not trace these links:

- Downstream links. The Model Testing Dashboard traces only links from lower-level requirements to higher-level requirements. For each unit, the dashboard shows the higher-level functional

requirements in the folder **Functional Requirements > Upstream**. Upstream links are directly or transitively linked to the implemented functional requirements for the unit. Check the direction of the link by using the Requirements Editor. In the Requirements Editor, in the **Analysis** section, click **Traceability Diagram**. In the **Impact Analysis** section, use the buttons **Upstream** and **Downstream** to check the direction of the links.

- Embedded links, which are requirements files that are saved directly in the model file.
- Links to requirements that are saved externally and linked using the Requirements Management Interface (RMI).
- Links to and from data dictionaries.
- Links to MATLAB code files.
- Links to MATLAB Function blocks if you do not have a Stateflow license. Analyzing MATLAB Function blocks requires a Stateflow license.
- Links to some Stateflow elements.
- Links in deprecated requirement files, which have the extension `.req`. To analyze requirement links in the dashboard, save the links in an `.slmx` file or create them in the requirements file (`.slreqx`) that has the requirements.
- Links to models for which the model file extension changed. If a requirement is linked to a model with the file extension `.slx`, but the model file extension is changed to `.mdl`, the dashboard lists the requirement link as unresolved. Modify the requirement link to reference the correct model file and re-save the requirement link.
- Symbolic file links in a project, such as shortcuts.
- Links to modeling elements that are not supported by the Model Testing Dashboard, such as library forwarding tables.

Fix an untraced requirement link set

Check that the requirement link set does not use the legacy Requirements Management Interface (RMI) format. To allow the dashboard to analyze your requirement link set, pass your requirement link set as the input argument to the function `slreq.refreshSourceArtifactPath`.

Fix an untraced design artifact

Check that the design artifact does not rely on a model callback to be linked with the model. The Model Testing Dashboard does not execute model loading callbacks when it loads the models for analysis. If a model relies on a callback to link a data dictionary, the data dictionary will not be linked when the dashboard runs the traceability analysis.

Fix an untraced test case

Check that the test case runs on the model or runs on a subsystem in the model by using a test harness.

Fix an untraced test result

Check that the project and test case are set up correctly and re-run your tests. If one of these conditions is met when you run your test case, the generated results are untraced because the dashboard cannot establish unambiguous traceability to the unit:

- No project is loaded.
- Artifact tracing is not enabled for the project. If artifact tracing is not enabled, the dashboard cannot track changes or trace from the test cases to the generated test results. For more information, see "Enable Artifact Tracing for the Project" on page 5-96.

- You do not have a Simulink Check license.
- The test file is stored outside the project.
- The test file has unsaved changes.
- The tested model has unsaved changes.
- The test file returns an error during traceability analysis.
- The tested model returns an error during traceability analysis.
- The test result comes from a test case that is not supported by the Model Testing Dashboard, such as a MATLAB-based Simulink test.

Check that the results and environment are set up correctly and re-export your test results. If one of these conditions is met when you export your test results, the generated results are untraced because the dashboard cannot establish unambiguous traceability to the unit:

- No project is loaded.
- Artifact tracing is not enabled for the project. For more information, see "Enable Artifact Tracing for the Project" on page 5-96.
- You do not have a Simulink Check license.
- The test result file returns an error during traceability analysis.

Metric Does Not Report Results for Requirement, Test Case, or Test Result

If an artifact traces to one of your unit models but does not appear in the metric results for that unit, depending on the type of artifact, try one of these solutions.

Fix a requirement that does not produce metric results

Check that the requirement directly links to the model with a link where the **Type** is set to `Implements`. The dashboard metrics analyze only implemented functional requirements. For each unit, the implemented functional requirements appear in the folder **Functional Requirements > Implemented**. Upstream requirements appear in the folder **Functional Requirements > Upstream**, but do not contribute to the metric results because upstream requirements are only indirectly or transitively linked to the implemented requirements.

Fix a test case that does not produce metric results

Check that the test case directly tests either the entire unit model or the model subsystems. The dashboard metrics analyze only unit tests. For each unit, the unit test cases appear in the folder **Tests > Unit Tests**. Other test cases, that are not unit tests, appear in the folder **Tests > Others**, but do not contribute to the metric results because other tests do not directly test the unit model or the model subsystems. For example, the dashboard does not consider tests on a library to be unit tests.

Fix a test result that does not produce metric results

Check that the results meet these criteria:

- The results are the most recent results generated from the test cases.
- The results are from unit tests which appear in the folder **Tests > Unit Tests**. Test cases in the folder **Tests > Others** do not contribute to the metric results.

If a test case includes multiple iterations, the metric results reflect the status of the whole test case and do not show individual iteration results.

For each unit, the test results that produce metric results appear in the folder **Test Results > Unit Simulation**. The test results in the folder **Test Results > Others** do not contribute to the metric results.

Fix a test that does not produce simulation test result analysis metric results

Check that the test case is a unit test and produces simulation results. For each unit, the metrics analyze the test results in the folder **Test Results > Unit Simulation**. The test results in the folder **Test Results > Others** are results that are not simulation results, are not from unit tests, or are only reports. For example, SIL results are not simulation results. The metrics in the **Simulation Test Result Analysis** section count results from only simulation tests, whereas the metrics in the **Test Case Analysis** section count all unit tests.

If a test is not counted in the metrics in the **Simulation Test Result Analysis** section, check that the test case meets these criteria for being a simulation test:

• The simulation mode is Normal, Accelerator, or Rapid Accelerator. If the test uses iterations to set a different simulation mode after testing one of these modes, the test is still considered a simulation test.

• The test is not a real-time test.

• If the test is an equivalence test, the first simulation meets one of the first two criteria.

• If the test contains multiple iterations, the test case or at least one iteration meets one of the first two criteria.

Metric Result Shows a Missing Link or Artifact

The metric results do not count all types of traceability links. If a metric shows that a test case or requirement is missing links when you expect it to be linked, try one of these solutions.

Fix a link that is not counted in traceability results

If there is a link between a requirement and a test case, but the traceability metrics show that the test or requirement is unlinked, check if the link is supported by the dashboard metrics. The metrics do not support these links:

• A requirement link to a justification. If a requirement is linked with a justification and not linked to a test case, it appears as unlinked in the metric results.

• A requirement link to a test harness.

Fix missing model coverage in test results

If the model coverage metrics report coverage gaps that you do not expect, re-run the test cases and re-collect the metric results for the new test results. The Model Testing Dashboard might show coverage gaps if:

• You change the test results file or the coverage filter file after you collect the metrics, including if you re-import the test results file after you make changes.

• You collect accumulated coverage results and make changes to the model file after running one or more tests.

See Also
"Model Testing Metrics"

Related Examples

- "Fix Requirements-Based Testing Issues" on page 5-61
- "Manage Requirements-Based Testing Artifacts for Analysis in the Model Testing Dashboard" on page 5-68

Collecting Requirements-Based Testing Metrics Using Continuous Integration

Overview

Requirements-based testing metrics allow you to assess the status and quality of your requirements-based testing activities. You can visualize the results by using the Model Testing Dashboard and integrate metric collection by using continuous integration workflows. Continuous collection of these metrics helps you to monitor the progression and quality of a project. This example uses GitLab® to host the project source and Jenkins® to build and test the project as well as archive the results.

Requirements

• Use of MATLAB® projects
• Use of the Simulink Test Manager test harness

Set Up the Project in Source Control

GitLab Setup

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

1 Install the Git Client.
2 Set up a branching workflow. Using 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 main branch. For more information, see https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows.
3 Under **Settings > Repository**, protect the main branch by enforcing the use of merge requests when developers want to merge their changes into the main branch.
4 Under **Settings > Integrations**, add a webhook to the URL of your Jenkins project. The webhook triggers a build job on Jenkins.

Add the Project

This example uses the example project for the dashboard. To create a working copy of the project, at the command line, enter:

```
dashboardCCProjectStart('incomplete')
```

Add all of the files in the project along with the files attached to this example to the main branch by using Git™. These scripts are used to run tests and collect the metrics.

Derived Artifact Filtering

Collecting metrics generates files that you typically do not want checked into source control. Git allows you to ignore files by adding filters to a text file named `.gitignore` located at the root directory. You can add the sample `.gitignore` file attached to this example which will filter the files generated by this example that do not need to be added to source control. For more information on `.gitignore` files, see https://git-scm.com/docs/gitignore.

Set Up the Project in the Continuous Integration Tool

The continuous integration tool automates the building and testing of the project. Many different tools can be used to automatically generate requirements-based testing results by following the same

general steps. In this example, use Jenkins as the automation tool. To run the example, you must install the GitLab and MATLAB plugins for Jenkins.

Creating the Project

The CI tool will need integration to the source control repository of the project. The integration allows the CI tool to listen to changes and access the project to build. Jenkins provides a Freestyle project which serves as a generic template for projects which can work with any source control management (SCM). In the Freestyle project, add the source control information to enable the SCM to access the hosted project.

1 Click **New Item**, fill in the name, and choose Freestyle project. Or, for an existing Freestyle Jenkins project, click **Configure**.
2 Click the **Source Code Management** tab and specify the URL of your GitLab repository in the **Repository URL** field.
3 Click the **Build Triggers** tab and select **Build when a change is pushed to GitLab**.
4 Click the **Build Environment** tab, select **Use MATLAB version**, and provide the **MATLAB root** to specify the MATLAB version for the build.

Building and Testing

The MATLAB plugin for Jenkins allows integration by specifying MATLAB commands as well as configuring testing, bypassing the need to use the command line. In this example, a single build step will be used to open the project, initialize the metrics infrastructure, run tests, and then collect results.

In the **Build** section, select **Add build step > Run MATLAB Command**. In the **Command** field, enter:

```
openProject(pwd);collectModelTestingResults();runTests();collectModelTestingResults();
```

Archiving and Consuming Metric Results

Metrics results can be archived during the build step and then reimported into MATLAB when you want to review them. In this example, the result collection script stores the metric data in the `derived` directory. Because some of the metrics rely on exported Simulink Test results, include the exported `.mldatx` files in the archive.

To archive results for later review, configure the CI system to export these files:

- All files located in `<project-root>/derived` directory.
- All test results exported to `<project-root>/testresults/.mldatx` files.

For this example, use the Jenkins provided post-build action to archive artifacts produced during the build.

Click the **Post-build Actions** tab and click **Add post-build action**. Choose **Archive the artifacts.** Enter the path:

```
derived/**,testresults/*.mldatx
```

to archive all files saved to that directory.

Click the **Save** button to save and close the configuration.

Running a Build Job in Jenkins

Jenkins is now configured to execute a new build job each time new changes to the project have been committed to the GitLab repository. You can also manually run a build by clicking on **Build Now** in the Jenkins project page.

Reviewing the Archived Results in MATLAB

Jenkins will store all the files generated and archived for each successful build and these can be viewed individually or downloaded together in a single zip file. To view the results in MATLAB:

1   Get the version of the project that was used to generate the results from source control.
2   Get the archived metric results from the archived location.
3   Download and copy or extract the `derived` directory and all files into the root directory of the project.
4   Download the archived, exported, Simulink Test results files and copy or extract these files.
5   Open the project in MATLAB and open the Model Testing Dashboard. The dashboard displays the results generated from the CI build.

Alternative CI Integration Using Command Line

If you use a different automation tool, you can alternatively use the command line for testing integration. Run the tests and collect metrics by running the appropriate commands through the command line interface along using the `-batch` flag.

For example, when you use this command, MATLAB opens the project, initializes the model testing results, runs all of the tests, collects the model metrics, and then shuts down.

```
matlab -c %LICENSE_PATH% -nosplash -logfile output.log -batch
"openProject(pwd);collectModelTestingResults();runTests();collectModelTestingResults(); exit;"
```

Hide Requirements Metrics in the Model Testing Dashboard and in API Results

If you do not want to track requirements metrics in the Model Testing Dashboard, or cannot because of a missing Requirements Toolbox license, you can hide the requirements metrics. When you hide the requirements metrics, the dashboard displays only the metrics in the **Test Case Breakdown**, **Model Test Status**, and **Model Coverage** sections. This example shows how to hide the requirements metrics when you use the Model Testing Dashboard or when you programmatically collect metrics.

If you want to collect requirements-based metrics, see "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51.

Open the Dashboard for the Project

1 Open a project that contains models and testing artifacts. For this example, at the MATLAB command line, enter:

```
dashboardCCProjectStart
```

2 Open the Model Testing Dashboard by using one of these approaches:

- On the **Project** tab, click **Model Testing Dashboard**.
- At the command line, enter:

```
modelTestingDashboard
```

Hide Requirements Metrics in the Model Testing Dashboard

1 Open the Project Options dialog box by clicking **Options**.
2 In the **Layout** section, select **Hide requirements metrics** and click **Apply**.

Note Because the setting of the **Hide requirements metrics** property is saved in the project data, the dashboard displays or hides requirements-based metrics for any users who view the project.

The dashboard shows only the widgets for the **Test Case Breakdown**, **Model Test Status**, and **Model Coverage** sections.
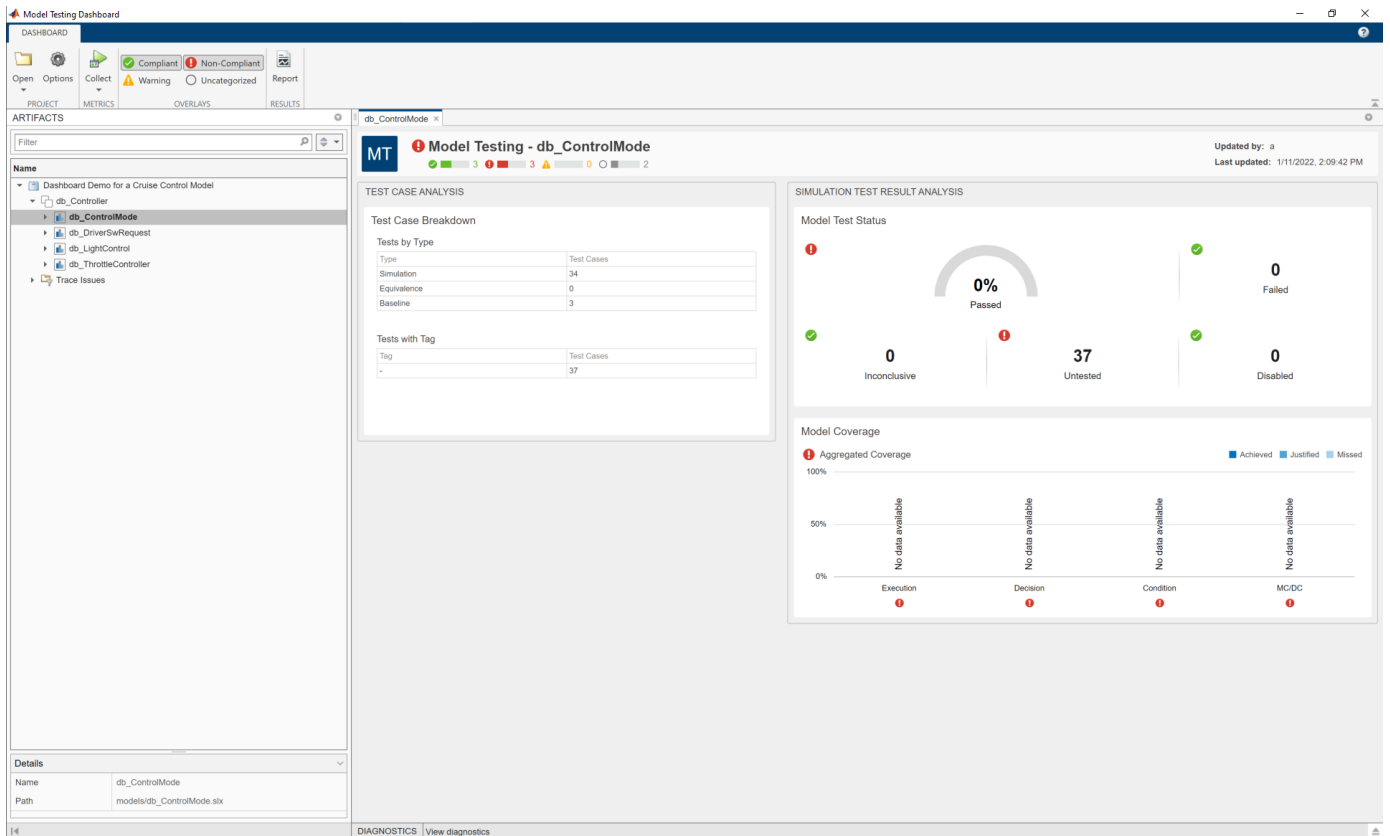
Hide Requirements Metrics in the API Results

You can also filter requirements metrics when you collect metric data by using the API.

1 Create a `metric.Engine` object for the current project.

```
metric_engine = metric.Engine();
```

2 Create a list of the metric identifiers that are not associated with requirements metrics. Use the function `getAvailableMetricIds` and specify the `'App'` as `'DashboardApp'` and the `'Dashboard'` as `'ModelUnitTesting'`.

```
nonRequirementMetrics = getAvailableMetricIds(metric_engine,...
'App','DashboardApp',...
'Dashboard','ModelUnitTesting');
```

3 Collect the results for each unit in the project. Use the `nonRequirementsMetrics` argument to exclude the requirements metrics from the results.

```
execute(metric_engine, nonRequirementMetrics);
```

4 Get the metric results for metric identifiers that are not associated with requirements metrics.

```
results = getMetrics(metric_engine, nonRequirementMetrics)
```

For more information about the metric API, see "Collect Metrics on Model Testing Artifacts Programmatically" on page 5-86.

See Also

`metric.Engine` | `execute` | `getAvailableMetricIds` | `getMetrics`

Related Examples

- "Collect Metrics on Model Testing Artifacts Programmatically" on page 5-86
- "Explore Status and Quality of Testing Activities Using the Model Testing Dashboard" on page 5-51
- "Fix Requirements-Based Testing Issues" on page 5-61
- "Model Testing Metrics"

Create Model Advisor Checks

Overview of the Customization File for Custom Checks

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

Function	Description	When Required
`sl_customization()`	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at start-up. See "Define Custom Model Advisor Checks" on page 6-45.	Required for programmatic customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See "Define Custom Model Advisor Checks" on page 6-45.	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 "Define Custom Model Advisor Checks" on page 6-45.	Required for custom checks. You must write one callback function for each custom check
One or more calls to check input parameters	Specifies input parameters to custom checks. See "Define Check Input Parameters" on page 6-48.	Optional
One or more calls to checklist views	Specifies calls to the Model Advisor Result Explorer for custom checks.	Optional
One or more calls to check actions	Specifies actions the software performs for custom checks. See "Define Custom Model Advisor Checks" on page 6-45.	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.

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 "Simulink Identifiers".
Stateflow API (Stateflow)	Programmatic access to Stateflow objects.

Review a Model Against Conditions that You Specify with the Model Advisor

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

Create an sl_customization Function

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

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

% Copyright 2019 The MathWorks, Inc.

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


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

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

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

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

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

```
function definePassFailCheck
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('simplePassFailCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Warn if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
```

6-5

```
rec.setCallbackFcn(@simplePassFailCheck,'None','DetailStyle')

mdladvRoot.publish(rec, 'Demo');

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

This check identifies Constant block values that are numbers and produces a warning, but it does not provide a fix action. For more information on how to create a check definition function with a fix, see "Fix a Model to Comply with Conditions that You Specify with the Model Advisor" on page 6-21.

Create the Check Definition Function for an Informational Check

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

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

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

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

```
function defineInformationCheck

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

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

end

% ----------------------------
% This callback function uses the DetailStyle CallbackStyle type.
% ----------------------------
```

```
function modelVersionChecksumCallbackUsingFT_Detail(system,CheckObj)

model = bdroot(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
ElementResults = ModelAdvisor.ResultDetail;
ElementResults.Description = 'Display model configuration and checksum information';

% If running the Model Advisor on a subsystem, add note to description.
if strcmp(system, model) == false
    ElementResults.IsInformer = true;
    ElementResults.Status = 'NOTE: The Model Advisor is reviewing a subsystem, but these results are based on root-level
    ElementResults(end + 1) = ModelAdvisor.ResultDetail;
end

% If error is encountered, use these values.
mdlver = 'Error - could not retrieve Version';
mdlauthor = 'Error - could not retrieve Author';
mdldate = 'Error - could not retrieve Date';
mdlsum = 'Error - could not retrieve CheckSum';

% Get model configuration and checksum information.
try
    mdlver = get_param(model,'ModelVersion');
    mdlauthor = get_param(model,'LastModifiedBy');
    mdldate = get_param(model,'LastModifiedDate');
    mdlsum = Simulink.BlockDiagram.getChecksum(model);
    mdlsum = [num2str(mdlsum(1)) ' ' num2str(mdlsum(2)) ' ' ...
                 num2str(mdlsum(3)) ' ' num2str(mdlsum(4))];
    ElementResults(end).IsInformer = true;
    mdladvObj.setCheckResultStatus(true);
catch
    ElementResults(end).IsViolation = true;
    mdladvObj.setCheckResultStatus(false);
end

lbStr ='<br/>';
resultStr = ['Model Version: ' mdlver lbStr 'Author: ' mdlauthor lbStr ...
        'Date: ' mdldate lbStr 'Model Checksum: ' mdlsum];
ElementResults(end).Status = resultStr;
CheckObj.setResultDetails(ElementResults);

end
```

Run the Custom Checks in the Model Advisor

1   In the Command Window, enter:

    ```
    Advisor.Manager.refresh_customizations
    ```

2   Open the model sldemo_fuelsys by typing this command in the MATLAB command prompt:

    ```
    sldemo_fuelsys
    ```

3   In the **Modeling** tab, select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Click **OK**.

4   In the left pane, select **By Product > Demo > Identify model configuration and checksum information**.

5   Click **Run Checks**.

    The check passes and displays the information.

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

7   Click **Run Checks**.

The check produces a warning because several blocks contain values that are numbers. The results contain links to these blocks. The result displays a **Recommended Action**.

8 Follow the **Recommended Action** to fix the Constant blocks.

See Also

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

More About

- "Define Custom Model Advisor Checks" on page 6-45
- "Fix a Model to Comply with Conditions that You Specify with the Model Advisor" on page 6-21
- "Create and Deploy a Model Advisor Custom Configuration" on page 7-23

Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor

In this example, you create three custom edit-time checks that check for compliance with certain software design standards. Custom edit-time checks help you catch issues earlier in the model design review process, but these checks also report issues in the Model Advisor.

The first check checks that Inport and Outport blocks have certain colors depending on their output data types.

The second check checks whether a Trigger block is higher than other blocks in a subsystem. This check must check edited blocks and other blocks in the same subsystem as the Trigger block.

The third check checks whether signals that connect to Outport blocks have labels.

Register and Define the Custom Edit-Time Checks

1   To register the custom edit-time check, create an `sl_customization` function. The `sl_customization` function accepts one argument, a customization manager object. To register the custom check, use the `addModelAdvisorCheckFcn` method. The input to this method is a handle to the check definition function. For this example, `defineCheck` is the check definition function. Create the `sl_customization` function and save it to your working folder.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineCheck);
```

2   Create the check definition function. Inside the function, create three `ModelAdvisor.Check` objects and specify the check IDs as input arguments. Then, specify the `ModelAdvisor.Check Title` and `CallbackHandle` properties. The `CallbackHandle` property is the name of the class that you create to define the edit-time check. For this example, `MyEditTimeChecks` is the package name and `PortColor`, `TriggerBlockPosition`, and `SignalLabel` are the class names. Then, publish the checks to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit Time Checks**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck

%% Check the background color of Inport and Outport blocks.
rec = ModelAdvisor.Check("advisor.edittimecheck.PortColor");
rec.Title = 'Check color of Inport and Outport blocks';
rec.CallbackHandle = 'MyEditTimeChecks.PortColor';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec,'DEMO: Edit Time Checks');

%% Check that determines whether Trigger block is the top-most block in a subsystem.
rec= ModelAdvisor.Check("advisor.edittimecheck.TriggerBlock");
rec.Title = 'Check that Trigger block position is higher than other blocks';
rec.CallbackHandle = 'MyEditTimeChecks.TriggerBlockPosition';
mdladvRoot.publish(rec,'DEMO: Edit Time Checks');

%% Check that determines whether signals with SignalPropagation 'on' have labels.
rec = ModelAdvisor.Check("advisor.edittimecheck.SignalLabel");
rec.Title = 'Check that signals have labels if they are to propagate those labels';
rec.CallbackHandle = 'MyEditTimeChecks.SignalLabels';
mdladvRoot.publish(rec,'DEMO: Edit Time Checks');
```

3   Create the first check by creating a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `PortColor.m`. Copy the code below into the `PortColor.m` file. Then, create a folder named `+MyEditTimeChecks` and save the `PortColor.m` file in that folder. Classes must be in a folder that has the same name as their package name.

The `PortColor` class defines three methods: `PortColor`, `blockDiscovered`, and `fix`. The `PortColor` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.BLKITER` because the check must check newly added and edited blocks, but it does not have to check for affected blocks in the same subsystem or model as the edited or newly added blocks. The `blockDiscovered` method contains an algorithm that checks the color of Inport and Outport blocks. Then, because the violation is on a block, the algorithm highlights a violating block by creating a `ModelAdvisor.ResultDetail` violation object with the `Type` property set to the default value of `SID`. The `fix` method updates blocks that do not have correct colors.

Check that Import Blocks Have Certain Colors Class Definition

```matlab
classdef PortColor < ModelAdvisor.EdittimeCheck
    % Check that ports conform to software design standards for background color.
    %
    %   Background Color              Data Types
    %   orange                        Boolean
    %   green                         all floating-point
    %   cyan                          all integers
    %   Light Blue                    Enumerations and Bus Objects
    %   white                         auto
    %


    methods
        function obj=PortColor(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            if strcmp(get_param(blk,'BlockType'),'Inport') || strcmp(get_param(blk,'BlockType'),'Outport')

                dataType = get_param(blk,'OutDataTypeStr');
                currentBgColor = get_param(blk,'BackgroundColor');

                if strcmp(dataType,'boolean')
                    if ~strcmp(currentBgColor, 'orange')
                        % Create a violation object using the ModelAdvisor.ResultDetail class.
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with Boolean outputs should be orange.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                elseif any(strcmp({'single','double'},dataType))
                    if ~strcmp(currentBgColor, 'green')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with floating-point outputs should be green.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
                    if ~strcmp(currentBgColor, 'cyan')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with integer outputs should be cyan.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                elseif contains(dataType,'Bus:')
                    if ~strcmp(currentBgColor, 'lightBlue')
```

```matlab
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with bus outputs should be light blue.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
              elseif contains(dataType,'Enum:')
                if ~strcmp(currentBgColor, 'lightBlue')
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with enumeration outputs should be light blue.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
            elseif contains(dataType, 'auto')
                if ~strcmp(currentBgColor, 'white')
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with auto outputs should be white.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
            end
        end
    end

    function violation = finishedTraversal(obj)
        violation = [];
    end

    function success = fix(obj, violation)
        success = true;
        dataType = get_param(violation.Data,'OutDataTypeStr');
        if strcmp(dataType,'boolean')
            set_param(violation.Data,'BackgroundColor','orange');
        elseif any(strcmp({'single','double'},dataType))
            set_param(violation.Data,'BackgroundColor','green');
        elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
            set_param(violation.Data,'BackgroundColor','cyan');
        elseif contains(dataType,'Bus:')  || contains(dataType,'Enum:')
            set_param(violation.Data,'BackgroundColor','lightBlue');
        elseif contains(dataType,'auto')
            set_param(violation.Data,'BackgroundColor','white');
        end
    end
  end
end
```

4 Create a second check by creating another class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `TriggerBlockPosition.m`. Copy the code below into the `TriggerBlockPosition.m` file and save it to the +MyEditTimeChecks folder.

The `TriggerBlockPosition` class defines three methods: `TriggerBlockPosition`, `blockDiscovered`, and `finishedTraversal`. The `TriggerBlockPosition` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.ACTIVEGRAPH` because it must check other blocks in the same subsystem as the Trigger block. The `blockDiscovered` method checks the position of Trigger blocks within subsystems. The `finishedTraversal` method checks whether the position of these Trigger blocks are higher than other blocks in a subsystem. Then, because the violation is on a block, the algorithm highlights a violating block by creating a `ModelAdvisor.ResultDetail` violation object with the `Type` property set to the default value of `SID`.

Check the Position of Trigger Blocks Class Definition

```matlab
classdef TriggerBlockPosition < ModelAdvisor.EdittimeCheck
    properties
        TriggerBlock = [];
```

```
            position = [];
        end

    methods
        function obj=TriggerBlockPosition(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.ACTIVEGRAPH;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            if strcmp(get_param(blk,'BlockType'),'TriggerPort')
                obj.TriggerBlock = blk;
            else
                h = get_param(blk,'Position');
                obj.position = [obj.position, h(2)];
            end
        end

        function violation = finishedTraversal(obj)
            violation = [];
            if isempty(obj.TriggerBlock)
                return;
            end
            triggerPosition = get_param(obj.TriggerBlock,'Position');
            if min(obj.position) < triggerPosition(2)
                violation = ModelAdvisor.ResultDetail;
                ModelAdvisor.ResultDetail.setData(violation,'SID',...
                    Simulink.ID.getSID(obj.TriggerBlock));
                violation.CheckID = obj.checkId;
                violation.title = 'Trigger Block Position';
                violation.Description = 'Trigger Block should be top block in subsystem';
                violation.ViolationType = 'Warning';
            end
            obj.TriggerBlock = [];
            obj.position =[];
        end
    end
end
```

5 Create a third and final check by creating another class that derives from the
   `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named
   `SignalLabel.m`. Copy the code below into the `SignalLabel.m` file and save it to the
   `+MyEditTimeChecks` folder.

   The `SignalLabel` class defines two methods: `SignalLabel` and `blockDiscovered`. The
   `SignalLabel` method sets the `CheckId` and `TraversalType` properties. This check has a
   traversal type of `edittimecheck.TraversalTypes.BLKITER` because the check must check
   newly added and edited blocks, but it does not have to check for affected blocks in the same
   subsystem or model as the edited or newly added blocks. Because this check is for signals, the
   `blockDiscovered` method must use the parameters on the line handles, `LineHandles`, of
   blocks to find signals with violations. Specifically, for signals that connect to Outport blocks, this
   algorithm checks whether the `Name` signal parameter has a value. Then, because the violation is
   on a signal, the algorithm highlights the signal by creating a violation object with the `Type`
   property value set to `Signal`.

Check the Labels of Signals That Connect to Outport Blocks Class Definition

```
classdef SignalLabels < ModelAdvisor.EdittimeCheck
    methods
        function obj=SignalLabels(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            ports    = get_param(blk,'Ports');
            lh = get_param(blk, 'LineHandles');
            if strcmp(get_param(blk,'BlockType'),'Outport')
                for j = 1 : ports(1)
                    if lh.Inport(j) ~= -1 % failure case: no connection
                        allsources = get_param(lh.Inport(j),'SrcPortHandle');
                        hiliteHandle = get_param(lh.Inport(j), 'DstPortHandle');
                        if (isempty(allsources) ~= 0) || (isempty(find(allsources==-1,1)) ~= 0)
                            lh_obj = get_param(lh.Inport(j),'Object');
                            if isempty(lh_obj.Name)
                                if strcmp(lh_obj.signalPropagation,'off') == 1
                                    allsources_parent = get_param(allsources,'Parent');
                                    if strcmp(get_param(allsources_parent,'BlockType'),'Inport')
                                        buscreator_outputs = get_param(allsources_parent,'IsBusElementPort');
                                    else
                                        buscreator_outputs = 'off';
                                    end
                                    if ~strcmp(buscreator_outputs,'on')
                                        violation = ModelAdvisor.ResultDetail;
                                        ModelAdvisor.ResultDetail.setData(violation,'Signal',hiliteHandle);
                                        violation.Description ='This signal should have a label.';
                                        violation.CheckID = obj.checkId;
                                        violation.Title = 'Signal Label Missing';
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

Run the Edit-Time Checks on a Model

1 To use the checks, copy the `AdvisorCustomizationExample.slx` model to your current working folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```

2 Refresh the Model Advisor to update the cache with the new checks on the path.

```
Advisor.Manager.refresh_customizations
```

3 Open the Model Advisor Configuration Editor by entering this command at the command prompt:

```
Simulink.ModelAdvisor.openConfigUI;
```

If a model is open, you can also open the editor by clicking the **Modeling** tab and selecting **Model Advisor > Configuration Editor**.

4 Create a custom configuration consisting of the three custom edit-time checks by deleting every folder except for the **DEMO: Edit Time Checks** folder.

5 Save the configuration as `my_config.json`.

6 When prompted to set this configuration as the default, click **No**.

7 Close the Model Advisor Configuration Editor.

8 Open the model by entering this command at the command prompt.

```
open_system('AdvisorCustomizationExample.slx');
```
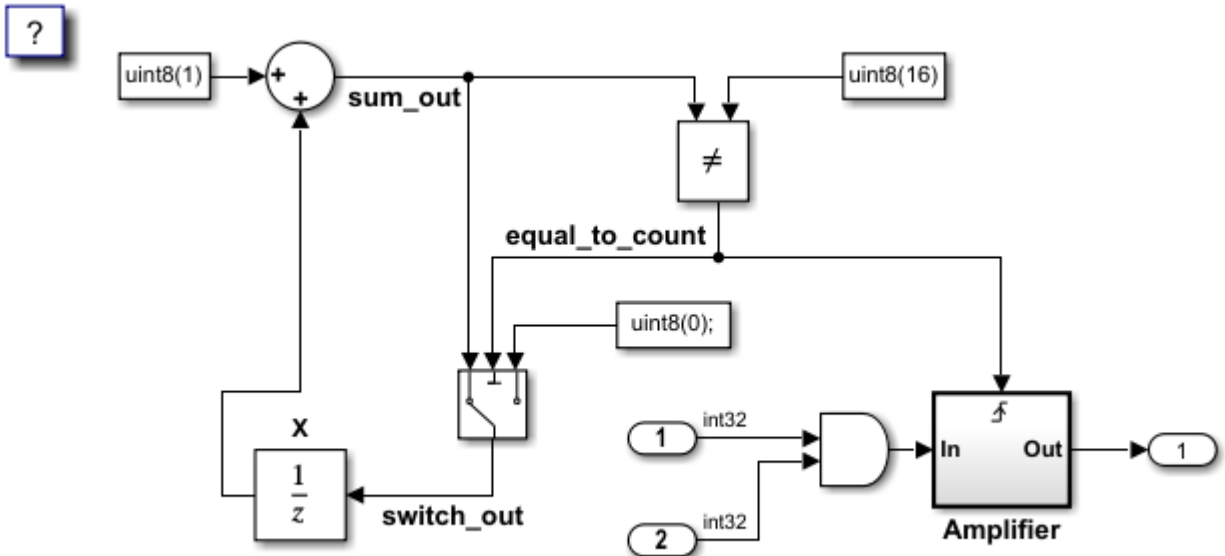

**9** Set the custom configuration to the my_config.json file by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, specify the path to the configuration file in the **Model Advisor configuration file** parameter. Alternatively, enter this command at the command prompt:

```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', 'my_config.json');
```

10 Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, select the **Edit-Time Checks** parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('AdvisorCustomizationExample','on');
```

11 Open the Model Advisor by clicking the **Modeling** tab and selecting **Model Advisor**. Observe that the three edit-time checks are the only ones in the Model Advisor. Close the Model Advisor.

12 To view the edit-time warnings, click the blocks and signals highlighted in yellow.
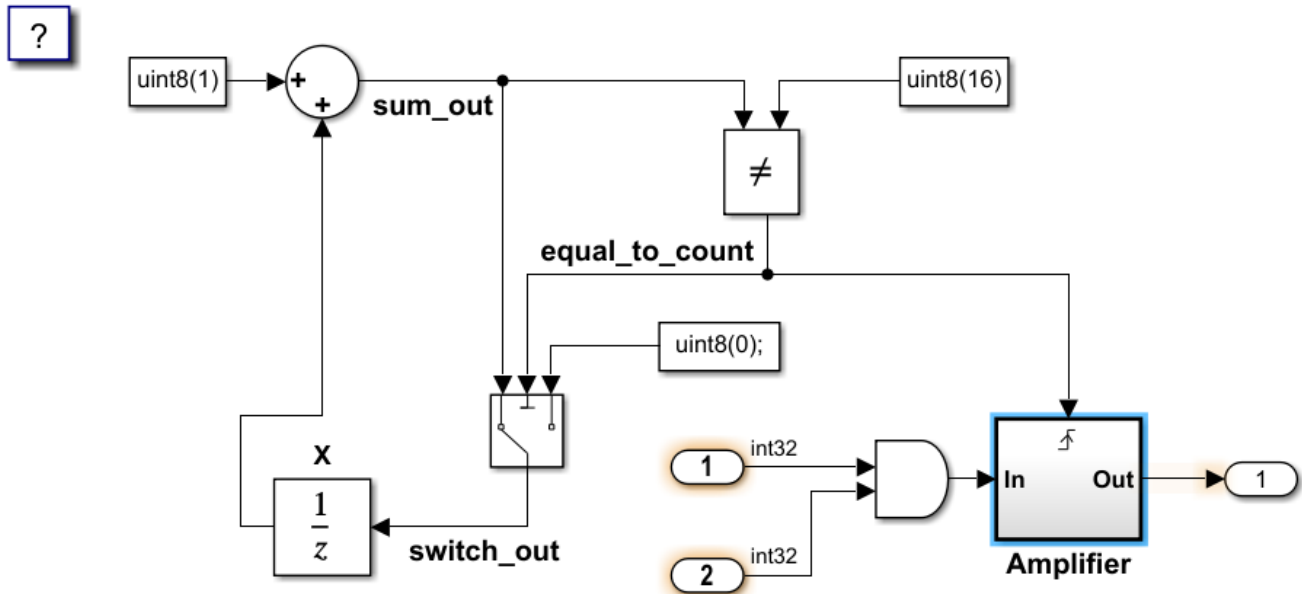
At the top level of the model, the two Inport blocks have an output data type of `int32`. They produce edit-time warnings because they should be cyan. The Outport block does not produce a violation because it has an `auto` data type and is white.

In the **Amplifier** subsystem, the Inport and Outport blocks do not produce edit-time warnings because they have a data type of `auto` and are white. The Trigger block does not produce an edit-time warning because it is the top-most block in the model. If you move the Trigger block below another block, the Trigger block has an edit-time warning.

The signal connecting to the Outport block produces a warning because it does not have a label.

13 To fix the edit-time warnings for the two Inport blocks, in the edit-time check warning window, click **Fix**.

Performance Considerations for Custom Edit-Time Checks

To help prevent custom edit-time checks from negatively impacting performance as you edit your model, the Model Advisor automatically disables custom edit-time checks if, in the current MATLAB session, the check takes longer than `500` milliseconds to execute in at least three different Simulink models.

If the Model Advisor disables a custom edit-time check, you will see a warning on the Simulink canvas. You can re-enable the edit-time check by either:

- Clicking the hyperlink text in the warning.
- Passing the check identifier, `checkID`, to the function `editime.enableCheck`:

  ```
  edittime.enableCheck(checkID)
  ```

To prevent a custom edit-time check from being disabled, author the check so that the check executes in less than `500` milliseconds on your models.

See Also

ModelAdvisor.EdittimeCheck | ModelAdvisor.Check | ModelAdvisor.ResultDetail

Related Examples

- "Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor" on page 6-9
- "Define Custom Model Advisor Checks" on page 6-45
- "Define Custom Edit-Time Checks that Fix Issues in Architecture Models" on page 6-17

Define Custom Edit-Time Checks that Fix Issues in Architecture Models

This example shows how to create a custom edit-time check that runs on architecture models that you create using System Composer. Edit-time checks help you catch issues earlier in the model design review process. The custom edit-time check in this example produces a warning if the names of connecting block port interface names do not match. For more information on the process for authoring custom checks, see "Define Custom Model Advisor Checks" on page 6-45.

Create a Simple Architecture Model

Create a simple architecture model with mismatched data interface names in ports that share a connector.

1   Create a temporary working directory.

2   In MATLAB, on the **Home** tab, click **Simulink**.

3   In the Simulink Start page, click **System Composer** and select **Architecture Model**.

4   Add three Component blocks.

5   Connect the outport of one Component block to the inport ports of the other two Component blocks as shown in this image.
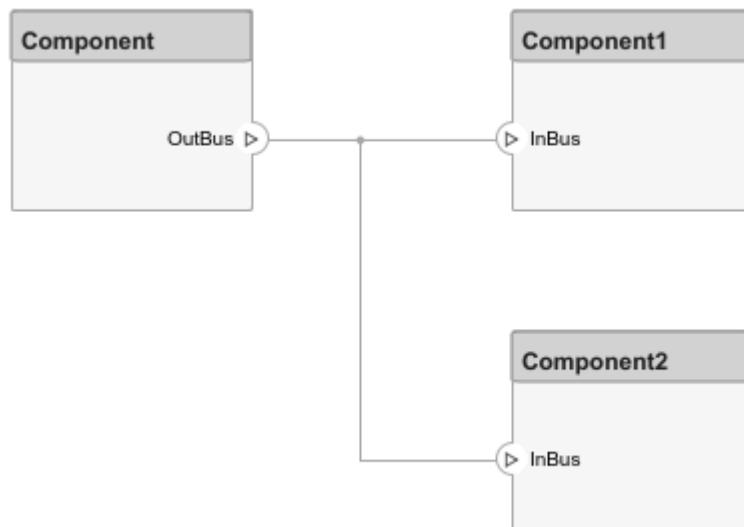
6   On the **Modeling** tab, click **Interface Editor**.

7   Create two data interfaces with the names `interface0` and `interface1`.

8   Open the Property Inspector.

9   For `Component`, click the `OutBus` port. In the **Interface** section of the Property Inspector, for the **Name** field, select `interface0`.

10   For `Component1`, click the `InBus` port. In the **Interface** section of the Property Inspector, for the **Name** field, select `interface1`.

11   For `Component2`, click the `InBus` port. In the **Interface** section of the Property Inspector, for the **Name** field, select `interface0`.

12 Save the model to your working directory. For this example, the model name is `myModel.slx`.

Create the Custom Edit-Time Check

Create a check that detects the mismatched data interface names in ports that share the same connector while a user is editing a model.

1 To register the custom edit-time check, create an `sl_customization` function. The `sl_customization` function accepts one argument, a customization manager object. To register the custom check, use the `addModelAdvisorCheckFcn` method. The input to this method is a handle to the check definition function. For this example, `defineCheck` is the check definition function. Create the `sl_customization` function and save it to your working folder.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineCheck);
```

2 Create the check definition function. Inside the function, create a `ModelAdvisor.Check` object and specify the Check ID as an input argument. Then, specify the `ModelAdvisor.Check Title` and `CallbackHandle` properties. The `CallbackHandle` property is the name of the class that you create to define the edit-time check. For this example, `MyEditTimeChecks` is the package name and `PortMismatch` is the class name. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **System Composer Edit-time Check**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
rec = ModelAdvisor.Check("advisor.edittimecheck.SystemComposerPortMismatch");
rec.Title = 'Check port mismatch for system composer components';
rec.CallbackHandle = 'MyEditTimeChecks.PortMismatch';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec,'System Composer Edit-time Check');
end
```

3 Create a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `PortMismatch`. Copy the code below into the `PortMismatch.m` file. Then, create a folder named +MyEditTimeChecks and save the `PortMismatch.m` file in that folder. The class must be in a folder that has the same name as the package name.

The `PortMismatch` class defines two methods: `PortMismatch` and `blockDiscovered`. The `PortMismatch` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.ACTIVEGRAPH` because the check must check newly added and edited blocks and affected blocks in the same subsystem or model. The `blockDiscovered` method contains an algorithm that checks whether the port interface names match.

```
classdef PortMismatch < ModelAdvisor.EdittimeCheck

    methods
        function obj=PortMismatch(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.ACTIVEGRAPH;
        end

        function violationArray = blockDiscovered(obj,blk)
            violationArray = [];
            blkHdl = get_param(blk, 'Handle');
            archMdl = systemcomposer.arch.Model(bdroot(blk));
            comp = archMdl.lookup('SimulinkHandle',blkHdl);
            if isa(comp, 'systemcomposer.arch.Component')
                for i = 1:length(comp.Ports)
```

```
                    compPort = comp.Ports(i);
                    if strcmp(compPort.Direction, 'Output')
                        srcInterfaceName = compPort.InterfaceName;
                        for j = 1:length(compPort.Connectors)
                            connector = compPort.Connectors(j);
                            destPort = connector.DestinationPort;
                            destInterfaceName = destPort.InterfaceName;
                            if(~strcmpi(srcInterfaceName, destInterfaceName))
                                hiliteHandle = destPort.SimulinkHandle;
                                violation = ModelAdvisor.ResultDetail;
                                ModelAdvisor.ResultDetail.setData(violation, 'Signal',hiliteHandle);
                                violation.CheckID = obj.checkId;
                                violation.Description = 'Connected port interface names should be the same.';
                                violation.Title = 'Port Interface Mismatch';
                                violation.ViolationType = 'warn';
                                violationArray = [violationArray violation]; %#ok<AGROW>
                            end
                        end
                    end
                else
                    compPort = comp;
                    if strcmp(compPort.Direction, 'Output')
                        srcInterfaceName = compPort.InterfaceName;
                        for j = 1:length(compPort.Connectors)
                            connector = compPort.Connectors(j);
                            destPort = connector.DestinationPort;
                            destInterfaceName = destPort.InterfaceName;
                            if(~strcmpi(srcInterfaceName, destInterfaceName))
                                hiliteHandle = destPort.SimulinkHandle;
                                violation = ModelAdvisor.ResultDetail;
                                ModelAdvisor.ResultDetail.setData(violation,'Signal',hiliteHandle);
                                violation.CheckID = obj.checkId;
                                violation.Description = 'Connected port interface names should be the same.';
                                violation.Title = 'Port Interface Mismatch';
                                violation.ViolationType = 'warn';
                                violationArray = [violationArray violation]; %#ok<AGROW>
                            end
                        end
                    end
                end
            end
        end
    end
end
```

Create a Custom Edit-Time Check Configuration

Create a custom configuration consisting of the edit-time check. Associate the configuration with the `myModel.slx` model.

1   Refresh the Model Advisor to update the cache with the new check on the path.

    `Advisor.Manager.refresh_customizations`

2   Open the Model Advisor Configuration Editor by clicking the **Modeling** tab and selecting **Model Advisor** > **Configuration Editor** or by entering this command at the command prompt:

   `Simulink.ModelAdvisor.openConfigUI;`

3   Create a custom configuration that consists only of the custom edit-time check. Select the **Product** > **System Composer Edit-time Checks** folder and then delete the other folders. Save the configuration as `sc_config.json`. Close the Model Advisor Configuration Editor.

4   Set the custom configuration to the `sc_config.json` file by clicking the **Modeling** tab and selecting **Model Advisor** > **Edit-Time Checks**. In the Configuration Parameters dialog box that opens, specify the path to the configuration file in the **Model Advisor configuration file** parameter. Alternatively, enter this command at the command prompt:

```
ModelAdvisor.setModelConfiguration('myModel', 'sc_config.json');
```

5   Turn on edit-time checking by selecting the **Model Advisor > Edit Time** configuration parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('myModel','on');
```

6   To view the edit-time warnings, click the signal highlighted in yellow.
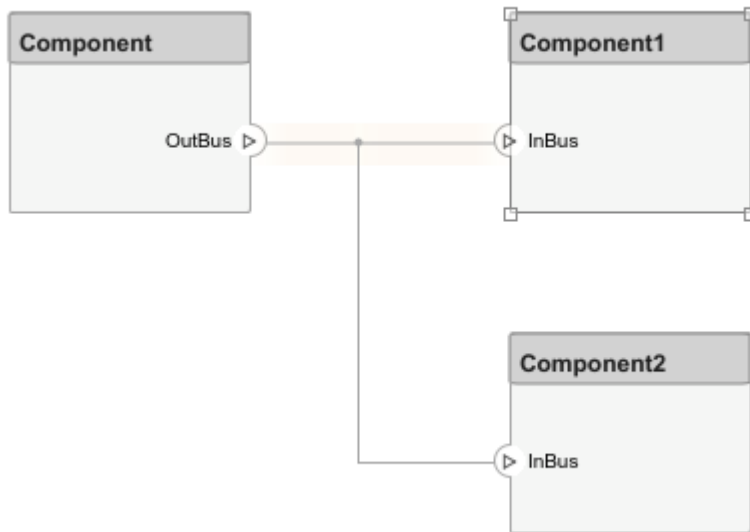
The connector between the `Component` and `Component1` blocks produces a warning because the data interface names in each port do not match.

See Also
ModelAdvisor.EdittimeCheck | ModelAdvisor.Check | ModelAdvisor.ResultDetail

Related Examples

- "Define Custom Model Advisor Checks" on page 6-45
- "Run Custom Model Advisor Checks on Architecture Models" on page 3-106
- "Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor" on page 6-9

Fix a Model to Comply with Conditions that You Specify with the Model Advisor

This example shows how to create a customized Model Advisor pass/fail check with a fix action. When a model does not contain a check violation, the results contain the check description and result status. When a model contains a check violation, the results contain the check description, result status, and the recommended action to fix the issue. This example adds a custom check to a Model Advisor **By Product > Demo** subfolder.

For this example, the custom check identifies blocks whose names do not appear below the blocks. The fix action is to make the block names appear below the blocks.

When a check does not pass, the results include a hyperlink to each model element that violates the check. Use these hyperlinks to easily locate areas in your model or subsystem. The code for this example consists of an `sl_customization.m` file and a `defineDetailStyleCheck.m` file.

Create the sl_customization File

1   In your working folder, create an `sl_customization.m` file.
2   To register the custom checks, create an `sl_customization(cm)` function as shown here. This function accepts one argument, a customization manager object. The customization manager object includes the `addModelAdvisorCheckFcn` method for registering the custom check. The input to this method is a handle to the function `defineModelAdvisorChecks`. `defineModelAdvisorChecks` contains a call to the check definition function for custom Model Advisor pass/fail check.

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

% Copyright 2019 The MathWorks, Inc.

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

% ----------------------------
% defines Model Advisor Checks
% ----------------------------
function defineModelAdvisorChecks
    defineDetailStyleCheck;
```

Create the Check Definition File

The check definition function defines the check and fix actions that the Model Advisor takes when you run the check. For this example, the completed check definition function file is `defineDetailStyleCheck.m`, and it contains this code:

```
function defineDetailStyleCheck

mdladvRoot = ModelAdvisor.Root;

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
```

```matlab
% Create ModelAdvisor.Action object for setting fix operation.
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
rec.setAction(myAction);

mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

end

% -----------------------------
% This callback function uses the DetailStyle CallbackStyle type.
% -----------------------------
function DetailStyleCallback(system, CheckObj)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

% Find all blocks whose name does not appear below blocks
violationBlks = find_system(system, 'Type','block',...
    'NamePlacement','alternate',...
    'ShowName', 'on');
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block.';
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
else
    for i=1:numel(violationBlks)
        ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)
        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below the block.';
        ElementResults(i).Status = 'The following blocks have names that do not display below the blocks:';
        ElementResults(i).RecAction =  'Change the location such that the block name is below the block.';
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end

% -----------------------------
% This action callback function changes the location of block names.
% -----------------------------
function result = ActionCB(taskobj)
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
resultDetailObjs = checkObj.ResultDetails;
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block,'NamePlacement','normal');
end

result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
mdladvObj.setActionEnable(false);
end
```

The following steps explain how to create the defineDetailStyleCheck.m file.

1 Create a ModelAdvisor.Root object.

```matlab
mdladvRoot = ModelAdvisor.Root;
```

2 Create a ModelAdvisor.Check object and define the unique check ID. For this check, the ID is com.mathworks.sample.detailStyle.

```matlab
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
```

3 Specify the `ModelAdvisor.Check.Title` and `ModelAdvisor.Check.TitleTips` properties.

```
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
```

4 Use the `setCallbackFcn` method to call the callback function. The `setCallbackFcn` method arguments are a handle to the callback function and the `ModelAdvisor.Check.CallbackStyle` property value. For this example, the `CallbackStyle` property value is `DetailStyle`. This style allows you to view results by block, subsystem, or recommended action. Applying this style produces default formatting, so that you do not have to use the `ModelAdvisor.FormatTemplate` class or the other Model Advisor formatting APIs to format the results that appear in the Model Advisor.

```
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
```

5 To set the fix operation, create a `ModelAdvisor.Action` object and define its properties. Use the `setCallback` method to call the action callback function. The input to this method is a handle to the action callback function.

```
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
```

6 Use the `setAction` method to set the action for the check.

```
rec.setAction(myAction);
```

7 Use the `publish` method to publish the check to a folder within the **By Product** folder. For this example, the folder name is **Demo**.

```
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.
```

Create the Check Callback Definition Function

1 In the `defineDetailStyleCheck.m` file, create the check callback function. In this example, the function name is `DetailStyleCallback`. The inputs to this function are a `ModelAdvisor.CheckObject` and the path to the model or system that the Model Advisor analyzes.

```
function DetailStyleCallback(system, CheckObj)
```

2 To create a `Simulink.ModelAdvisor` object, use the function `Simulink.ModelAdvisor.getModelAdvisor`.

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

3 To identify blocks that violate the check, use the `find_system` function. For each model element, this function creates a `ModelAdvisor.ResultDetail` object.

```
violationBlks = find_system(system, 'Type','block',...
                           'NamePlacement','alternate',...
                           'ShowName', 'on');
```

4 Write code for the case when the `find_system` function does not identify blocks whose names do not appear below the block. In this case, `ElementResults` is one instance of a `ModelAdvisor.ResultDetail` object and provides information content only. The method specifies that there is no check violation and displays **Passed** in the Model Advisor.

```
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block.';
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
```

5 Write code for the case when the `find_system` function returns a list of blocks whose names do not appear below the block (`violationBlks`). `ElementResults` includes each `ModelAdvisor.ResultDetail` object that violates the check and provides a recommended action message for fixing the check violation.

For this case, the `setCheckResultStatus` method specifies the check violation and displays **Warning** or **Failed** in the Model Advisor. The `Simulink.ModelAdvisor.setActionEnable(true)` method enables the ability to fix the check violation issue from the Model Advisor.

```
else
    for i=1:numel(violationBlks)
     ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)
        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below the block.';
        ElementResults(i).Status = 'The following blocks have names that do not display below the blocks:';
        ElementResults(i).RecAction =  'Change the location such that the block name is below the block.';
    end
mdladvObj.setCheckResultStatus(false);
mdladvObj.setActionEnable(true);
end
```

6 To associate the results with a check object, use the `setResultDetails` method.

```
CheckObj.setResultDetails(ElementResults);
end
```

Create the Action Callback Definition Function

1 In the `defineDetailStyleCheck.m` file, create the action callback function. In this example, the function name is `sampleActionCB`. The input to this function is a `ModelAdvisor.Task` object.

```
function result = ActionCB(taskobj)
```

2 Create handles to `Simulink.ModelAdvisor` and `ModelAdvisor.Check` objects.

```
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
```

3 Create an array of `ModelAdvisor.ResultDetail` objects for storing the information for blocks that violate the check.

```
resultDetailObjs = checkObj.ResultDetails;
```

4 Write code that changes the block name location to below the block.

```
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block,'NamePlacement','normal');
end
result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
```

5 Disable the **Action** box.

```
mdladvObj.setActionEnable(false);
```

Run the Check

1 Save the `sl_customization.m` and `defineDetailStyleCheck.m` files.

2   In the MATLAB command window, enter:

`Advisor.Manager.refresh_customizations`

3   Open the model `sldemo_fuelsys` by typing this command in the MATLAB command prompt:

`sldemo_fuelsys`

4   In the top model, select the block named `Engine Speed`. In the toolstrip, on the **Format** tab, click **Flip Name**.

5   Open the `fuel_rate_control` subsystem. Select the block named `validate_sample_time`. In the toolstrip, on the **Format** tab, click **Flip Name**.

Return to the top model and save as `example_sldemo_fuelsys`.

6   In the **Modeling** tab, select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Click **OK**. The Model Advisor opens.

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

8   Select **Run Checks**. The Model Advisor check produces a warning for the blocks that you changed.

9   Review the results by selecting either the **Report** or **Result Detail** tabs.

Both tabs provide a recommended action for each block that violates the check. You can click the hyperlink path to open the block in the model editor. For example:

10 Follow the recommended action for fixing the violating blocks by using one of these methods:

- Update each violation individually by double-clicking the hyperlink to open the block. Select the block. In the toolstrip, on the **Format** tab, select **Flip Name**.
- In the toolstrip, click **Fix**. The Model Advisor automatically fixes the issues in the model. Notice that the button is dimmed after the violations are fixed.

11 Rerun the Model Advisor check. The check passes.
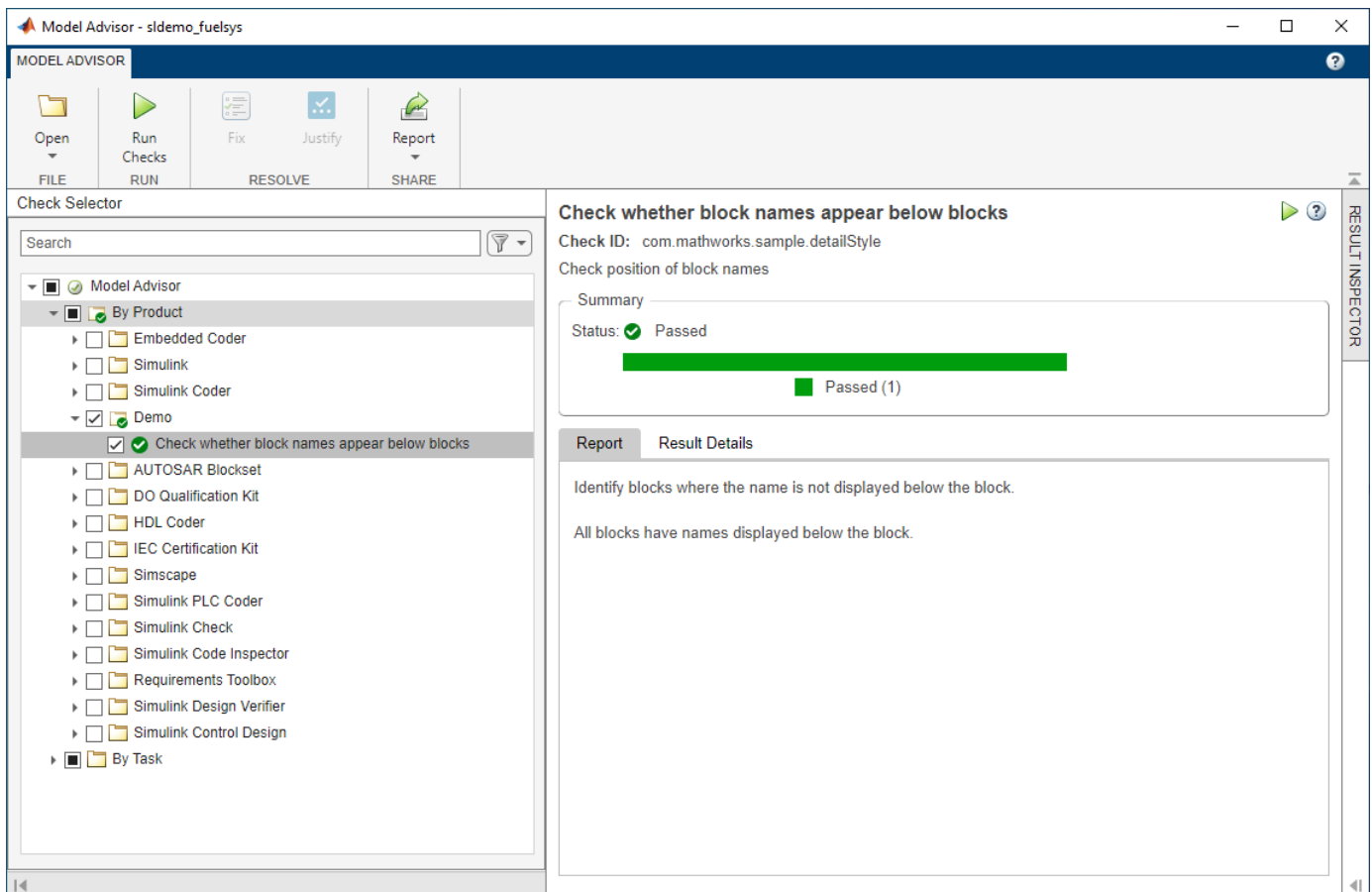
See Also

ModelAdvisor.Check | ModelAdvisor.FormatTemplate | ModelAdvisor.FormatTemplate | ModelAdvisor.Check.CallbackContext | Simulink.ModelAdvisor | Simulink.ModelAdvisor.getModelAdvisor | Simulink.ModelAdvisor.openConfigUI | Simulink.ModelAdvisor.reportExists

More About

- "Define Custom Model Advisor Checks" on page 6-45
- "Review a Model Against Conditions that You Specify with the Model Advisor" on page 6-5
- "Create and Deploy a Model Advisor Custom Configuration" on page 7-23

Create Model Advisor Check for Model Configuration Parameters

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

Decide which configuration parameter settings to use for your model. If desired, review the modelling guidelines:

- MathWorks Advisory Board (MAB) Modeling Guidelines
- High-Integrity System Modeling Guidelines
- Code Generation Modeling Guidelines

1 Create an XML data file containing the configuration parameter settings you want to check. You can use `Advisor.authoring.generateConfigurationParameterDataFile` or manually create the file yourself.

2 Register the model configuration parameter check using an `sl_customization.m` file.

3 Run the check on your models.

Create a Data File for a Configuration Parameter Check

This example shows how to create a data file that specifies configuration parameter values in the **Diagnostics** pane. A custom check warns when the configuration parameters values do not match the values defined in the data file.

At the command prompt, type `vdp` to open the `van der Pol Equation` model.

Right-click in the model window and select **Model Configuration Parameters**. In the **Diagnostics** pane, set the configuration parameters as follows:

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

Use the `Advisor.authoring.generateConfigurationParameterDataFile` function to create a data file specifying configuration parameter constraints in the **Diagnostics** pane. Also, to create a check with a fix action, set `FixValue` to true. At the command prompt, type:

```
model='vdp';
dataFileName = 'ex_DataFile.xml';
Advisor.authoring.generateConfigurationParameterDataFile(dataFileName,...
model, 'Pane', 'Diagnostics', 'FixValues', true);
```

In the Command Window, select `ex_DataFile.xml`. The data file opens in the MATLAB editor.

- The **Minimize algebraic loop** (`ArtificialAlgebraicLoopMsg`) configuration parameter tagging specifies a `value` of `error` with a `fixvalue` of `error`. When you run the configuration parameter check using `ex_DataFile.xml`, the check fails if the **Minimize algebraic loop** setting is not `error`. The check fix action modifies the setting to `error`.

- The **Block Priority Violation** (`BlockPriorityViolationMsg`) configuration parameter tagging specifies a `value` of `error` with a `fixvalue` of `error`. When you run the configuration parameter check using `ex_DataFile.xml`, the check fails if the **Block Priority Violation** setting is not `error`. The check fix action modifies the setting to `error`.

In `ex_DataFile.xml`, edit the **Algebraic loop** (`AlgebraicLoopMsg`) parameter tagging so that the check warns if the `value` is `none`. Because you are specifying a configuration parameter that you do not want, you need a `NegativeModelParameterConstraint`. Also, to create a subcheck that does not have a fix action, remove the line with `<fixvalue>` tagging. The tagging for the configuration parameter looks as follows:

```
<!-- Algebraic loop: (AlgebraicLoopMsg)-->
    <NegativeModelParameterConstraint>
        <parameter>AlgebraicLoopMsg</parameter>
        <value>none</value>
    </NegativeModelParameterConstraint>
```

In `ex_DataFile.xml`, delete the lines with tagging for configuration parameters that you do not want to check. The data file `ex_DataFile.xml` provides tagging only for **Algebraic loop**, **Minimize algebraic loop**, and **Block Priority Violation**. For example, `ex_DataFile.xml` looks similar to:

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.w3schools.com
MySchema.xsd">
    <checkdata>
    <!-- Algebraic loop: (AlgebraicLoopMsg)-->
        <NegativeModelParameterConstraint>
            <parameter>AlgebraicLoopMsg</parameter>
            <value>none</value>
        </NegativeModelParameterConstraint>
    <!--Minimize algebraic loop: (ArtificialAlgebraicLoopMsg)-->
        <PositiveModelParameterConstraint>
            <parameter>ArtificialAlgebraicLoopMsg</parameter>
            <value>error</value>
            <fixvalue>error</fixvalue>
        </PositiveModelParameterConstraint>
    <!--Block priority violation: (BlockPriorityViolationMsg)-->
        <PositiveModelParameterConstraint>
            <parameter>BlockPriorityViolationMsg</parameter>
            <value>error</value>
            <fixvalue>error</fixvalue>
        </PositiveModelParameterConstraint>
    </checkdata>
</customcheck>
```

Verify the data syntax with `Advisor.authoring.DataFile.validate`. At the command prompt, type:

```
dataFile = 'ex_DataFile.xml';
msg = Advisor.authoring.DataFile.validate(dataFile);

if isempty(msg)
    disp('Data file passed the XSD schema validation.');
else
```

```
    disp(msg);
end
```

Create Check for Diagnostics Pane Model Configuration Parameters

This example shows how to create a check for **Diagnostics** pane model configuration parameters using a data file and an `sl_customization.m` file. First, you register the check using an `sl_customization.m` file. Using `ex_DataFile.xml`, the check warns when:

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

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

The check uses the `ex_DataFile.xml` data file created in "Create a Data File for a Configuration Parameter Check" on page 6-27.

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**.

```matlab
function sl_customization(cm)

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


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

Note that model configuration parameter settings checks must use the `setCallbackFcn` type of `StyleOne`.

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

```
Advisor.Manager.refresh_customizations
```

At the command prompt, type `vdp` to open the `van der Pol Equation` model.

Right-click in the model window and select **Model Configuration Parameters**. In the **Diagnostics** pane, set the configuration parameters as follows:

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

In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.

In the left pane, select **DemoExample: Check model configuration parameters**. In the right pane, **Data File** is set to `ex_DataFile.xml`.

Click **Run Checks**. 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 toolstrip, click **Fix**. The Model Advisor updates the configuration parameters for **Block priority violation** and **Minimize algebraic loop**.

Run the **Demo > Example: Check model configuration parameters** check again. The check warns because **Algebraic loop** is set to `none`.

In the right pane of the Model Advisor window, use the `Algebraic loop (AlgebraicLoopMsg)` link to edit the configuration parameter. Set **Algebraic loop** to `warning` or `error`.

Run the check a final time. The check passes.

Data File for Configuration Parameter Check

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

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

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

The `<messages>` tag contains:

- `Description` - (Optional) Description of the check. Displayed in Model Advisor window.
- `PassMessage` - (Optional) Pass message displayed in Model Advisor window.
- `FailMessage` - (Optional) Fail message displayed in Model Advisor window.
- `RecommendedActions` - (Optional) Recommended actions displayed in Model Advisor window when check does not pass.

Note The `<messages>` tag is optional.
`Advisor.authoring.generateConfigurationParameterDataFile` does not generate
`<messages>` tagging.

In the `<checkdata>` tag, the data file specifies two types of constraints:

- `PositiveModelParameterConstraint` - Specifies the configuration parameter setting that you want.
- `NegativeModelParameterConstraint` - Specifies the configuration parameter setting that you do not want.

Within the tag for each of the two types of constraints, for each configuration parameter that you want to check, the data file has the following tags:

- `parameter` - Specifies the configuration parameter that you want to check. The tagging uses the command line name for the configuration parameter. For example:

```
<PositiveModelParameterConstraint>
    <parameter>BlockPriorityViolationMsg</parameter>
</PositiveModelParameterConstraint>
<NegativeModelParameterConstraint>
    <parameter>AlgebraicLoopMsg</parameter>
</NegativeModelParameterConstraint>
```

- `value` - Specifies the setting(s) for the configuration parameter. You can specify more than one `value` tag.

  When using `PositiveModelParameterConstraint`, `value` specifies the setting(s) that you want for the configuration parameter. For `NegativeModelParameterConstraint`, `value` specifies the setting(s) you that do not want for the configuration parameter.

  You can specify the `value` using a format in this table.

Type	Format	Example

Scalar value	`<value>xyz</value>`	In this example, constraint `NegativeModelParameterConstraint` warns when the configuration parameter settings for configuration parameter is not `error` or `none`.
		
		`<NegativeModelParameterConstraint>`
		`    <value>error</value>`
		`    <value>none</value>`
		`</NegativeModelParameterConstraint>`

Structure or object	`<value>`
	`    <param1>xyz</param1>`
	`    <param2>yza</param2>`
	`</value>`
		In this example, constraints `PositiveModelParameterConstraint` warns when the configuration parameter settings are not a valid structure:
		
		`<PositiveModelParameterConstraint>`
		`    <value>`
		`        <double>a</double>`
		`        <single>b</single>`
		`    </value>`
		`</PositiveModelParameterConstraint>`

Array	`<value>`
	`    <element>value</element>`
	`    <element>value</element>`
	`</value>`
		In this example, constraint `NegativeModelParameterConstraint` warns when the configuration parameter settings are an invalid array:
		
		`<NegativeModelParameterConstraint>`
		`    <value>`
		`        <element>A</element>`
		`        <element>B</element>`
		`    </value>`
		`</NegativeModelParameterConstraint>`

Type	Format	Example
Structure Array	```<value> <element> <param1>xyz</param1> <param2>yza</param2> </element> <element> <param1>xyz</param1> <param2>yza</param2> </element> </value>```	In this example, constraint `NegativeModelParameterConstraint` warns when the configuration parameter settings are an invalid structure array: ```<NegativeModelParameterConstraint> <value> <element> <double>a</double> <single>b</single> </element> <element> <double>a</double> <single>b</single> </element> </value> </NegativeModelParameterConstraint>```

- `fixvalue` - (Optional) Specifies the setting to use when applying the Model Advisor fix action.

  You can specify the `fixvalue` using a format in this table.

Type	Format	Example
Scalar value	`<fixvalue>xyz</fixvalue>`	In this example, the fix action tag specifies the new configuration parameter setting as `warning`. ```<PositiveModelParameterConstraint> <value>error</value> <fixaction>warning</fixaction> </PositiveModelParameterConstraint>```
Structure or object	```<fixvalue> <param1>xyz</param1> <param2>yza</param2> </fixvalue>```	In this example, the fix action tag specifies the new configuration parameter setting for a structure. ```<PositiveModelParameterConstraint> <value> <double>a</double> <single>b</single> </value> <fixvalue> <double>c</double> <single>d</single> </fixvalue> </PositiveModelParameterConstraint>```

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

- `dependson` - (Optional) Specifies a prerequisite subcheck.

  In this example, `dependson` specifies that configuration parameter subcheck `ID_B` must pass before configuration parameter subcheck `ID_A` runs.

  ```
  <PositiveModelParameterConstraint id="ID_A">
      <dependson>ID_B</value>
  </PostitiveModelParameterConstraint>
  ```

Data file tagging specifying a configuration parameter

The following tagging specifies a subcheck for configuration parameter `SolverType`. If the configuration parameter is set to `Fixed-Step`, the subcheck passes.

```
<PositiveModelParameterConstraint id="ID_A">
    <parameter>SolverType</parameter>
    <value>Fixed-step</value>
</PostitiveModelParameterConstraint>
```

Data file tagging specifying configuration parameter with fix action

The following tagging specifies a subcheck for configuration parameter `AlgebraicLoopMsg`. If the configuration parameter is set to `none` or `warning`, the subcheck passes. If the subcheck does not pass, the check fix action modifies the configuration parameter to `error`.

```
<PositiveModelParameterConstraint id="ID_A">
    <parameter>AlgebraicLoopMsg</parameter>
    <value>none</value>
    <value>warning</value>
    <fixvalue>error</value>
</PostitiveModelParameterConstraint>
```

Data file tagging specifying an array type configuration parameter

```
<PositiveModelParameterConstraint id="A">
    <parameter>ReservedNameArray</parameter>
    <value>
        <element>A</element>
        <element>B</element>
    </value>
    <value>
        <element>A</element>
        <element>C</element>
    </value>
</PositiveModelParameterConstraint>
```

Data file tagging specifying a structure type configuration parameter with fix action

```
<PositiveModelParameterConstraint id="A">
    <parameter>ReplacementTypes</parameter>
    <value>
        <double>a</double>
        <single>b</single>
    </value>
    <value>
        <double>c</double>
        <single>b</single>
    </value>
    <fixvalue>
        <double>a</double>
        <single>b</single>
    </fixvalue>
</PositiveModelParameterConstraint>
```

Data file tagging specifying configuration parameter with fix action and prerequisite check

The following tagging specifies a subcheck for configuration parameter `SolverType`. The subcheck for `SolverType` runs only after the `ID_B` subcheck passes. If the `ID_B` subcheck does not pass, the subcheck for `SolverType` does not run. The Model Advisor reports that the prerequisite constraint is not met.

If the `SolverType` subcheck runs and `SolverType` is set to `Fixed-Step`, the `SolverType` subcheck passes. If the subcheck runs and does not pass, the check fix action modifies the configuration parameter to `Fixed-Step`.

```
<PositiveModelParameterConstraint id="ID_A">
    <parameter>SolverType</parameter>
    <value>Fixed-step</value>
    <fixvalue>Fixed-step</value>
    <dependson>ID_B</value>
</PostitiveModelParameterConstraint>
```

Data file tagging specifying unwanted configuration parameter

The following tagging specifies a subcheck for configuration parameter `SolverType`. The subcheck does not pass if the configuration parameter is set to `Fixed-Step`.

```
<NegativeModelParameterConstraint id="ID_A">
    <parameter>SolverType</parameter>
    <value>Fixed-step</value>
</NegativeModelParameterConstraint>
```

Data file tagging specifying unwanted configuration parameter with fix action

The following tagging specifies a subcheck for configuration parameter `SolverType`. If the configuration parameter is set to `Fixed-Step`, the subcheck does not pass. If the subcheck does not pass, the check fix action modifies the configuration parameter to `Variable-Step`.

```
<NegativeModelParameterConstraint id="ID_A">
    <parameter>SolverType</parameter>
    <value>Fixed-step</value>
    <fixvalue>Variable-step</value>
</NegativeModelParameterConstraint>
```

Data file tagging specifying unwanted configuration parameter with fix action and prerequisite check

The following tagging specifies a check for configuration parameter `SolverType`. The subcheck for `SolverType` runs only after the `ID_B` subcheck passes. If the `ID_B` subcheck does not pass, the subcheck for `SolverType` does not run. The Model Advisor reports that the prerequisite constraint is not met.

If the `SolverType` subcheck runs and `SolverType` is set to `Fixed-Step`, the subcheck does not pass. The check fix action modifies the configuration parameter to `Variable-Step`.

```
<NegativeModelParameterConstraint id="ID_A">
    <parameter>SolverType</parameter>
    <value>Fixed-step</value>
    <fixvalue>Variable-step</value>
    <dependson>ID_B</value>
</NegativeModelParameterConstraint>
```

See Also

```
Advisor.authoring.CustomCheck.actionCallback |
Advisor.authoring.CustomCheck.checkCallback |
Advisor.authoring.DataFile.validate |
Advisor.authoring.generateConfigurationParameterDataFile
```

More About

- "Organize and Deploy Model Advisor Checks"

Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters

You can create Model Advisor checks that check whether blocks use specific block or parameter values. You can specify constraints for:

- Supported or unsupported block parameter values
- Supported or unsupported model parameter values
- Supported or unsupported blocks
- Whether blocks or parameters meet a combination of constraints

You can also use the `addPreRequisiteConstraintID` function to add prerequisite constraints that must pass before Model Advisor checks the actual constraint. You can check your model against these constraints as you edit or by running the checks from the Model Advisor.

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 the model against those constraints.
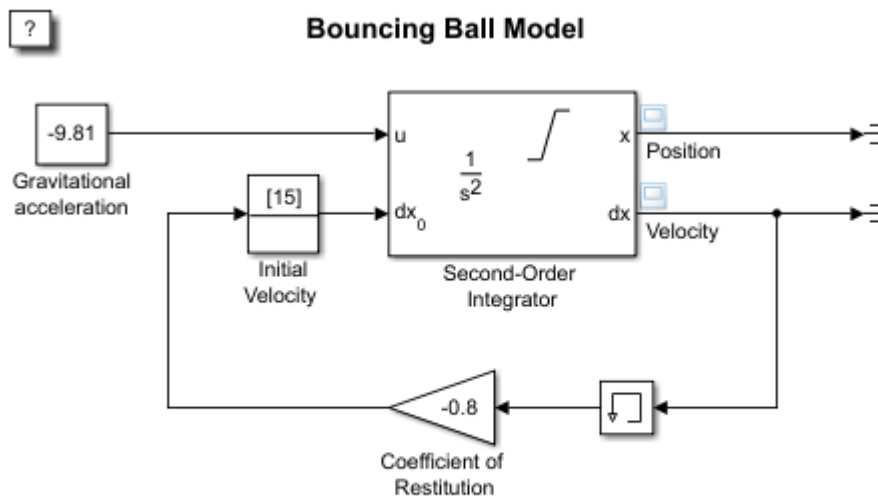
Copyright 2004-2013 The MathWorks, Inc.

Prepare Your Working Directory

1. Set your current folder to a writeable directory.

2. Copy the script `prepare_chk_define_code` to your current folder and run the script. The script copies the files necessary for this example to your current folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main','prepare_chk_define_code.m'),...
    'prepare_chk_define_code.m','f');
run('prepare_chk_define_code.m');
```

Create a Check for Supported or Unsupported Block Parameters

First, create a Model Advisor check that contains three block parameter constraints, `c1`, `c2`, and `c3`, that specify the supported and unsupported block parameter values.

1. Define a new function.

```
function constraints = createConstraints_Check1()
end
```

2. Inside the function, create two block parameter constraints, `c1` and `c2`.

```
function constraints = createConstraints_Check1()

    c1=Advisor.authoring.PositiveBlockParameterConstraint;
    c1.ID='ID_1';
    c1.BlockType='Gain';
    c1.ParameterName='Gain';
    c1.SupportedParameterValues={'-0.7'};
    c1.ValueOperator='eq'; % equal to

    c2=Advisor.authoring.NegativeBlockParameterConstraint;
    c2.ID='ID_2';
    c2.BlockType='InitialCondition';
    c2.ParameterName='Value';
    c2.UnsupportedParameterValues={'0'};
    c2.ValueOperator='le'; % less than or equal to

end
```

Constraint `c1` specifies that a Gain block must have a value equal to `-0.7`. Constraint `c2` specifies that an Initial Condition block with a value less than or equal to zero is unsupported.

3. Create a positive block constraint, `c3`, and set `constraints` equal to a cell array of constraints `c1`, `c2`, and `c3`.

```
function constraints = createConstraints_Check1()

    c1=Advisor.authoring.PositiveBlockParameterConstraint;
    c1.ID='ID_1';
    c1.BlockType='Gain';
    c1.ParameterName='Gain';
    c1.SupportedParameterValues={'-0.7'};
    c1.ValueOperator='eq'; % equal to

    c2=Advisor.authoring.NegativeBlockParameterConstraint;
    c2.ID='ID_2';
    c2.BlockType='InitialCondition';
    c2.ParameterName='Value';
    c2.UnsupportedParameterValues={'0'};
    c2.ValueOperator='le'; % less than or equal to

    c3=Advisor.authoring.PositiveBlockTypeConstraint;
    c3.ID='ID_3';
    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','');
    c3.SupportedBlockTypes={s1;s2;s3;s4;s5;s6;s7;};

    constraints = {c1,c2,c3};

end
```

Constraint `c3` specifies the supported blocks. `constraints` is a cell array of the block constraints.

4. Define a new Model Advisor check by creating another function, `check1`. Use the function `Advisor.authoring.createBlockConstraintCheck` to create a Model Advisor check, `rec`, with these block constraints. Then use `mdladvRoot.register(rec)` to register the block constraints check with the Model Advisor.

```
function check1()

    rec = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0001',...
                                'Constraints',@createConstraints_Check1);

    rec.Title = 'Example 1: Check three block parameter constraints';
    rec.TitleTips = 'Example check three block parameter constraints';

    mdladvRoot = ModelAdvisor.Root;
    mdladvRoot.publish(rec,'Example: My Group')

end

function constraints = createConstraints_Check1()

    c1=Advisor.authoring.PositiveBlockParameterConstraint;
    c1.ID='ID_1';
    c1.BlockType='Gain';
    c1.ParameterName='Gain';
    c1.SupportedParameterValues={'-0.7'};
    c1.ValueOperator='eq'; % equal to

    c2=Advisor.authoring.NegativeBlockParameterConstraint;
    c2.ID='ID_2';
    c2.BlockType='InitialCondition';
    c2.ParameterName='Value';
    c2.UnsupportedParameterValues={'0'};
    c2.ValueOperator='le'; % less than or equal to

    c3=Advisor.authoring.PositiveBlockTypeConstraint;
    c3.ID='ID_3';
    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','');
    c3.SupportedBlockTypes={s1;s2;s3;s4;s5;s6;s7;};

    constraints = {c1,c2,c3};
```

```
end
```

Create a Check for a Composite Constraint

Next, create a Model Advisor check that contains three block parameter constraints `cc1`, `cc2`, and `cc`. Constraints `cc1` and `cc2` specify which block parameters are supported and constraint `cc` is a composite constraint which contains `cc1` and `cc2`.

1. Define a new function.

```
function constraints = createConstraints_Check2()
end
```

2. Create two block parameter constraints, `cc1` and `cc2`, and a composite constraint, `cc`. Set `constraints` equal to a cell array of constraints `cc1`, `cc2`, and `cc`.

```
function constraints = createConstraints_Check2()

    cc1=Advisor.authoring.PositiveBlockParameterConstraint;
    cc1.ID='ID_cc1';
    cc1.BlockType='SecondOrderIntegrator';
    cc1.ParameterName='UpperLimitX';
    cc1.SupportedParameterValues={'inf'};
    cc1.ValueOperator='eq'; % equal to

    cc2=Advisor.authoring.PositiveBlockParameterConstraint;
    cc2.ID='ID_cc2';
    cc2.BlockType='SecondOrderIntegrator';
    cc2.ParameterName='LowerLimitX';
    cc2.SupportedParameterValues={'0.0'};
    cc2.ValueOperator='eq'; % equal to

    cc=Advisor.authoring.CompositeConstraint;
    cc.addConstraintID('ID_cc1');
    cc.addConstraintID('ID_cc2');
    cc.CompositeOperator='and'; % Model Advisor checks multiple constraints

    constraints = {cc1,cc2,cc};

end
```

Constraint `cc1` specifies that for a Second-Order Integrator block, the **Upper limit x** parameter must have a value equal to `inf`. Constraint `cc2` additionally specifies that 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` must pass. `constriants` is a cell array of the block constraints.

3. Define a new Model Advisor check in a new function, `check2`. Use the function `Advisor.authoring.createBlockConstraintCheck` to create a Model Advisor check for the block constraints defined by the function `createConstraints_Check2`.

```
function check2()

    rec = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0002',...
                                        'Constraints',@createConstraints_Check2);

    rec.Title = 'Example 2: Check three block parameter constraints';
```

```matlab
    rec.TitleTips = 'Example check three block parameter constraints';

    mdladvRoot = ModelAdvisor.Root;
    mdladvRoot.publish(rec,'Example: My Group')



end

function constraints = createConstraints_Check2()

    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';

    constraints = {cc1,cc2,cc};

end
```

Create and Run Model Advisor Checks

1. To register the new checks, use an `sl_customization.m` file. For this example, rename the `sl_customization_DefineChecks.m` file to `sl_customization.m`.

```matlab
function sl_customization(cm)

    % register custom checks.
    cm.addModelAdvisorCheckFcn(@check1);
    cm.addModelAdvisorCheckFcn(@check2);

```

2. At the command prompt, create the **Example 1: Check block parameter constraints** and **Example 2: Check block parameter constraints** checks by typing this command:

```matlab
Advisor.Manager.refresh_customizations
```

3. At the command prompt, open the model `sldemo_bounce`.

```matlab
open_system('sldemo_bounce')
```

4. In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.

5. In the left pane, select **By Product** > **Example: My Group**.

6. Click **Run Checks**.

The **Example 1: Check three block parameter constraints** check produces a warning because the Gain block has a value of -0.8. The **Example 2: Check three block parameter constraints** check passes because the Second-Order Integrator block meets both constraints.

Create Model Advisor Edit-Time Checks using Constraints

You can use edit-time checking to highlight blocks with block constraint violations in the model canvas. You can choose which Model Advisor checks evaluate during edit-time checking by selecting the desired checks in the Model Advisor Configuration Editor and saving a custom configuration.

1. To open the Model Advisor Configuration Editor, open the Model Advisor and select **Open > Open Configuration Editor**.

2. The checks you created appear in the **By Product > Example: My Group**.

3. For this example, delete all folders except for the **Example: My Group** folder.
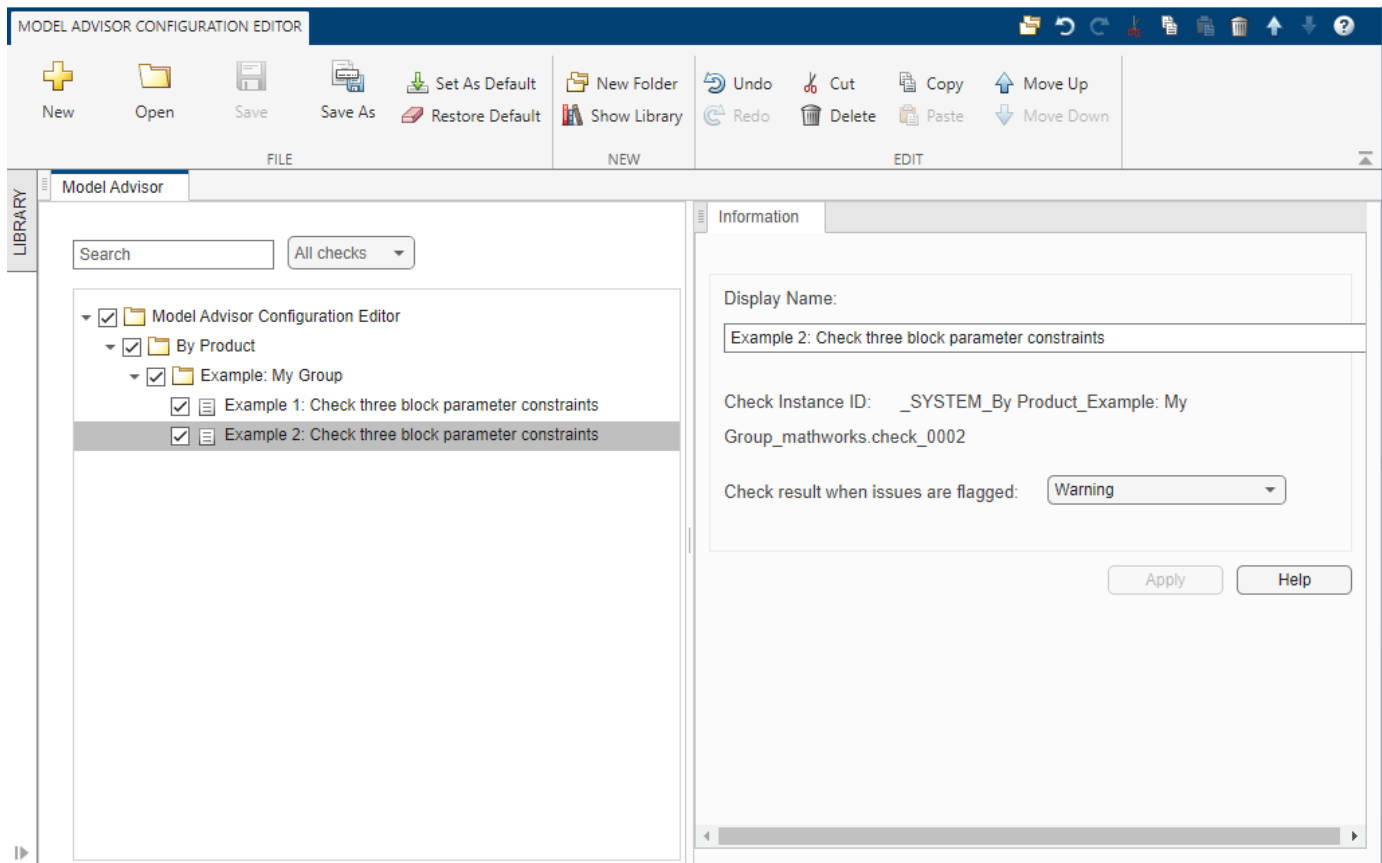
4. Click **Save** and save the configuration as my_config.json.

5. In the dialog box, click **No** because you do not want to set this configuration as the default configuration.

6. In the Simulink model editor, on the **Modeling** tab, click **Model Advisor > Edit-Time Checks**.

7. For the **Model advisor configuration file** parameter, click **Browse**, and select `my_config.json`.

8. Select the **Edit-Time Checks** parameter and close the Configuration Parameters dialog box.

9. Click on the highlighted block in the model to view the edit-time warning.
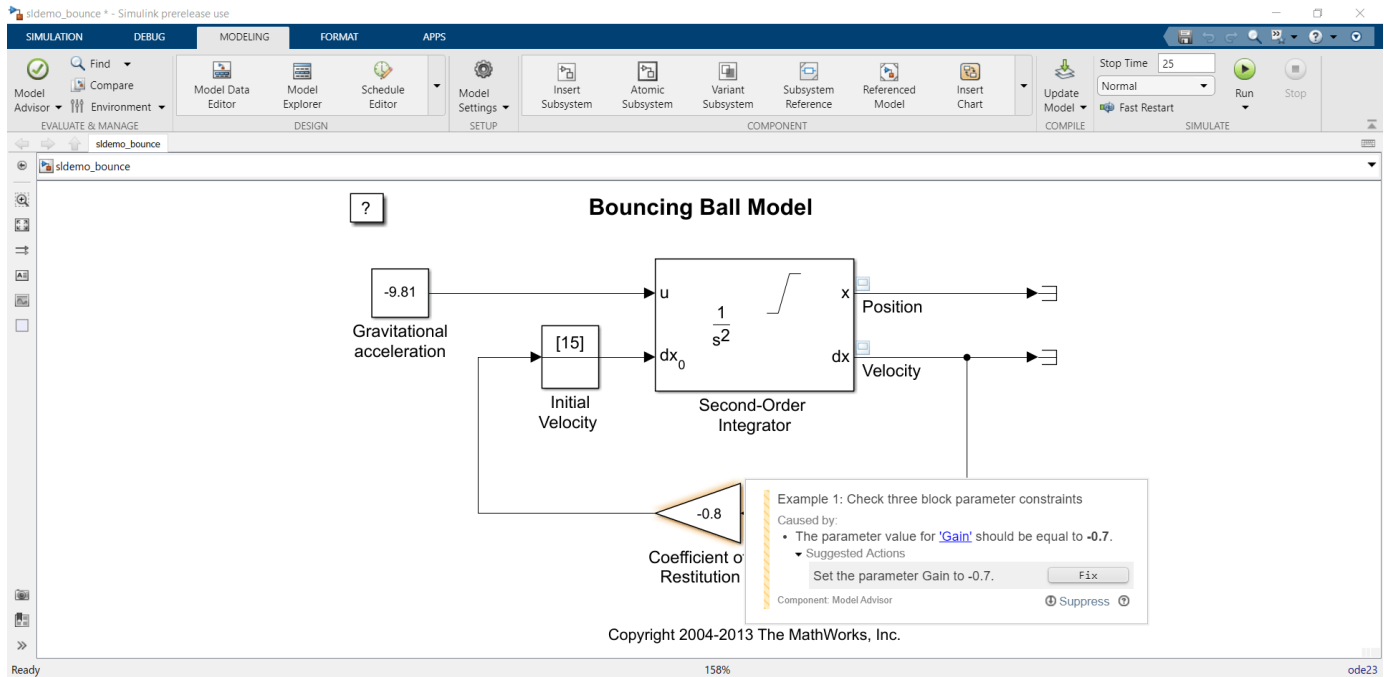
You can edit the block parameter values from the edit-time check diagnostics window by clicking the **Fix** button or by clicking the hyperlinks of the unsupported parameter to open the **Block Parameters** window.

See Also

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

More About

*   "Check Model Compliance Using Edit-Time Checking" on page 3-6
*   "Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor" on page 6-9
*   "Define Custom Edit-Time Checks that Fix Issues in Architecture Models" on page 6-17

Define Custom Model Advisor Checks

You can create your own conditions and model configuration settings for the Model Advisor to review by defining custom checks. You can create custom checks that run during edit-time and in the Model Advisor or only in the Model Advisor.

Custom edit-time checks help you identify issues earlier in the model design process, but they look only at blocks and signals at the same level of the model or subsystem that a user is editing. However, these checks do aggregate over the levels of a model hierarchy and report issues in the Model Advisor. If your check must check for impacted blocks at other levels of the model, create a custom check that runs only in the Model Advisor. For example, if your check must check for mismatched From and Goto blocks across a model hierarchy, define this check to run only in the Model Advisor.

These steps show the process for creating checks that run during edit-time or only in the Model Advisor.

1 "Create sl_customization Function" on page 6-45
2 "Register Custom Checks" on page 6-45
3 "Create Check Definition Function" on page 6-46

    a "Create an Instance of the ModelAdvisor.Check Class" on page 6-46
    b Define your custom check by following the steps in either "Define Custom Model Advisor Checks" on page 6-46 or "Define Custom Edit-Time Checks" on page 6-47.
    c "Define Check Input Parameters" on page 6-48
    d "Publish Custom Check" on page 6-49

Create sl_customization Function

To define a custom check, begin by creating an `sl_customization.m` file on the MATLAB path. In the `sl_customization.m` file, create an `sl_customization` function. The `sl_customization` function accepts one argument, a customization manager object:

```
function sl_customization(cm)
```

Tip

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

Register Custom Checks

To register custom checks, use the `addModelAdvisorCheckFcn` method, which is part of the customization manager object that you input to the `sl_customization` function. This code shows a sample `sl_customization.m` function:

```
function sl_customization(cm)
% register custom checks
```

```
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

% ----------------------------
% defines Model Advisor Checks
% ----------------------------
function defineModelAdvisorChecks
defineDetailStyleCheck;
defineConfigurationParameterCheck;
defineNewBlockConstraintCheck;
defineEditTimeChecks;
```

The `addModelAdvisorCheckFcn` method registers the checks to the **By Product** folder of the Model Advisor. The `defineModelAdvisorChecks` argument is a handle to the function that contains calls to the functions that define the custom checks. For each custom Model Advisor check that you create, you should create a check definition function. You can create one check definition function for your edit-time checks because each edit-time check contains its own class definition.

Create Check Definition Function

The check definition function defines the actions that the Model Advisor takes when you run the check. These sections describe the key components of the check definition function for custom edit-time checks and checks that run only in the Model Advisor.

Create an Instance of the ModelAdvisor.Check Class

For each custom check, create one instance of the `ModelAdvisor.Check` class. Use the `ModelAdvisor.Check` properties and methods to define the check user interface and actions. This table describes some key check components.

Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this ID to access the check.
(*Custom Model Advisor check only*) Handle to the check callback function (required)	Function that specifies the contents of a check.
(*Custom Model Advisor check only*) Handle to action callback function (optional)	Adds a fixing action.
(*Custom Edit-time check only*) Handle to class (required)	Derived class that defines the actions for the edit-time check. Optionally, this class can also define a fix for the edit-time check.
Check name (recommended)	Specifies a name for the check in the Model Advisor.
Model compiling (optional)	Specifies whether the model is compiled for check analysis. The `PostCompileForCodegen` value of the `CallbackContext` property is not supported for edit-time checks.
Input parameters (optional)	Adds input parameters that request input from the user. The Model Advisor uses the input to perform the check.

Define Custom Model Advisor Checks

For a custom check that only appears in the Model Advisor, the check definition function contains a check callback function that specifies the actions that you want the Model Advisor to perform on a

model or subsystem. Define the check callback function and pass a handle to it to the `setCallbackFcn` method. The Model Advisor executes the callback function when you run the check. Callback functions provide one or more return arguments that display the results after executing the check. The Model Advisor executes the callback function when you run the check.

If you are specifying a custom check fix, the check definition function should also contain an action callback function. In the check definition function, create an instance of the `ModelAdvisor.Action` class. Define the action callback function and pass a handle to it to the `setCallbackFcn` method. In the Model Advisor, the check user clicks **Fix** to apply the custom fix to their model.

Callback and action callback functions provide one or more return arguments for displaying the results after executing the check. See "Create the Check Callback Definition Function" on page 6-23 and "Create the Action Callback Definition Function" on page 6-24.

To use default formatting for Model Advisor results, specify the callback function type as `DetailStyle` in the `setCallbackFcn` method. If the default formatting does not meet your needs, use either the `ModelAdvisor.FormatTemplate` class or these other Model Advisor formatting classes:

Class	Description
`ModelAdvisor.Text`	Create a Model Advisor text output.
`ModelAdvisor.List`	Create a list.
`ModelAdvisor.Table`	Create a table.
`ModelAdvisor.Paragraph`	Create and format a paragraph.
`ModelAdvisor.LineBreak`	Insert a line break.
`ModelAdvisor.Image`	Include an image in the Model Advisor output.

Define Custom Edit-Time Checks

To create a custom edit-time check, create a MATLAB class that derives from the `ModelAdvisor.EdittimeCheck` class. In the check definition function, specify this class as the value of the `ModelAdvisor.Check CallbackHandle` property. Inside the derived class, define these methods:

- Define a method that specifies the check ID and the `ModelAdivsor.EdittimeCheck.TraversalType` properties of the check. The `TraversalType` property specifies how the Model Advisor runs the check.

- Define a `blockDiscovered` method that looks for blocks that violate your edit-time algorithm.

- If the violation is on a block, highlight the block during edit-time by creating a `ModelAdvisor.ResultDetail` violation object with the `Type` property set to the default value of `SID`. If the violation is on a signal, highlight the signal by creating a violation object with the `Type` property set to `Signal`.

- If you specify a `TraversalType` property of `edittimecheck.TraversalTypes.ACTIVEGRAPH`, define a `finishedTraversal` method that specifies what the edit-time check does with the data the check collects as part of the `blockDiscovered` method.

- Optionally, define a `fix` method for edit-time check violations.

For an example, see "Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor" on page 6-9.

To help prevent custom edit-time checks from negatively impacting performance as you edit your model, the Model Advisor automatically disables custom edit-time checks if, in the current MATLAB session, the check takes longer than 500 milliseconds to execute in at last three different Simulink models.

If the Model Advisor disables a custom edit-time check, you will see a warning on the Simulink canvas. You can re-enable the edit-time check by either:

- Clicking the hyperlink text in the warning.
- Passing the check identifier, `checkID`, to the function `edittime.enableCheck:edittime.enableCheck(checkID)`.

To prevent a custom edit-time check from being disabled, author the check so that the check executes in less than 500 milliseconds on your models.

Define Check Input Parameters

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

Specify the layout of input parameters in the Model Advisor by using these methods.

Purpose	Method
Specifies the size of the input parameter grid	`.setInputParametersLayoutGrid`
Specifies the number of rows the parameter occupies in the input parameter layout grid.	`setRowSpan`
Specifies the number of columns the parameter occupies in the input parameter layout grid.	`setColSpan`

The Model Advisor displays input parameters in the **Input Parameters** box.

Display and Enable Check

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

- If the `Visible` property is `false`, the check or task is not displayed in the Model Advisor and the `Enable` and `Value` properties are ignored.
- If the `Visible` property is `true` and the `Enable` property is `false`:

  - The check is displayed in the Model Advisor.
  - The initial status of the check is `Value`.
  - The check box appears dimmed.

- If the `Visible` property is `true` and the `Enabled` property is `true`, the check or task is displayed in the Model Advisor and the check box is active.

Publish Custom Check

Create a folder for custom checks in the **By Product** folder by using the `publish` method. Then, use the Model Advisor Configuration Editor to customize the folders within the Model Advisor tree. For more information, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

See Also

ModelAdvisor.Check | ModelAdvisor.EdittimeCheck | ModelAdvisor.InputParameter | ModelAdvisor.Action | `publish`

Related Examples

- "Create and Deploy a Model Advisor Custom Configuration" on page 7-23
- "Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor" on page 6-9
- "Define Custom Edit-Time Checks that Fix Issues in Architecture Models" on page 6-17
- "Fix a Model to Comply with Conditions that You Specify with the Model Advisor" on page 6-21
- "Create Model Advisor Check for Model Configuration Parameters" on page 6-27
- "Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters" on page 6-38

Define the Compile Option for Custom Model Advisor Checks

Depending on the implementation of your model and what you want your custom check to achieve, it is important that you specify the correct compile option. You specify the compile option for the check definition function of a `ModelAdvisor.Check` object by setting the `CallbackContext` property as follows:

- `None` specifies that the Model Advisor does not have to compile your model before analysis by your custom check. `None` is the default setting of the `CallbackContext` property.

- `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. The Model Advisor does not flag modeling issues that fail during code generation because these issues do not affect the simulated model.

- `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 the code generation readiness of the model. This option is not supported for custom edit-time checks.

Checks that Evaluate the Code Generation Readiness of the Model

You can create custom Model Advisor checks that enable the Model Advisor engine to identify code generation setup issues in a model at an earlier stage so you can avoid unexpected errors during code generation. For example, in this model, the `Red` enumeration in `BasicColors` and `OtherColors` are OK for use in a simulated model. In the generated code, however, these `Red` enumerations result in an enumeration clash. By using the `'PostCompileForCodegen'` option, your custom Model Advisor check can identify this type of code generation setup issue.

The `'PostCompileForCodegen'` option compiles the model for all variant choices. This compilation enables you to analyze possible issues present in the generated code for active and inactive variant paths in the model. An example is provided in "Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model" on page 6-51.

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` instead of `PostCompile` as the value for the `ModelAdvisor.Check.CallbackContext` property when generating code from the model is the final objective.

Update Model to Analyze All Variant Choices

For the Model Advisor to evaluate active and inactive paths in a variant system, you must set the **Variant activation time** parameter to `Code compile` or `startup` for the variant blocks (Variant Sink, Variant Source, and Variant Subsystem, Variant Model). You must also set the **System target file** configuration parameter to `ert.tlc`.

Note: Selecting this option can affect the execution time and increase 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 set the **Variant activation time** parameter to `Code compile`.
3 Save the model to your local working folder.
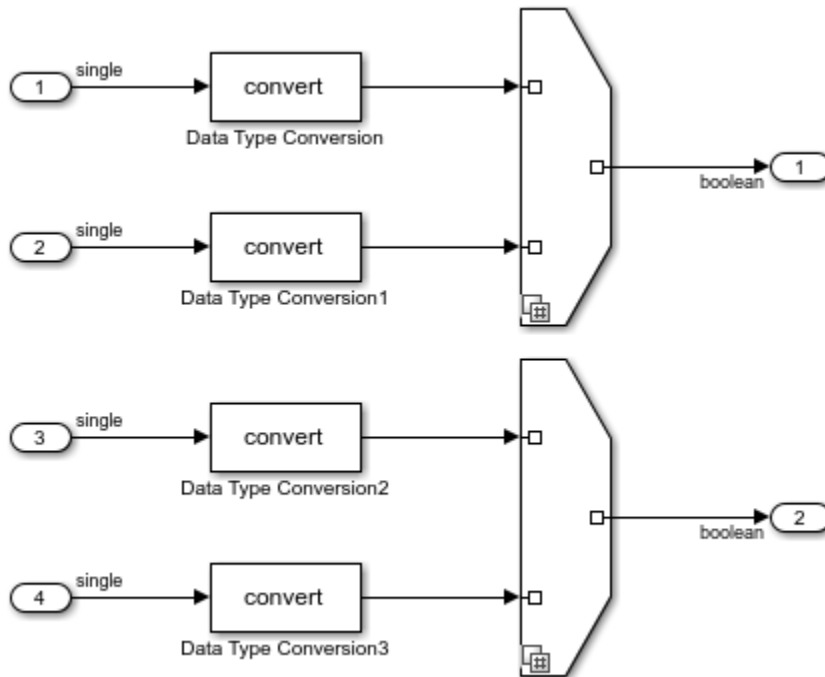
Create an `sl_customization` Function

In your working folder, create this `sl_customization` function and save it.

```
function sl_customization(cm)

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

% ------------------------------
% defines Model Advisor Checks
% ------------------------------
function defineModelAdvisorCheck
CheckSingleToBoolConversion;
```

The `sl_customization` function accepts a customization manager object. The customization manager object includes the `addModelAdvisorCheckFcn` method for registering custom checks. The input to this method is a handle to a function (`defineModelAdvisorCheck`). This function contains a call to the check definition function that corresponds to the custom check.

Open and inspect the check definition function,`CheckSingleToBoolConversion.m`:

```
function CheckSingleToBoolConversion
mdladvRoot = ModelAdvisor.Root;

rec = ModelAdvisor.Check('exampleCheck1');
```

```matlab
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(@DetailStyleCallback,'None','DetailStyle');
rec.CallbackContext = 'PostCompileForCodegen'; % Compile for Code Generation

mdladvRoot.publish(rec, 'Demo');

end

function DetailStyleCallback(system, CheckObj)

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);

violationBlks = find_system(system, 'BlockType', 'DataTypeConversion');
for ii = numel(violationBlks):-1:1
    dtcBlk = violationBlks{ii};
    compDataTypes = get_param(dtcBlk, 'CompiledPortDataTypes');
    if isempty(compDataTypes)
        violationBlks(ii) = [];
        continue;
    end
    if ~(strcmp(compDataTypes.Inport, 'single') && strcmp(compDataTypes.Outport, 'boolean'))
        violationBlks(ii) = [];
        continue;
    end
end

if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults(1,numel(violationBlks))=ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'This check looks for data type conversion blocks that convert s
    ElementResults.Status = 'Check has passed. No data type conversion blocks that convert single
    mdladvObj.setCheckResultStatus(true);
else

    for i=1:numel(violationBlks)
    ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)

        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'This check looks for data type conversion blocks that co
        ElementResults(i).Status = 'Check has failed. The following data type conversion blocks
        ElementResults(i).RecAction =  'Modify the model to avoid converting data type from sing
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end
```

For more information on creating custom checks, see "Define Custom Model Advisor Checks" on page 6-45.

Open Model Advisor and Execute Custom Check

Before opening the Model Advisor and running the custom check, you must refresh the Model Advisor check information cache. In the MATLAB Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

To open the Model Advisor and execute the custom check:

1   Open your saved model.
2   In the **Modeling** tab, select **Model Advisor**. A **System Selector - Model Advisor** dialog box opens. Click **OK**. The Model Advisor opens.
3   In the left pane, select **By Product > Demo > Check to identify Single to Bool conversion**.
4   Right-click the check and select **Run this Check**. The Model Advisor compiles the model and executes the check. The Model Advisor updates the model diagram. The inactive variant paths appear dimmed.
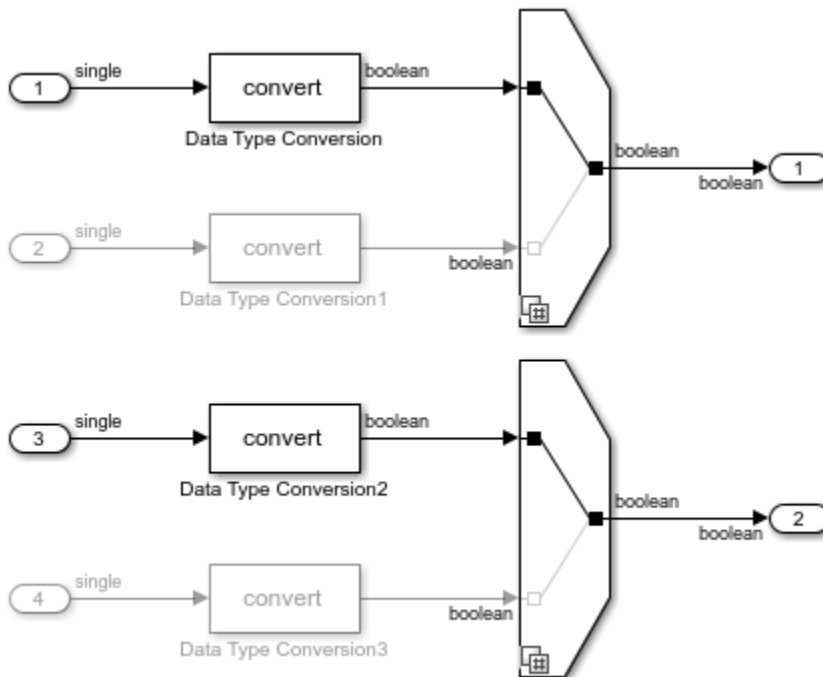
Review the Model Advisor Results

Review the check analysis results in the Model Advisor. Click the hyperlinks 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 systems.
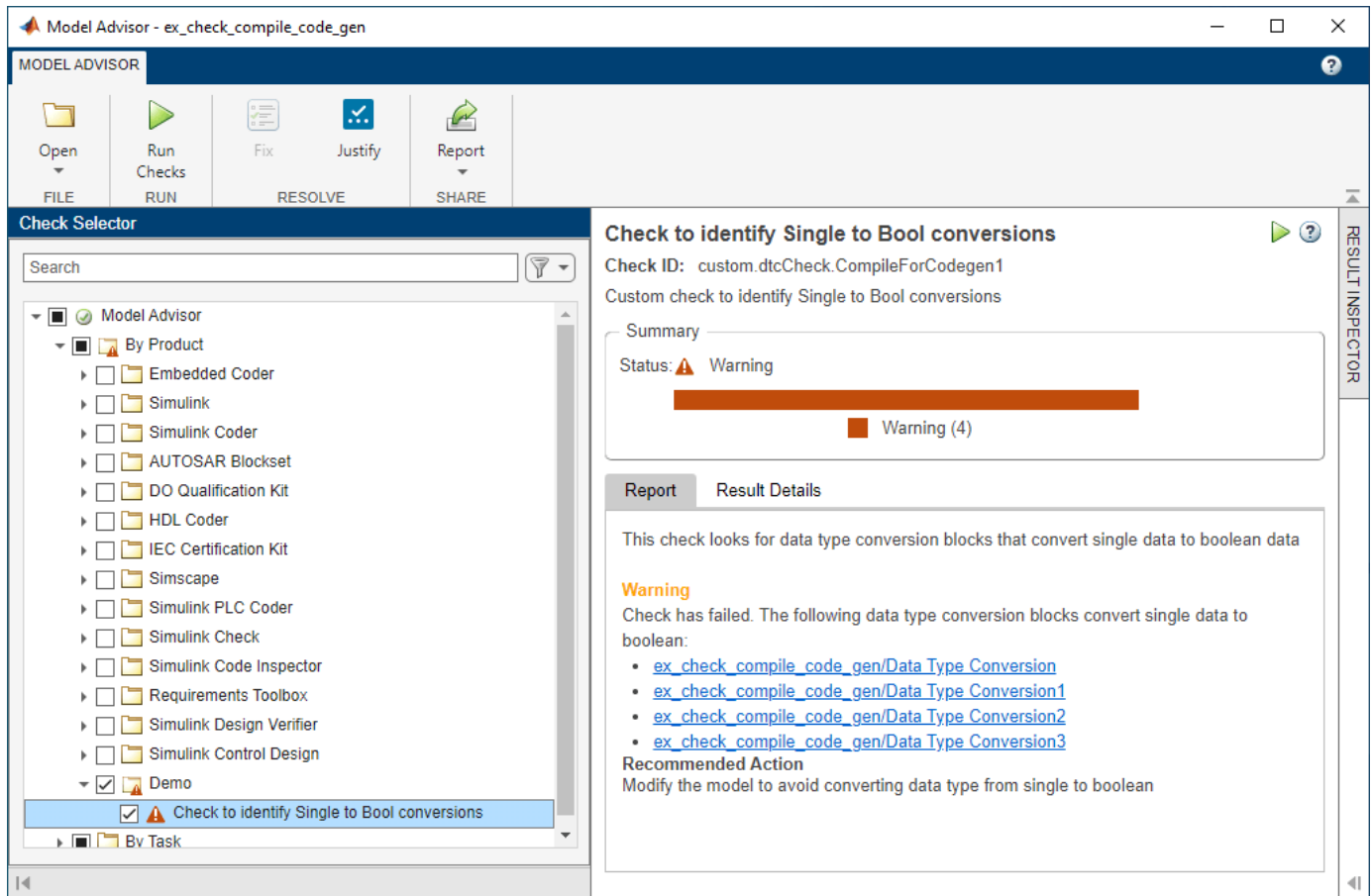
If you defined the compile option in the `sl_customization.m` file as

```
rec.CallbackContext = 'PostCompile';
```

the results include only the Data Type Conversion blocks in the active path.
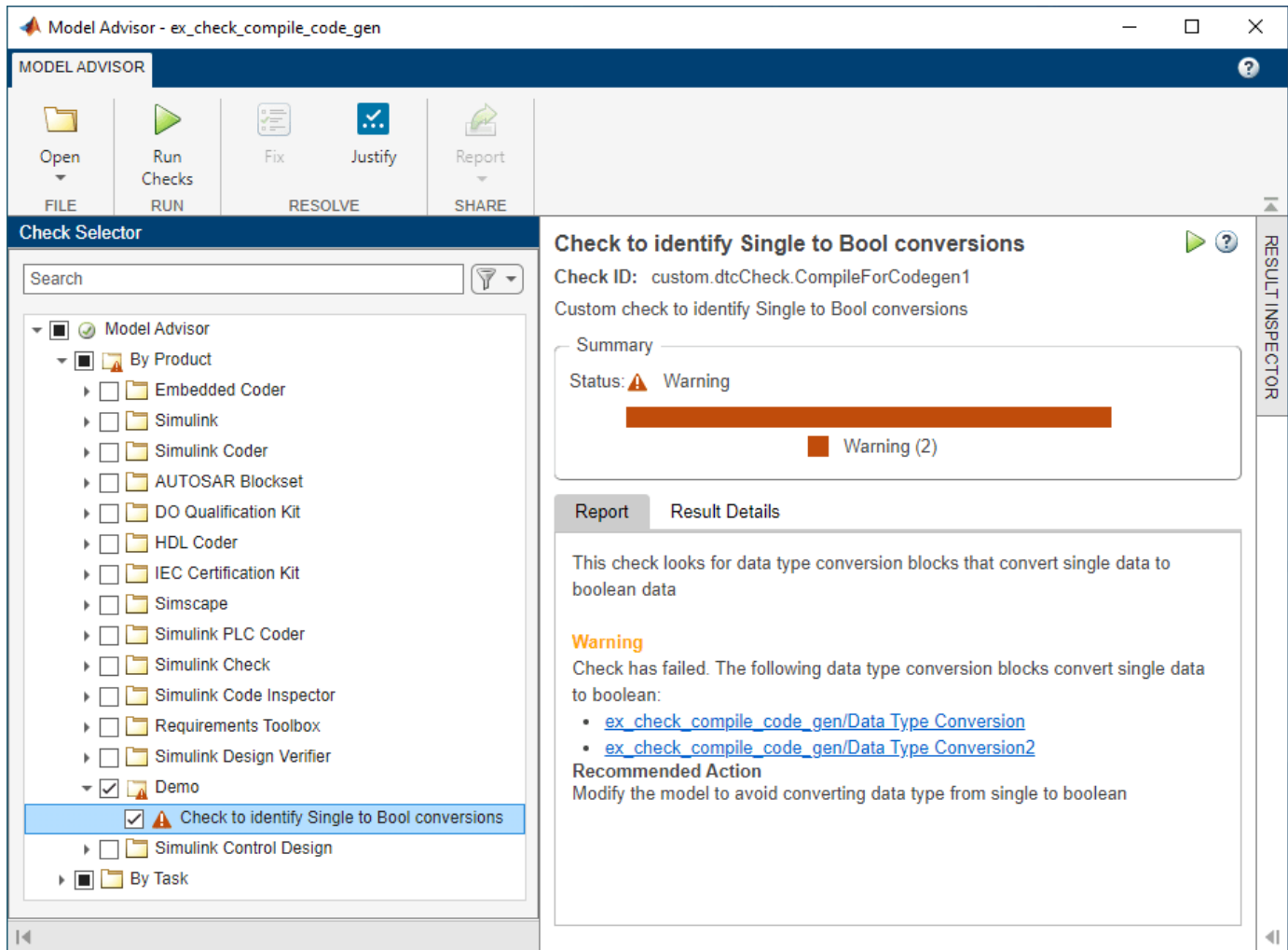
See Also
`ModelAdvisor.Check` | `ModelAdvisor.Check.CallbackContext`

More About
- "Define Custom Model Advisor Checks" on page 6-45
- "Variant Systems"

Exclude Blocks From Custom Checks

This example shows how to exclude blocks from custom checks. To save time during model development and verification, you can exclude individual blocks from custom checks during a Model Advisor analysis. To exclude custom checks from Simulink blocks and Stateflow charts, use the `ModelAdvisor.Check.supportExclusion` and `Simulink.ModelAdvisor.filterResultWithExclusion` functions in the check definition file.

Update the Check Definition File

1 Set your folder to a writeable directory.

2 Copy the script `prepare_cust_chk_code` to your current folder and run the script. The script copies the files for this example to your current folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main','prepare_cust_chk_code.m'),...
    'prepare_cust_chk_code.m','f');
run('prepare_cust_chk_code.m');
```

3 Open the `defineDetailStyleCheck` file.

4 To update the **Check whether block names appear below blocks** check to exclude blocks during Model Advisor analysis, make two modifications to the `defineDetailStyleCheck` file.

   a Enable the **Check whether block names appear below blocks** check to support check exclusions by using the `ModelAdvisor.Check.supportExclusion` property. After `rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');`, add `rec.supportExclusion = true;`. The first section of the `function defineDetailStyleCheck` now looks like:

   ```
   % Create ModelAdvisor.Check object and set properties.
   rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
   rec.Title = 'Check whether block names appear below blocks';
   rec.TitleTips = 'Check position of block names';
   rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
   rec.supportExclusion = true;
   ```

   b Use the `Simulink.ModelAdvisor.filterResultWithExclusion` function to filter model objects causing a check warning or failure with checks that have exclusions enabled. To do this, modify the `DetailStyleCallback(system, CheckObj)` function as follows:

   ```
   % Find all blocks whose name does not appear below blocks
   violationBlks = find_system(system, 'Type','block',...
       'NamePlacement','alternate',...
       'ShowName', 'on');
   violationBlks = mdladvObj.filterResultWithExclusion(violationBlks);
   ```

5 Save the `DefineDetailStyleCheck` file. If you are asked if it is OK to overwrite the file, click **OK**.

Create and Save Exclusions

1 In order for your customizations to be visible in the Model Advisor, you must refresh the Model Advisor check information cache. At the MATLAB command prompt, type this command:

```
Advisor.Manager.refresh_customizations();
```

2 To open the model, double-click `AdvisorCustomizationExample.slx`.

3 In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.

4 In the left pane of the Model Advisor window, select the **By Product > Demo > Check whether block names appear below blocks** check. In the right pane, select **Run This Check**. The check fails.

5 In the model window, right-click the X block and select **Model Advisor > Exclude block only > Select Checks**. Navigate to the **Demo** folder and select the **Check whether block names appear below blocks** check.

6 In the Model Advisor Exclusion Editor, click **Save** to create an exclusion file.

7 In the model window, open the `Amplifier` subsystem and right-click the `GainBlock` block and select **Model Advisor > Exclude block only > Select Checks**. Navigate to the **Demo** folder and select the **Check whether block names appear below blocks** check.

8 In the Model Advisor Exclusion Editor, click **Save** to update the exclusion file.

Review Exclusions

1 In the left pane of the Model Advisor window, select the **By Product > Demo > Check whether block names appear below blocks** check. In the right pane, select **Run Checks**. The check now passes. In the right-pane of the Model Advisor window, you can see the **Check Exclusion Rules** that the Model Advisor applies during the analysis.

2 Close the model and the Model Advisor.

See Also

supportExclusion | `Simulink.ModelAdvisor`

Related Examples

• Excluding Blocks From Model Advisor Checks on page 3-9

More About

• "Run Model Advisor Checks and Review Results" on page 3-4

Create Help for Custom Model Advisor Checks

You can define help files for your custom Model Advisor checks to make the checks easier to use. Custom help files allow you to verify the check capabilities and avoid potential warnings in the model.

You can define custom help for:

- Custom checks - To Add help for custom Model Advisor checks, use `ModelAdvisor.Check.setHelp`. For more information, see `setHelp`.
- Folders that have custom checks - To add help for folders that contain custom Model Advisor checks, use `ModelAdvisor.Group.setHelp`. For more information, see `setHelp`.

To point the custom check help to a PDF or an HTML page of your choice:

1 Open the `sl_customization.m` file.

2 Use `setHelp()` on the check or group object created in the `sl_customization.m` file.

```
setHelp('format','webpage','path','custom_path');
```

The supported name-value arguments are:

Format - `"webpage"` , `"pdf"`

Path - Path of the user-defined help page or document

Example:

```
checkObj = ModelAdvisor.Check('SimplePassFailCheck');
checkObj.setHelp('format','webpage','path','custom_path');
```

3 Close the `sl_customization.m` file.

4 Refresh the customizations by entering:

```
Advisor.Manager.refresh_customizations
```

To view the custom help, right-click the custom checks or the folder and click **What's This?**.

See Also

"Define Custom Model Advisor Checks" on page 6-45

"Create and Deploy a Model Advisor Custom Configuration" on page 7-23

Model Advisor Customization

Customize the Configuration of the Model Advisor Overview

You can use Model Advisor API and the Model Advisor Configuration Editor to customize the configuration of the Model Advisor, including:

- Define which built-in (shipped) and custom Model Advisor checks are available in the Model Advisor and their order of execution.
- Create custom folders and organize checks.
- Designate the default configuration file for the Model Advisor.
- Associate a configuration file with a model.
- Suppress the warning about missing checks when loading the Model Advisor configuration.

To customize the Model Advisor to include custom checks and a custom configuration, perform the following tasks:

1 (Optional) Author custom checks in a customization file. For more information, see "Create Model Advisor Checks".

2 Use the Model Advisor Configuration Editor to specify the folders and checks that you want to include in your custom Model Advisor configuration. For more information, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

3 Update your Simulink environment so that the Model Advisor uses your configuration files. For more information, see "Update the Environment to Include Your Custom Configuration" on page 7-19.

4 Open the Model Advisor, load the configuration, and verify that the Model Advisor is using the correct configuration and that your checks are available. Optionally, you can associate the configuration with your model. For more information, see "Load and Associate a Custom Configuration with a Model" on page 7-20.

5 (Optional) Deploy the custom configurations to your users. For more information, see "Deploy Custom Configurations" on page 7-22.

6 Verify that models comply with modeling guidelines. For more information, see "Run Model Advisor Checks and Review Results" on page 3-4.

Use the Model Advisor Configuration Editor to Customize the Model Advisor

Overview of the Model Advisor Configuration Editor

The Model Advisor Configuration Editor provides a way for you to specify the checks that you want to use for edit-time checking, as well as the checks included in the Model Advisor. This organizational hierarchy is saved as a configuration file, which is loaded when you initiate the Model Advisor. You can use the Model Advisor Configuration Editor to modify existing configurations, create new Model Advisor configurations, and specify the default configuration.

The Model Advisor Configuration Editor gives you the flexibility to customize the Model Advisor analysis to meet the needs of your organization by allowing you to:

- Review all available Model Advisor checks.
- Add, remove, and organize built-in checks and folders in the Model Advisor tree.
- Integrate custom Model Advisor checks in your verification and validation workflow.
- Disable and enable checks and folders.
- Rename checks and folders.
- Specify whether a check is marked as a warning or failure when it is flagged during a Model Advisor analysis.
- Suppress the warning about missing checks when loading the Model Advisor configuration.

The Model Advisor Configuration Editor includes:

- The **Library** pane — A read-only pane that lists all checks and folders that are available for use in the configuration, delineated by the **By Product** and **By Task** tabs. To permanently display the Library tab, click **Show Library** on the toolstrip.
- The **Model Advisor** pane — This pane lists the checks and folders in the current Model Advisor configuration, filtered by:

  - `All checks` — Lists all Model Advisor checks included in the current configuration
  - `Edit time supported checks` — Lists only the Model Advisor checks that are supported as edit-time checks
- **Information** tab — This tab provides details about the check or folder, such as the **Display Name**, **Check Instance ID** or **Check Group ID**, and the **Check result when issues are flagged**.

Use the search functionality in the **Library** and **Model Advisor** panes to locate specific checks and folders.
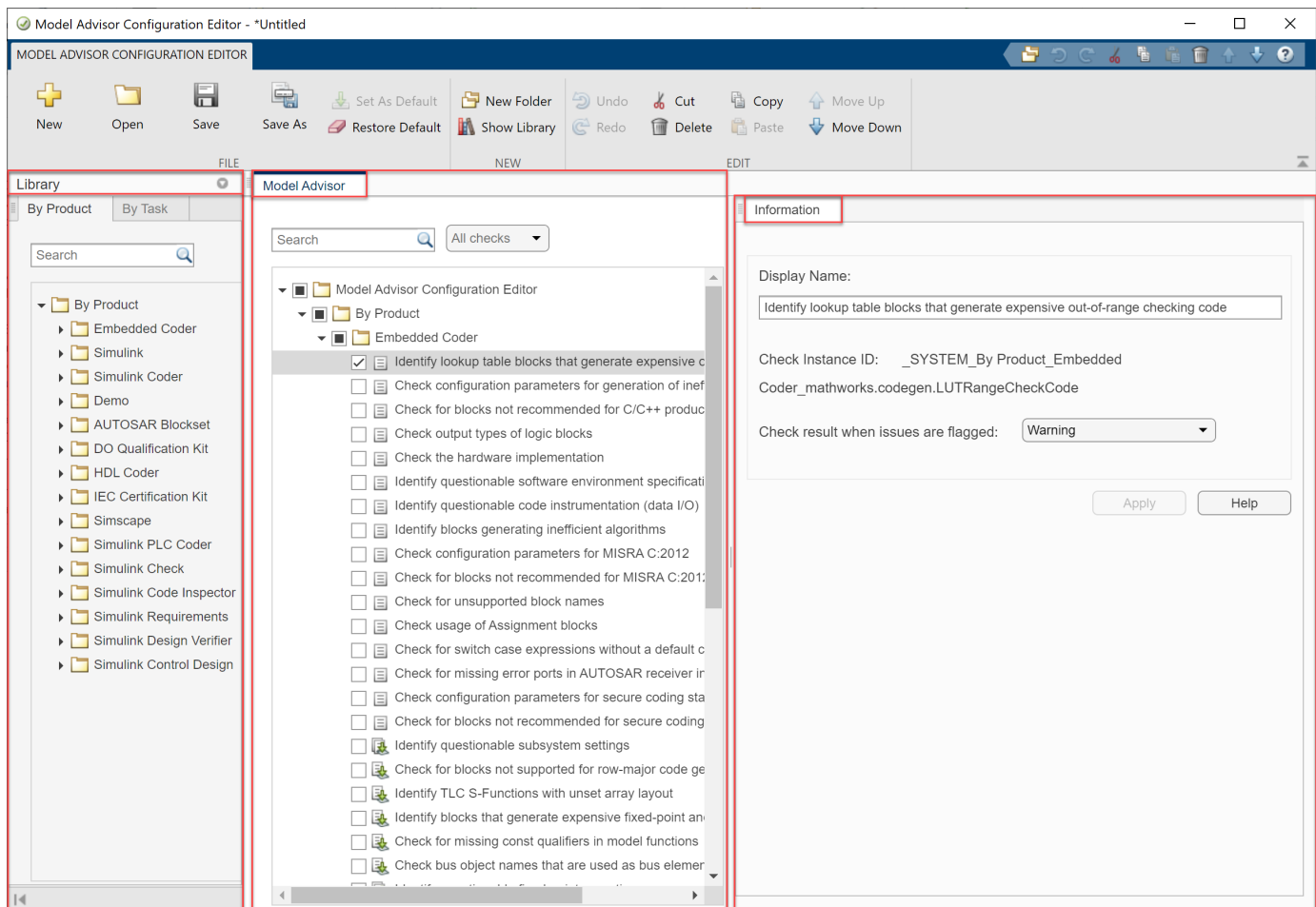
Open the Model Advisor Configuration Editor

Before opening the Model Advisor Configuration Editor, verify that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.

When implementing custom checks or Model Advisor customizations by using the Model Advisor API, you must first update the Simulink environment to include your `sl_customization.m` file. At the MATLAB command line, enter

```
Advisor.Manager.refresh_customizations
```

Use one of these methods to open the Model Advisor Configuration Editor:

- Programmatically — At the MATLAB command line, enter `Simulink.ModelAdvisor.openConfigUI`.
- From the Simulink editor — In the **Modeling** tab, select **Model Advisor > Configuration Editor** .
- From the Model Advisor — Select **Open > Open Configuration Editor**.

The configuration file that is currently being used by the Model Advisor displays when you open the Model Advisor Configuration Editor. The file name for the configuration is displayed at the top of the

window. Verify that you are evaluating the correct configuration file. To open a different configuration file, click **Open** and browse to the file you would like to review.

To create a new configuration, click the **New** button on the toolstrip. Use **Save As** to rename the configuration file. Model Advisor configuration files are saved in `.json` format.

Specify a Default Configuration File

You can use the Model Advisor Configuration Editor to specify a default configuration that loads automatically when the Model Advisor opens. To set the default configuration, open the configuration file in the Model Advisor Configuration Editor and click the **Set As Default** button on the toolstrip.

Note If you have previously designated a default configuration, you can use **Clear default configuration setting** to clear the setting that designates the current default configuration file. Clicking the button does not modify the configuration that is currently displayed in the Model Advisor Configuration Editor. When you do not specify a default configuration, the Model Advisor uses the standard configuration that is defined by your system administrator.

If you do not specify the configuration file as the default, when you save the file, you are prompted as to whether to make the file the default configuration. To make this file the default configuration, click **Yes**.

To associate a custom configuration with a model, so that the Model Advisor uses that configuration each time you open that model, see "Load and Associate a Custom Configuration with a Model" on page 7-20.

Customize the Model Advisor Configuration

You can use the Model Advisor Configuration Editor to customize the Model Advisor configuration tree, including adding and removing checks and folders and specifying the order in which checks are executed. You can also disable the ability for users to select whether to include or exclude a check from an analysis. You can also use the Model Advisor Configuration Editor to define the input parameters for a check.

Note Checks that are copied from the **Library** tab retain their default parameter settings. When they are pasted into your custom configuration folder, the box beside each check is not selected.

Checks that are copied or cut from a folder in the **Model Advisor** tab retain their user-define parameter settings. When a check is included in multiple folders, you can specify different parameters for each check individually.

Organize the Hierarchy

You can customize the layout of the checks and folders in the Model Advisor configuration tree by using:

• **New Folder** to create a folder.
• **Copy**, **Cut**, and **Paste** to add, copy, and move checks and folders.

- **Delete** to remove checks and folders.
- **Move Up** or **Move Down** to shift the position of the check or folder in the configuration tree. The folders and checks that are higher in the configuration tree are executed first in the analysis.

Note, you can customize the Model Advisor by using the `ModelAdvisor.Group` and `ModelAdvisor.FactoryGroup` classes instead of the Model Advisor Configuration Editor. However, these APIs are a less flexible and more time-consuming way of customizing the Model Advisor. To place customized checks in custom folders at the top-level of the Model Advisor tree (the Model Advisor root), use the `ModelAdvisor.Group` class. To place customized checks in new folders in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. You must include methods that register these tasks and folders in the `sl_customization` function.

Enable or Disable Checks

You can use the Model Advisor Configuration Editor to disable the check box control for checks and folders in the Model Advisor. By doing so, the check is still listed in the Model Advisor configuration tree, but it is dimmed and you do not have the ability to add or remove the check from the analysis.

In the **Model Advisor** pane, right-click on a folder or check and select **Disable**. Depending on the check box selection in the Model Advisor Configuration Editor, the following results occur in the Model Advisor:

- If the box beside check is selected in the Model Advisor Configuration Editor, then in the Model Advisor, the check is automatically selected. Because you selected **Disable**, the check is dimmed and you cannot choose to remove the check from the analysis.

  If the box beside the check is *not* selected and the **Disable** option is applied in the Model Advisor Configuration Editor, then in the Model Advisor, the check is not selected and you cannot included it in the analysis.

- If the box beside folder is selected in the Model Advisor Configuration Editor, then in the Model Advisor, the checks within the folder are automatically selected. Because you selected **Disable**, the folder and its checks are dimmed and you cannot choose to remove the checks from the analysis.

  If the box beside the folder is *not* selected and the **Disable** option is applied in the Model Advisor Configuration Editor, then in the Model Advisor, none of the checks within the folder are selected and you cannot include it in the analysis.

When a check or folder is disabled, you can use the **Enable** option to allow users to determine whether to include the check(s) in an analysis.

Note **Enable** and **Disable** affects the execution of checks in the analysis for both the Model Advisor user interface and edit-time checking.

Specify Parameters for Check Customization

You can use the Model Advisor Configuration Editor to customize a Model Advisor check, such as the display name and input parameters for the check.

In the **Information** tab, review the content that you can customize for the check:

- **Display Name** — Provide a new name for the check, which is displayed in the Model Advisor. Note that changing the display name does not change the check title.

- **Check result when issues are flagged** — Specify whether you want the check to be marked as a warning or failure in the results when the check flags an issue in your model. The default value is `Warning`. Select `Fail` to mark a flagged check as failed in the results.
- **Input Parameters** — Specify additional characteristics and functionality for the check. The Model Advisor uses these parameters to further define the emphasis of the analysis. For example, you can choose to include only subcheck **jc_0736_b** and specify the acceptable number of single-byte spaces in the analysis for Model Advisor check "Check indentation of code in Stateflow states".

Suppress Warning Message for Missing Checks

The Model Advisor automatically warns you of checks that are missing when loading a Model Advisor configuration. You can use the Model Advisor Configuration Editor to suppress this message. Select the **Model Advisor Configuration Editor** root node and, in the **Information** tab, select **Suppress warning message for missing checks when loading configuration**.

Alternatively, you can programmatically suppress the Model Advisor warning by entering this command at the MATLAB command line:

```
warning('off','Simulink:tools:MALoadConfigMissCorrespondCheck')
```

Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration

You can use the Model Advisor Configuration Editor to organize the hierarchy of the Model Advisor and specify checks that are included in check analyses. This example shows how to create a new configuration file, specify checks for the Model Advisor and edit-time checking, define check parameters, and load the configuration to the Model Advisor.

Create a Model Advisor Configuration

In this example, you will create a custom configuration file named custom_`Configuration.json`. This configuration will consist of MathWorks Advisory Board (MAB) modeling guidelines checks and industry standard checks that you want to execute by using the Model Advisor.

1. Open the Model Advisor Configuration editor by entering this command at the command prompt:

```
Simulink.ModelAdvisor.openConfigUI
```

2. In the toolstrip, select **Show Library** to display the **Library** pane. In the **By Product** tab search field, enter **ISO 26262**.

3. Right-click on the **Simulink Check > Modeling Standards > IEC 61508, IEC 62304, ISO 26262, ISO 225119, EN 50128, and EN 50657 Checks** folder and select **Copy**. Right-click on the Model Advisor Configuration Editor root folder and click **Paste**. Verify that the folder and checks have been copied to the root folder.

Note: Checks that are copied from the **Library** pane retain their default parameter settings. When they are pasted into your custom configuration folder, the box beside each check is not selected.

4. In the **By Task** folder on the **Model Advisor** pane, right-click on the **Modeling Standards for MAB** folder and select **Cut**. Click on the **Model Advisor Configuration Editor** root folder and click **Paste**. The folder is removed from the **By Task** folder and is added as a new subfolder in the **Model Advisor Configuration Editor** root folder.

Note: Checks that are copied or cut from a folder in the **Model Advisor** pane retain their user-defined parameter settings. When a check is included in multiple folders, you can specify different parameters for each check individually.

5. Select the **IEC 61508, IEC 62304, ISO 26262, ISO 225119, EN 50128, and EN 50657 Checks** folder and use the **Move Down** button to change the position this folder in the hierarchy. The Model Advisor will execute the checks in the **Modeling Standards for MAB** folder first.

6. Select the **By Product** and **By Task** folders and select **Delete**.

7. In the **Model Advisor** pane, set the configuration focus option to `Edit-Time supported checks`. The `Edit-Time supported checks` option displays the checks in this configuration that support edit-time checking.

8. Click **Save As** and name the configuration file to custom_`Configuration`. Select **Yes** at the prompt to save the configuration as the default configuration. The file automatically saves in `.json` format.

Note: If you have previously designated a default configuration, you can use **Clear default configuration** setting to clear the flag that specifies the current default configuration file. Clicking the button does not modify the configuration that is currently displayed in the Model Advisor Configuration Editor.

9. Close the Model Advisor Configuration Editor.

10. Open the `AdvisorCustomizationExample.slx` model by entering the following in the MATLAB command line:

```
open_system('AdvisorCustomizationExample.slx');
```

11. On the **Modeling** tab, click **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, select **Edit-Time Checks** and **Apply**. Close the Configuration Parameters dialog box.

12. In the model, notice that three blocks are highlighted. These blocks contain edit-time check violations for this configuration. Place your cursor over a warning and click the block to discover the issue.
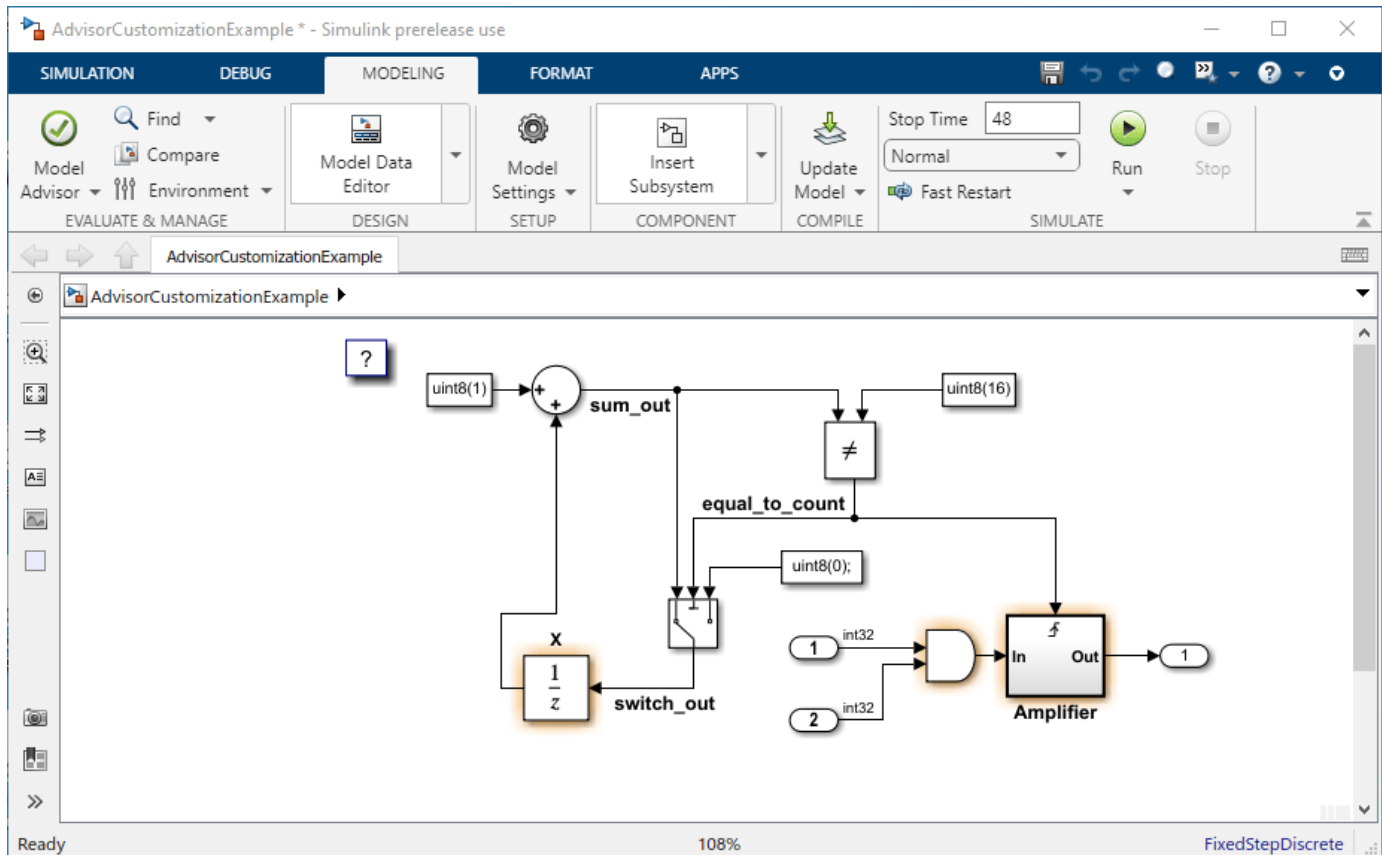
13. Open the Model Advisor and confirm that the Model Advisor displays the folders **Modeling Standards for MAB** and **IEC 61508, IEC 62304, ISO 26262, ISO 225119, EN 50128, and EN 50657 Checks**.

14. Close the Model Advisor.

Update a Model Advisor Configuration

You will now customize the checks in your custom configuration file, custom_`Configuration.json` file and review the effect that your customizations have on the Model Advisor analysis of the `AdvisorCustomizationExample.slx` model.

1. On the **Modeling** tab, click **Model Advisor > Configuration Editor**.

2. Clear the check box the box beside the **Model Advisor Configuration Editor** root node folder. (This step is optional. However, deselecting the checks allows you to more easily view the results of using the Model Advisor Configuration Editor to specify checks for display in the Model Advisor.)

3. To allow you to use the Model Advisor to specify which checks to include in the Model Advisor analysis, right-click on the **Model Advisor Configuration Editor** root folder and click **Enable**. (Note: Enable is the default setting. This option is dimmed when none of the checks are disabled.)

4. Check the box beside the **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block name** check.

5. Right-click on each of these checks and select **Disable**:

- **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names**
- **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem name**

6. Click the **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check and, in the **Information** tab, select `Fail` for the **Check result when issues are flagged** option. Click **Apply**.

Note: The default for the **Check result when issues are flagged** option is `Warning`.

7. Click **Save** to save the configuration. Close the Model Advisor Configuration Editor and the model.

8. Refresh the Model Advisor cache and open model `AdvisorCustomizationExample.slx` by entering the following in the MATLAB command line:

```
Advisor.Manager.refresh_customizations();
open_system('AdvisorCustomizationExample.slx');
```

9. Open the Model Advisor.

Observe these checks, which reflect the settings that you chose in the Model Advisor Configuration Editor:

- The **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names check** is dimmed and the check box is selected. This check will always execute in a Model Advisor analysis and, because it is dimmed, you cannot choose to exclude it from the analysis.
- The **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem names check** is dimmed and the check box is not selected. This check will not be included in the analysis and, because it is dimmed, you cannot select it for inclusion in the analysis.

10. Check the box beside the **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check.

11. To run the Model Advisor analysis, right-click on the **Model Advisor Standards for MAB** root node and select **Run Selected Checks**.

12. Click on the following checks and review the Model Advisor analysis results:

- The **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names** check is marked with a warning icon and the results specify the check violation is in the Gain block.
- There are no results for the **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem** check because it could not be selected for the analysis.
- The **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check is marked with a fail icon. This behavior is intended; you specified this check settings by using the Model Advisor Configuration Editor.

See Also
ModelAdvisor.setDefaultConfiguration | ModelAdvisor.Check

More About

- "Define Custom Model Advisor Checks" on page 6-45
- "Customize the Configuration of the Model Advisor Overview" on page 7-2
- "Update the Environment to Include Your Custom Configuration" on page 7-19
- "Load and Associate a Custom Configuration with a Model" on page 7-20

Programmatically Customize Tasks and Folders for the Model Advisor

Customization File Overview

The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

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

Function	Description	Required or Optional
`sl_customization()`	Registers custom checks and tasks, folders with the Simulink customization manager at startup. See "Define Custom Model Advisor Checks" on page 6-45.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See "Define Custom Model Advisor Checks" on page 6-45.	Required for custom checks and to add custom checks to the **By Product** folder.
One or more task definitions	Defines custom tasks. See "Define Custom Tasks" on page 7-13.	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 7-13.	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.

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.Group.addTask` method and the task ID.

Create Custom Tasks Using MathWorks Checks

You can add MathWorks checks to your custom folders by defining the checks as custom tasks. When you add the checks as custom tasks, you identify checks by the check ID.

To find MathWorks check IDs:

1 In the hierarcy, navigate to the folder that contains the MathWorks check.

2 In the left pane of the Model Advisor, select the check.

3 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.

4 Select and copy the **Check ID** of the check that you want to add from the **Command Window** as a task.

Display and Enable Tasks

The `Visible`, `Enable`, and `Value` properties interact the same way for tasks as they do for checks.

Define Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor using the following guidelines:

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Task Definition Function

The following example shows a task definition function. This function defines three tasks.

Define Custom Folders

About Custom Folders

Use folders to group checks in the Model Advisor by functionality or usage. You define custom folders in:

- A factory group definition function that specifies the properties of each instance of the `ModelAdvisor.FactoryGroup` class.
- A task definition function that specifies the properties of each instance of the `ModelAdvisor.Group` class.

Define one instance of the group classes for each folder that you want to add to the Model Advisor.

Add Custom Folders

To add a custom folder:

1   Create the folder using the `ModelAdvisor.Group` or `ModelAdvisor.FactoryGroup` classes.
2   Register the folder.

Define Where Custom Folders Appear

You can specify the location of custom folders within the Model Advisor using the following guidelines:

*   To define a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
*   To define a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Note To define a new folder in the **By Product** folder, use the `ModelAdvisor.Root.publish` method within a custom check. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

Group Definition

The following examples shows a group definition. The definition places the tasks inside a folder called **My Group** under the **Model Advisor** root. The task definition function includes this group definition.

The following example shows a factory group definition function. The definition places the checks into a folder called **Demo Factory Group** inside of the **By Task** folder.

See Also
ModelAdvisor.Check | ModelAdvisor.FactoryGroup | ModelAdvisor.Group | ModelAdvisor.Task | ModelAdvisor.Procedure | publish

More About
*   "Define Custom Model Advisor Checks" on page 6-45
*   "Customize the Configuration of the Model Advisor Overview" on page 7-2

Programmatically Create Procedural-Based Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

To create a procedural-based configuration, perform the following tasks:

1   Review the information in "Customize the Configuration of the Model Advisor Overview" on page 7-2.

2   Decide on order of changes to your model.

3   Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.

4   (Optional) Author custom checks in a customization file. See "Create Model Advisor Checks".

5   Organize the checks into procedures for a procedural-based configuration:

   a   Create procedures by using the procedure API. For detailed information, see "Create Procedural-Based Configurations" on page 7-16.

   b   Create the custom configuration "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3

       .

6   (Optional) Deploy the custom configurations to your users. For detailed information, see "Deploy Custom Configurations" on page 7-22.

7   Verify that models comply with modeling guidelines. For detailed information, see "Run Model Advisor Checks and Review Results" on page 3-4.

Create Procedural-Based Configurations

Create Procedures Using the Procedures API

You create procedures with the `ModelAdvisor.Procedure` class API. You first add the checks to tasks, which are wrappers for the checks. The tasks are added to procedures.

Note When creating procedural checks, be aware of potential conflicts with the checks. Verify that it is possible to pass both checks.

You use the `ModelAdvisor.Procedure` class to create procedural checks.

1   Add each check to a task using the `ModelAdvisor.Task.setCheck` method. The task is a wrapper for the check. You cannot add checks directly to procedures.

2   Add each task to a procedure using the `ModelAdvisor.Procedure.addTask` method.

Define Procedures

You define procedures in a procedure definition function that specifies the properties of each instance of the `ModelAdvisor.Procedure` class. Define one instance of the procedure class for each procedure that you want to add to the Model Advisor. Then register the procedure using the `ModelAdvisor.Root.register` method.

You can add subprocedures or tasks to a procedure. The tasks are wrappers for checks.

- Use the `ModelAdvisor.Procedure.addProcedure` method to add a subprocedure to a procedure.
- Use the `ModelAdvisor.Procedure.addTask` method to add a task to a procedure.

The following code example adds subprocedures to a procedure:

```
%Create a procedure
MAP = ModelAdvisor.Procedure('com.mathworks.example.Procedure');

%Create 3 sub procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub1');
MAP2=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub2');
MAP3=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub3');

%Add sub procedures to procedure
addProcedure(MAP, MAP1);
addProcedure(MAP, MAP2);
addProcedure(MAP, MAP3);

%register the procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAP);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

The following code example adds tasks to a procedure:

```
%Create three tasks
MAT1=ModelAdvisor.Task('com.mathworks.tasksample.myTask1');
MAT2=ModelAdvisor.Task('com.mathworks.tasksample.myTask2');
MAT3=ModelAdvisor.Task('com.mathworks.tasksample.myTask3');

%Create a procedure
MAP = ModelAdvisor.Procedure('com.mathworks.tasksample.myProcedure');

%Add the three tasks to the procedure
addTask(MAP, MAT1);
addTask(MAP, MAT2);
addTask(MAP, MAT3);

%register the procedure and tasks
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAP);
mdladvRoot.register(MAT1);
mdladvRoot.register(MAT2);
mdladvRoot.register(MAT3);
```

You can specify where the Model Advisor places a procedure using the `ModelAdvisor.Group.addProcedure` method. The following code example adds procedures to a group:

```
%Create three procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure3');

%Create a group
MAG = ModelAdvisor.Group('com.mathworks.sample.myGroup');

%Add the three procedures to the group
addProcedure(MAG, MAP1);
addProcedure(MAG, MAP2);
addProcedure(MAG, MAP3);

%register the group and procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAG);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

See Also
`ModelAdvisor.Check` | `ModelAdvisor.Procedure`

More About
* "Define Custom Model Advisor Checks" on page 6-45
* "Customize the Configuration of the Model Advisor Overview" on page 7-2

Update the Environment to Include Your Custom Configuration

To make custom configuration available for use by the Model Advisor, you need to first update your Simulink environment to refresh the Model Advisor cache. This includes the creation of new or modifications to existing:

- `.json` files by using the Model Advisor Configuration Editor. See "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.
- `sl_customization.m` files for custom Model Advisor checks. See "Create Model Advisor Checks".

To update your environment:

1 If you previously started the Model Advisor:

    a Close the model from which you started the Model Advisor

    b Clear the data associated with the previous Model Advisor session by removing the `slprj` folder from your code generation folder.

2 In the MATLAB command line, enter:

    `Advisor.Manager.refresh_customizations`

3 Open your model

4 In the **Modeling** tab, select **Model Advisor** to open the Model Advisor. If you have customized the configuration by using the Model Advisor Configuration Editor, load and verify the configuration as described in "Load and Associate a Custom Configuration with a Model" on page 7-20.

Load and Associate a Custom Configuration with a Model

Custom configurations allow you to specify which checks run during Model Advisor analysis. When you load a custom configuration, the Model Advisor uses the folders and checks specified by the configuration. You can also associate a custom configuration with your model so that the Model Advisor uses that configuration each time you open that model.

For example, to create a custom configuration and associate the configuration with a model newModel:

1 Set your current folder to a writeable directory.
2 Create and save a new model called newModel. In the MATLAB Command Window, enter:

```
new_system("newModel");
save_system("newModel");
```
3 Open the Model Advisor by entering:

```
modeladvisor("newModel")
```
4 In the Model Advisor, click **Open > Open Configuration Editor** to open the Model Advisor Configuration Editor.

You can use the Model Advisor Configuration Editor to specify the folders and checks that you want to include in the Model Advisor. For information on how to create a custom configuration, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.
5 In the Model Advisor Configuration Editor, click **Save As** to save a new, custom configuration file. For this example, save the configuration file as customConfig.json. The Model Advisor Configuration Editor prompts you to save the configuration as the default configuration. For this example, click **No**.
6 Update your Simulink environment to include the custom configuration file. At the MATLAB command line, enter:

```
Advisor.Manager.refresh_customizations
```

For more information, see "Update the Environment to Include Your Custom Configuration" on page 7-19.
7 Re-open the Model Advisor for the model newModel. In the MATLAB Command Window, enter:

```
modeladvisor("newModel")
```
8 In the Model Advisor, click **Open > Load Configuration** and select the custom configuration file customConfig.json to load the configuration file into the Model Advisor.

If the Model Advisor is unable to load a check listed in your configuration, the Model Advisor returns a warning in the MATLAB Command Window. To turn off these warnings, follow the steps listed in "Suppress Warning Message for Missing Checks" on page 7-7.
9 In the Model Advisor Check Selector pane, you can verify that you see the folders and checks specified in the custom configuration. If you expect a folder or check to appear in the Model Advisor and it does not, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3 and "Update the Environment to Include Your Custom Configuration" on page 7-19.
10 If you want the Model Advisor to use this configuration each time you open the model newModel, associate the loaded configuration file with the model by using one of these approaches:

   • In the Model Advisor, click **Open > Associate Configuration to Model**. The Model Advisor opens the Configuration Parameters dialog box for the model. The **Model Advisor configuration file** parameter lists the configuration file associated with the model. Click **OK**.

- In the MATLAB Command Window, provide the model name and configuration file name as inputs to the function `ModelAdvisor.setModelConfiguration`.

  ```
  ModelAdvisor.setModelConfiguration("newModel","customConfig.json");
  ```

For Model Advisor configuration files created in R2021b or earlier, use R2022a or later to open and re-save the configuration file in the Model Advisor Configuration Editor before associating the file with a model.

11 You can view the Model Advisor configuration file associated with the model by entering:

```
ModelAdvisor.getModelConfiguration("newModel")
```

12 To return to your default configuration, click **Open > Restore Default Configuration**.

For information on how to restore the default shipping configuration or set a new default configuration, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

Note The Model Advisor looks for configuration files in the following order:

1 The configuration that you specify by using the `configfile` input argument of the `modeladvisor` function.

2 The configuration associated with the model.

3 The default configuration. The *default configuration* is either the default shipping configuration or the default configuration that you set. For more information on the default configuration, see "Use the Model Advisor Configuration Editor to Customize the Model Advisor" on page 7-3.

The Model Advisor loads the first configuration file that it finds and ignores other configuration files.

Deploy Custom Configurations

When you create a custom configuration, often you *deploy* the custom configuration to your development group. Deploying the custom configuration allows your development group to review models using the same checks. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

To deploy a custom configuration:

1   Determine which files to distribute. You might need to distribute more than one file.

If You...	Using the...	Distribute...
Created custom Model Advisor checks	Customization file	• `sl_customization.m` • Files containing check and action callback functions (if separate)
Created custom Model Advisor configuration files	Model Advisor Configuration Editor	Configuration `.json` file

1   Distribute the files and tell the user to include these files on the MATLAB path.
2   Instruct the user to load the custom configuration.

Create and Deploy a Model Advisor Custom Configuration

To check that a model meets the standards and modeling guidelines of your company, you can customize the Model Advisor. This example shows you how to add custom checks to the Model Advisor and remove shipping checks that you do not require. You can save the custom check configuration and deploy it to others in your development group. Deploying a custom configuration allows your development group to review models using the same set of checks.

Define Custom Checks

This example defines four types of custom checks:

- An edit-time check that provides a fix action.
- A check that runs in only the Model Advisor and groups results by blocks and subsystems and provides a fix action.
- A check that runs only in the Model Advisor and verifies model configuration parameter settings.
- An edit-time check that specifies a constraint for a block parameter setting and provides a fix action.

The example files include the `sl_customization.m` file. This file contains the `sl_customization` function, which contains calls to functions that define the custom checks. Open and inspect the `sl_customization.m` file.

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

% Copyright 2019 The MathWorks, Inc.

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


% ----------------------------
% defines Model Advisor Checks
% ----------------------------
function defineModelAdvisorChecks
defineEditTimeCheck
defineDetailStyleCheck;
defineConfigurationParameterCheck;
defineNewBlockConstraintCheck;
```

The `sl_customization` function accepts a customization manager object that includes the `addModelAdvisorCheckFcn` method for registering custom checks. The input to this method is a handle to a function, `defineModelAdvisorChecks`, which contains calls to the four check definition functions that correspond to the four custom checks.

Edit-Time Check with Fix

The `defineEditTimeCheck.m` file contains the `defineEditTimeCheck` check definition function, which defines a check that checks whether Inport and Outport blocks have certain colors depending on their output data types. This check must check other edited blocks, but it does not have to check for affected blocks at the same level or across the entire model hierarchy. This check provides a fix that updates the color of the blocks that do not have the correct colors. The name of this check is **Check color of Inport and Outport blocks**. This check runs at edit-time and in the Model Advisor. Open and inspect the `defineEditTimeCheck.m` file.

```matlab
function defineEditTimeCheck

% Check the background color of Inport and Outport blocks.
rec = ModelAdvisor.Check("advisor.edittimecheck.PortColor");
rec.Title = 'Check color of Inport and Outport blocks';
rec.CallbackHandle = 'MyEditTimeChecks.PortColor';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec,'Demo');
```

The edit-time check has a class definition, `PortColor`, that derives from the `ModelAdvisor.EdittimeCheck` base class. For more information on how to create this type of check, see "Define Edit-Time Checks to Comply with Conditions that You Specify with the Model Advisor" on page 6-9. Create a folder named `+MyEditTimeChecks` and save `PortColor.m` to this folder.

```matlab
copyfile PortColor.m* +MyEditTimeChecks
```

Open and inspect the `PortColor.m` file.

```matlab
classdef PortColor < ModelAdvisor.EdittimeCheck
    % Check that ports conform to software design standards for background color.
    %
    %   Background Color              Data Types
    %   orange                        Boolean
    %   green                         all floating-point
    %   cyan                          all integers
    %   Light Blue                    Enumerations and Bus Objects
    %   white                         auto
    %


    methods
        function obj=PortColor(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            if strcmp(get_param(blk,'BlockType'),'Inport') || strcmp(get_param(blk,'BlockType'),

                dataType = get_param(blk,'OutDataTypeStr');
                currentBgColor = get_param(blk,'BackgroundColor');

                if strcmp(dataType,'boolean')
                        if ~strcmp(currentBgColor, 'orange')
                            % Create a violation object using the ModelAdvisor.ResultDetail class
                            violation = ModelAdvisor.ResultDetail;
                            ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID
                            violation.CheckID = obj.checkId;
                            violation.Description = 'Inport/Outport blocks with Boolean outputs
                            violation.title = 'Port Block Color';
                            violation.ViolationType = 'Warning';
                        end
                elseif any(strcmp({'single','double'},dataType))
                        if ~strcmp(currentBgColor, 'green')
                            violation = ModelAdvisor.ResultDetail;
                            ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID
```

```matlab
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with floating-point ou
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
            elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
                    if ~strcmp(currentBgColor, 'cyan')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with integer outputs s
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
            elseif contains(dataType,'Bus:')
                if ~strcmp(currentBgColor, 'lightBlue')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with bus outputs should be
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                end
             elseif contains(dataType,'Enum:')
                if ~strcmp(currentBgColor, 'lightBlue')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with enumeration outputs s
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                end
            elseif contains(dataType, 'auto')
                if ~strcmp(currentBgColor, 'white')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with auto outputs should b
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                end
            end
        end
end

function violation = finishedTraversal(obj)
    violation = [];
end

function success = fix(obj, violation)
    success = true;
    dataType = get_param(violation.Data,'OutDataTypeStr');
    if strcmp(dataType,'boolean')
        set_param(violation.Data,'BackgroundColor','orange');
    elseif any(strcmp({'single','double'},dataType))
        set_param(violation.Data,'BackgroundColor','green');
    elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
        set_param(violation.Data,'BackgroundColor','cyan');
```

7-25

```
                elseif contains(dataType,'Bus:')  || contains(dataType,'Enum:')
                    set_param(violation.Data,'BackgroundColor','lightBlue');
                elseif contains(dataType,'auto')
                    set_param(violation.Data,'BackgroundColor','white');
                end
            end
        end
end
```

Model Advisor Check with Fix

The `defineDetailStyleCheck.m` file contains the `defineDetailStyleCheck` check definition function, which defines a check that lists blocks whose names are not displayed below the blocks. This check provides a fix that moves those names below the blocks. The name of this check is **Check whether block names appear below blocks**. This check authoring style is for checks that only run in the Model Advisor. Open and inspect the `defineDetailStyleCheck.m` file.

```
function defineDetailStyleCheck

mdladvRoot = ModelAdvisor.Root;

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
% Create ModelAdvisor.Action object for setting fix operation.
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
rec.setAction(myAction);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

end


% -----------------------------
% This callback function uses the DetailStyle CallbackStyle type.
% -----------------------------
function DetailStyleCallback(system, CheckObj)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

% Find all blocks whose name does not appear below blocks
violationBlks = find_system(system, 'Type','block',...
    'NamePlacement','alternate',...
    'ShowName', 'on');
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
else

    for i=1:numel(violationBlks)
        ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)
```

```
        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below th
        ElementResults(i).Status = 'The following blocks have names that do not display below the
        ElementResults(i).RecAction =  'Change the location such that the block name is below the
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end

% -----------------------------
% This action callback function changes the location of block names.
% -----------------------------
function result = ActionCB(taskobj)
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
resultDetailObjs = checkObj.ResultDetails;
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block,'NamePlacement','normal');
end

result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
mdladvObj.setActionEnable(false);
end
```

This check uses the `setCallbackFcn` type of `DetailStyle`, which produces default formatting, so that you do not have to use the `ModelAdvisor.FormatTemplate` or the other Model Advisor formatting APIs to format the results that appear in the Model Advisor. For more information on how to create this type of check definition function, see "Fix a Model to Comply with Conditions that You Specify with the Model Advisor" on page 6-21.

Model Configuration Parameter Settings Check

The `defineConfigurationParameterCheck.m` file contains the `defineConfigurationParameterCheck` check definition function, which defines a check that identifies model configuration parameter settings that might impact MISRA C:2012 compliant code generation. The name of this check is **Check model configuration parameters**.

This check requires a supporting XML data file that must be on the MATLAB path and contain the model configuration parameter settings that you want to check. For this example, that file is `configurationParameterDataFile.xml`. For more information on how to create this check type, see "Create Model Advisor Check for Model Configuration Parameters" on page 6-27.

Open and inspect the `defineConfigurationParameterCheck.m` file.

```
function defineConfigurationParameterCheck

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.configurationParameter');
rec.Title = 'Check model configuration parameters';
rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec.TitleTips = 'Identify configuration parameters that might impact MISRA C:2012 compliant code
```

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

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

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

end
```

Block Parameter Constraint Check

The `defineNewBlockConstraintCheck.m` file contains the `defineNewBlockConstraintCheck` check definition function, which defines a check that identifies Logical Operator blocks that do not have a rectangular shape. The name of this check is **Check icon shape of Logical Operator blocks**.

A block parameter constraint check supports edit-time checking. For more information on this check type, see "Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters" on page 6-38.

Open and inspect the `defineNewBlockConstraintCheck.m` file.

```
function defineNewBlockConstraintCheck

rec = Advisor.authoring.createBlockConstraintCheck('com.mathworks.sample.blockConstraint',...
    'Constraints',@createBlockConstraints); % constraint creation is part of block constraint che
rec.Title = 'Check icon shape of Logical Operator blocks';
rec.TitleTips = 'Checks icon shape of Logical Operator blocks. Icon shape of Logical Operator sho

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

end

function constraints = createBlockConstraints()

% Create block constraints.
c1 = Advisor.authoring.PositiveBlockParameterConstraint;
c1.ID = 'ID_c1';
c1.BlockType = 'Logic';
c1.ParameterName = 'IconShape';
```

```
c1.SupportedParameterValues = {'rectangular'};
c1.ValueOperator = 'eq';

constraints = {c1};

end
```

The `createBlockConstraints` function defines the block constraint `c1`. The `Advisor.authoring.createBlockConstraintCheck` function has a `'Constraints'` name-value argument that calls the constraints creation function `createBlockConstraints`.

View Custom Checks in the Model Advisor

To confirm that your custom checks are available, open the Model Advisor.

1. In order for your custom checks to be visible in the Model Advisor, you must refresh the Model Advisor check information cache. At the MATLAB command prompt, enter:

```
Advisor.Manager.refresh_customizations();
```

2. Open the example model.

```
open_system('AdvisorCustomizationExample.slx');
```

3. On the **Modeling** tab, open the **Model Advisor**. You can also open the Model Advisor by entering this command at the MATLAB command prompt:

```
modeladvisor('AdvisorCustomizationExample.slx');

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

4. Expand the **By Product > Demo** folder. In the check definition functions, the `publish` command adds the checks to the **By Product > Demo** folder.
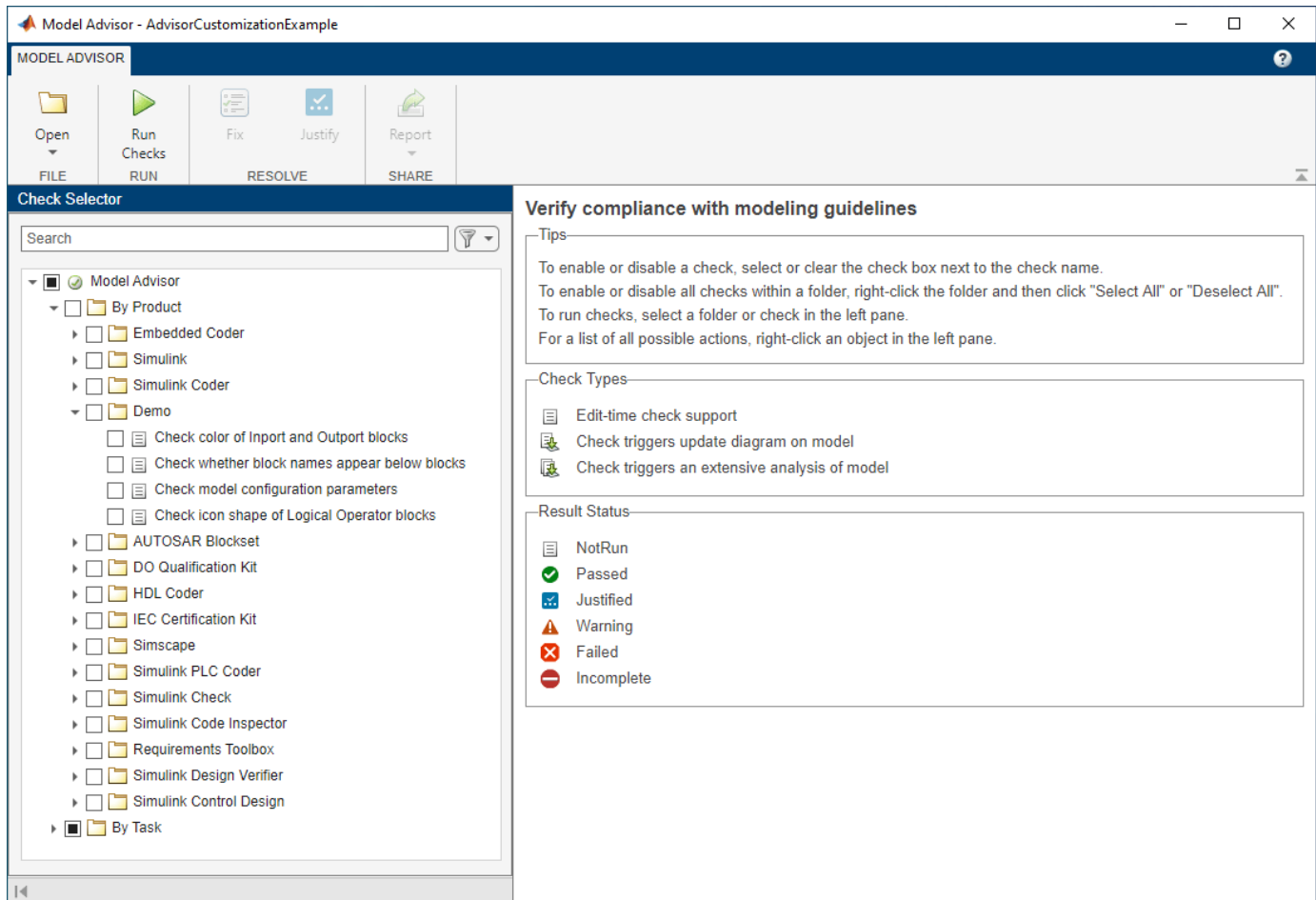
Specify and Deploy a Model Advisor Custom Configuration

To specify which checks to include in the Model Advisor and which checks to use during edit-time checking, use the Model Advisor Configuration Editor.

1. To open the Configuration Editor, in the Model Advisor, click **Open > Open Configuration Editor**.

2. To add or remove checks and folders, select from the options in the **Edit** section of the Model Advisor Configuration Editor.

3. To save a configuration, select **Save**. A window opens and prompts you to save the configuration as a JSON file. For this example, you do not have to save the configuration, because the file `demoConfiguration.json` file contains the four custom checks for this example.

4. Close the model and the Model Advisor Configuration Editor.

```
bdclose;
```

Associate a Custom Check Configuration with a Model and Address Check Issues

To address check issues, first associate the configuration with a model. Then, you can address issues during edit-time and in the Model Advisor.

1. Open the example model.

```
open_system('AdvisorCustomizationExample.slx');
```

2. Associate the custom configuration, `demoConfiguration.json`, with the model. When you associate a custom configuration with a model, the model uses the same check configuration every time you open a model. Click the **Modeling** tab and select **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, specify the path to the configuration file for the **Model Advisor** configuration file parameter. Alternatively, enter this command at the command prompt:

```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', 'demoConfiguration.json');
```

3. Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens. Select the **Edit-Time Checks** parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('AdvisorCustomizationExample','on');
```

At the top level of the model, the two Inport blocks have an output data type of `int32`. The blocks produce edit-time warnings because they should be cyan. The Outport block does not produce a warning because it has an `auto` data type and is white.
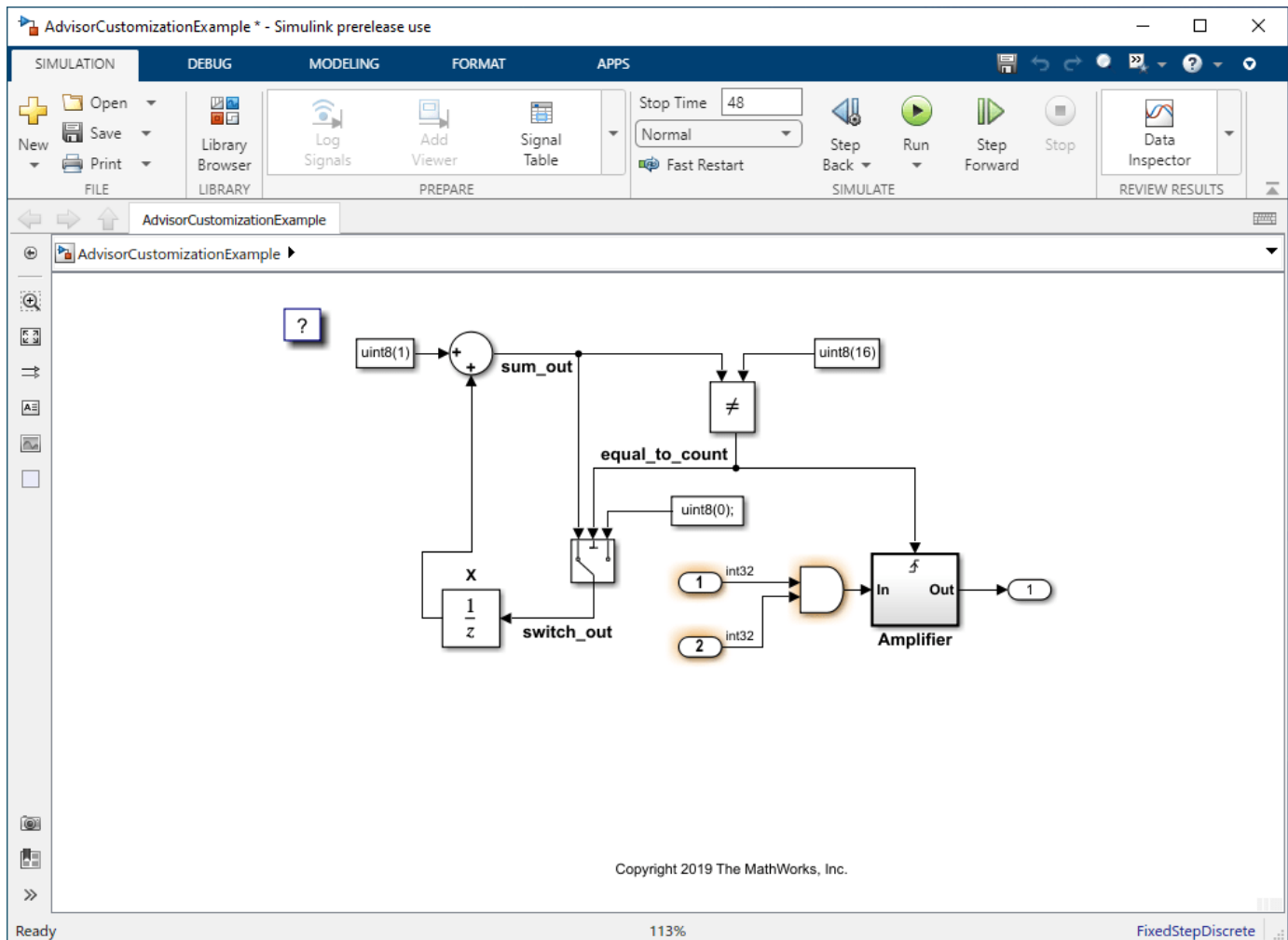
4. For each Inport block, click the edit-time warning window. Then click **Fix**. The color of the blocks changes to cyan and the warning goes away.

5. The Logical Operator block produces a warning because it should have a rectangular shape. Click the edit-time warning window and then click **Fix**. The shape of the Logical Operator block changes to a rectangle, and the warning goes away.

6. Now that you have addressed the edit-time check warnings, open the Model Advisor to address any remaining check issues.

```
modeladvisor('AdvisorCustomizationExample.slx');
```

Model Advisor is removing the existing report.

7. The Model Advisor contains the four checks in the custom configuration. Click **Run Checks**. The two checks that you addressed during edit-time pass. The other two checks produce warnings.
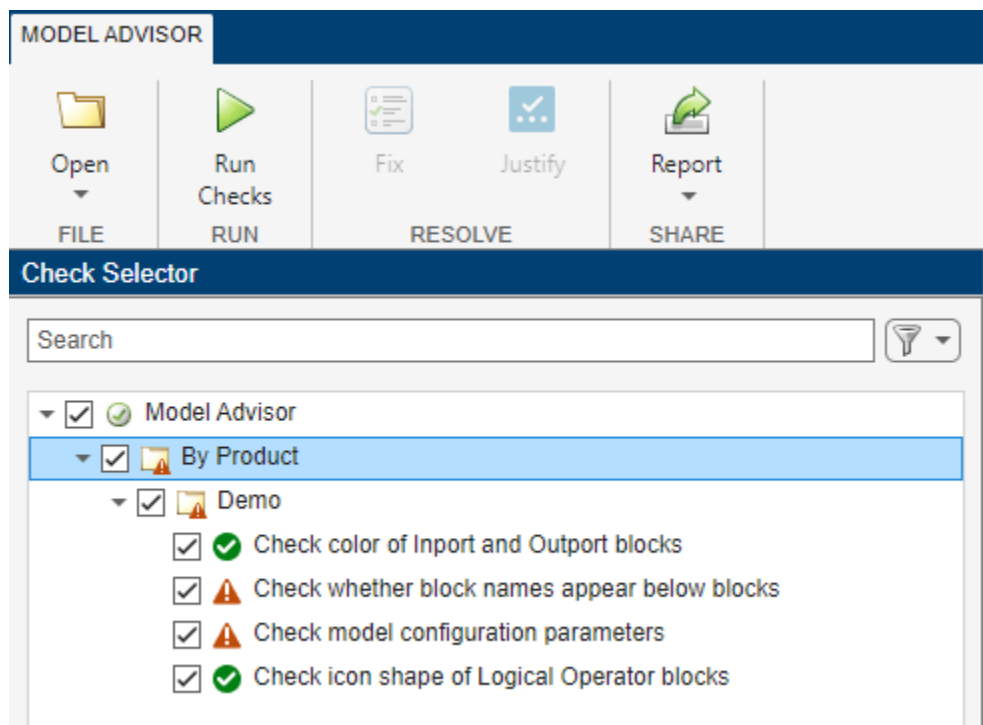
8. Click the **Check whether block names appear below blocks** check. To apply a fix and resolve the warnings, in the right pane, click **Fix**.

9. Click the **Check model configuration parameters** check. To apply a fix and resolve the warnings, click **Fix**.

10. Rerun the checks. They now pass.

11. Close the model and the Model Advisor.

```
bdclose;
```

12. Remove the files from your working directory. Refresh the Model Advisor check information cache by entering this command:

```
Advisor.Manager.refresh_customizations
```

Programmatically Run a Model Advisor Custom Configuration and View Results

You can programmatically run a Model Advisor configuration and then open the results in the Model Advisor.

1. Call the `ModelAdvisor.run` function.

```
SysResultObjArray = ModelAdvisor.run({'AdvisorCustomizationExample'},...
'Configuration','demoConfiguration.json');
```

2. View the results in the Model Advisor:

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

3. Click **Continue** in the dialog box. You can now apply fixes and resolve warnings.

4. Close the model and the Model Advisor.

```
bdclose;
```

5. Remove the files from your working directory. Refresh the Model Advisor check information cache by entering this command:

```
Advisor.Manager.refresh_customizations
```

See Also
`ModelAdvisor.Check` | `ModelAdvisor.EdittimeCheck`

More About
- "Define Custom Model Advisor Checks" on page 6-45
- "Justify Violated Blocks from the Model Advisor Check Analysis" on page 3-16
- "Define Custom Edit-Time Checks that Fix Issues in Architecture Models" on page 6-17
- "Run Custom Model Advisor Checks on Architecture Models" on page 3-106

8

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 To open the Model Slice Manager, on the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

   When you open the Model Slice Manager, Model Slicer compiles the model. You then configure the model slice properties.

4 In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.

5 Set the slice properties:

   • **Name**: `Out1Slice`
   • **Color**: ▮ (magenta)
   • **Signal Propagation**: `upstream`

   Model Slicer can also highlight the constructs downstream of or bidirectionally from a block in your model, depending on which direction you want to trace the signal propagation.

6 Add `Out1` as a starting point. In the model, right-click `Out1` and select **Model Slicer > Add as Starting Point**.
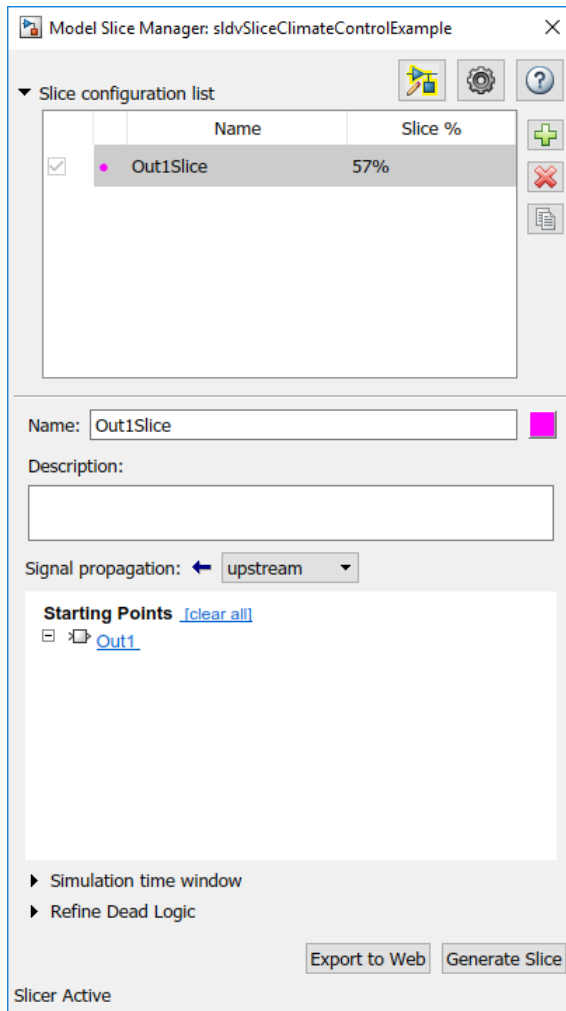
The Model Slicer now highlights the upstream constructs that affect `Out1`.
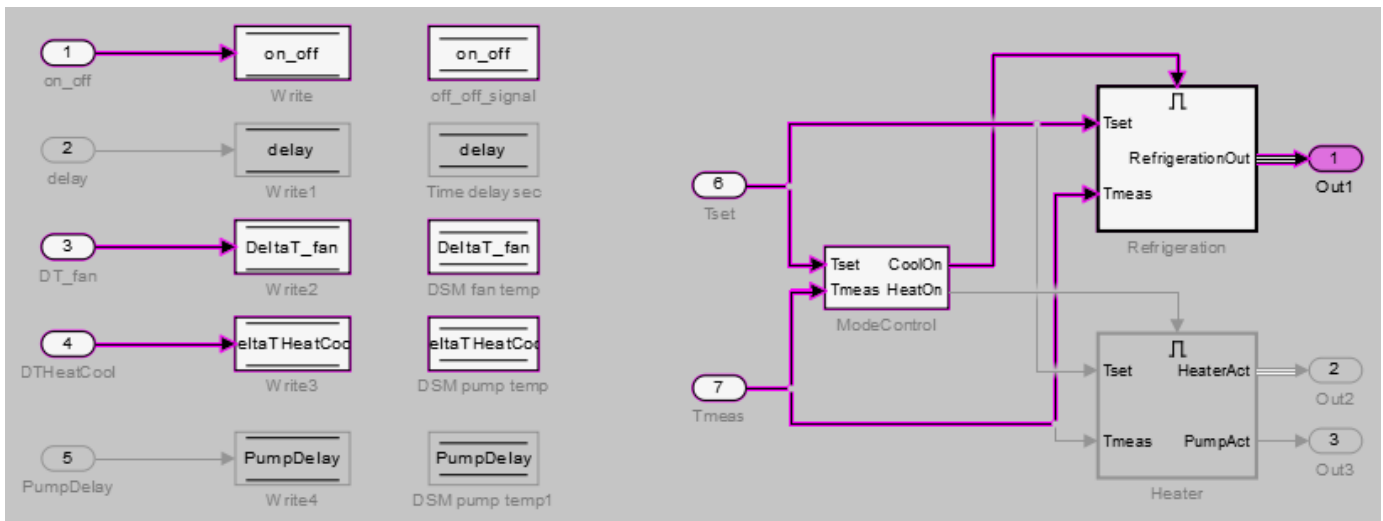
If you create two slice configurations, you can highlight the intersecting portions of their highlights. Create a new slice configuration and view the intersecting portions of the slice configuration you created above and the new slice configuration:

1   Create a new slice configuration with the following properties

   - **Name**: Out3Slice
   - **Color**: ▉ (red)
   - **Signal Propagation**: upstream
   - **Starting point**: Out3
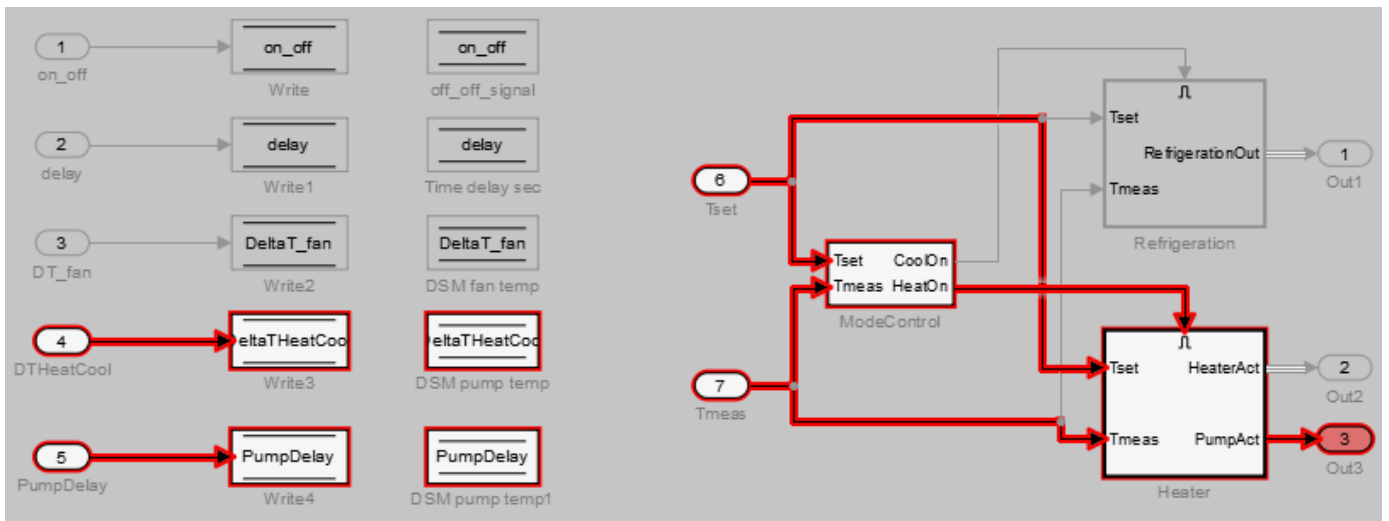
2   In the Model Slice Manager, select both the Out1Slice slice configuration and the Out3Slice slice configuration.
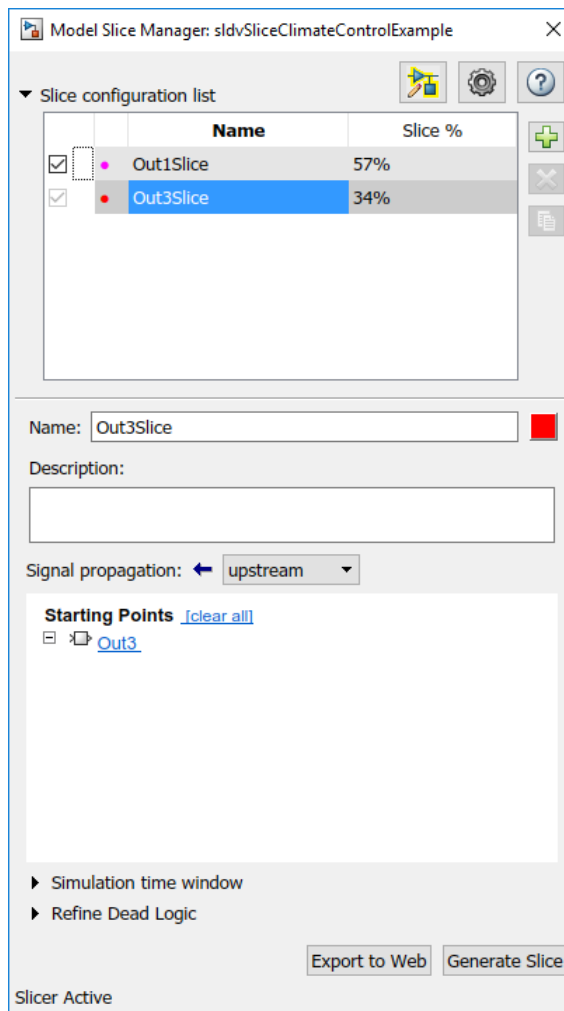
Model Slicer highlights portions of the model as follows:

- The portions of the model that are exclusively upstream of `Out1` are highlighted in cyan.
- The portions of the model that are exclusively upstream of `Out3` are highlighted in red.
- The portions of the model that are upstream of both `Out1` and `Out3` are highlighted in black.
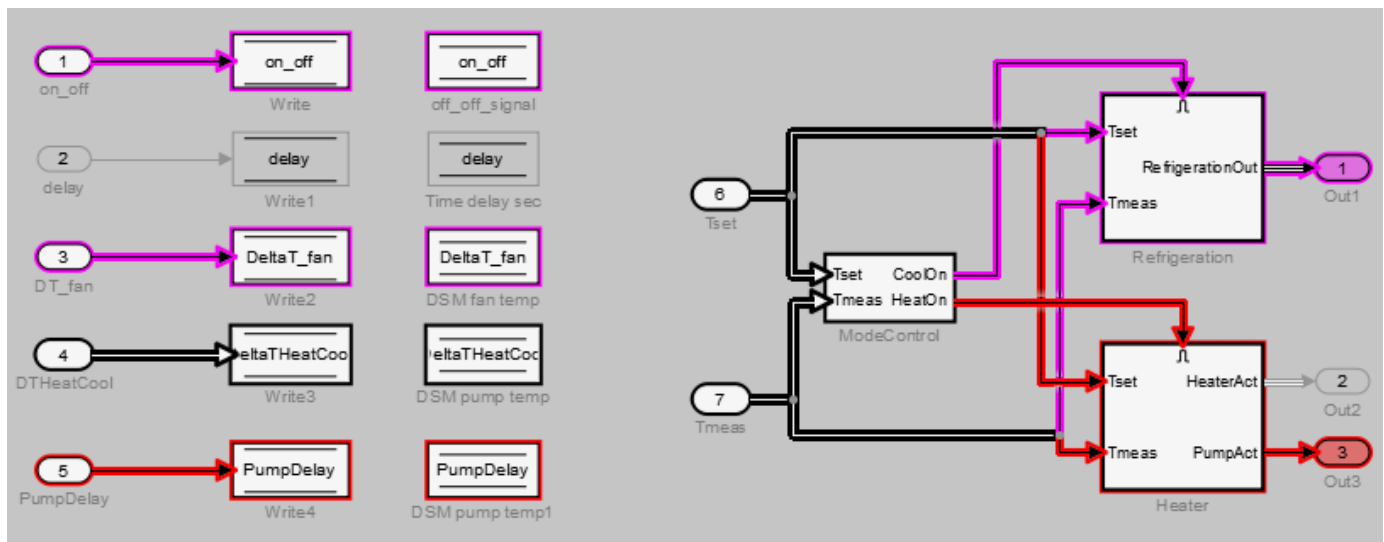
After you highlight a portion of your model, you can then refine the highlighted model to an area of interest. Or, you can create a simplified standalone model containing only the highlighted portion of your model.

To view the details of the highlighted model in web view, click **Export to Web**. The web view HTML file is stored in `<current folder>\<model_name>\webview.html`.

See Also

More About

- "Refine Highlighted Model" on page 8-12
- "Create a Simplified Standalone Model" on page 8-28
- "Model Slicer Considerations and Limitations" on page 8-43

Highlight Dependencies for Multiple Instance Reference Models

To highlight the functional dependencies in a Simulink model with multiple instances of a referenced model, use Model Slicer. You can use Model Slicer on a Simulink model that contains single or multiple references to a same model in normal simulation mode.

This example shows the behavior 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');
```
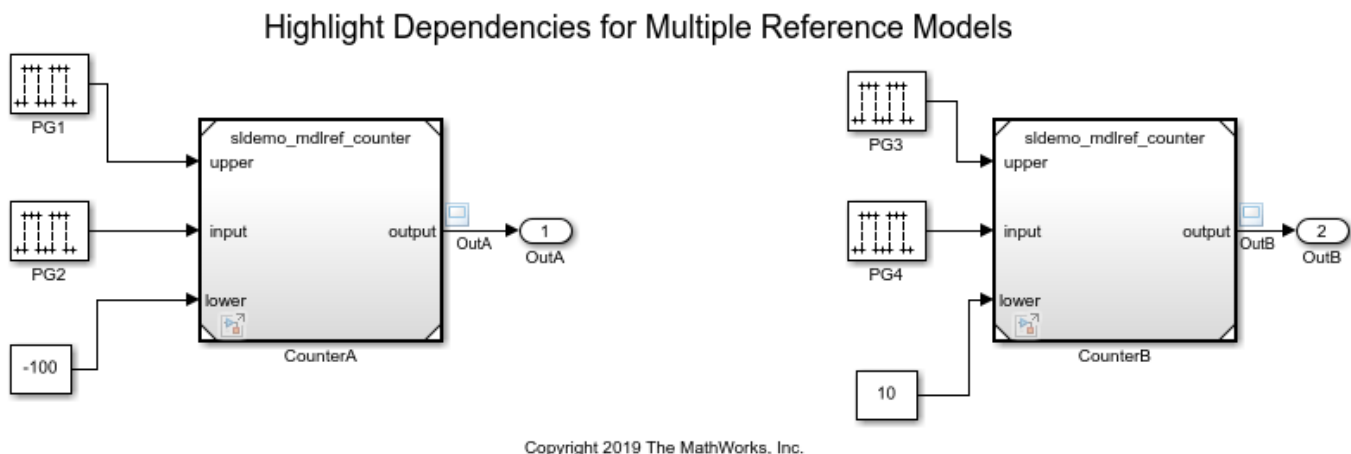

Highlight Dependencies for Multiple Reference Models

Copyright 2019 The MathWorks, Inc.

2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

3. In the Model Slicer window, click **Add all outports**. This sets **OutA** and **OutB** as starting points.

4. Ensure that the **Signal Propagation** is set to **upstream**.

5. In the **Simulation time window** section, click **Run simulation**.
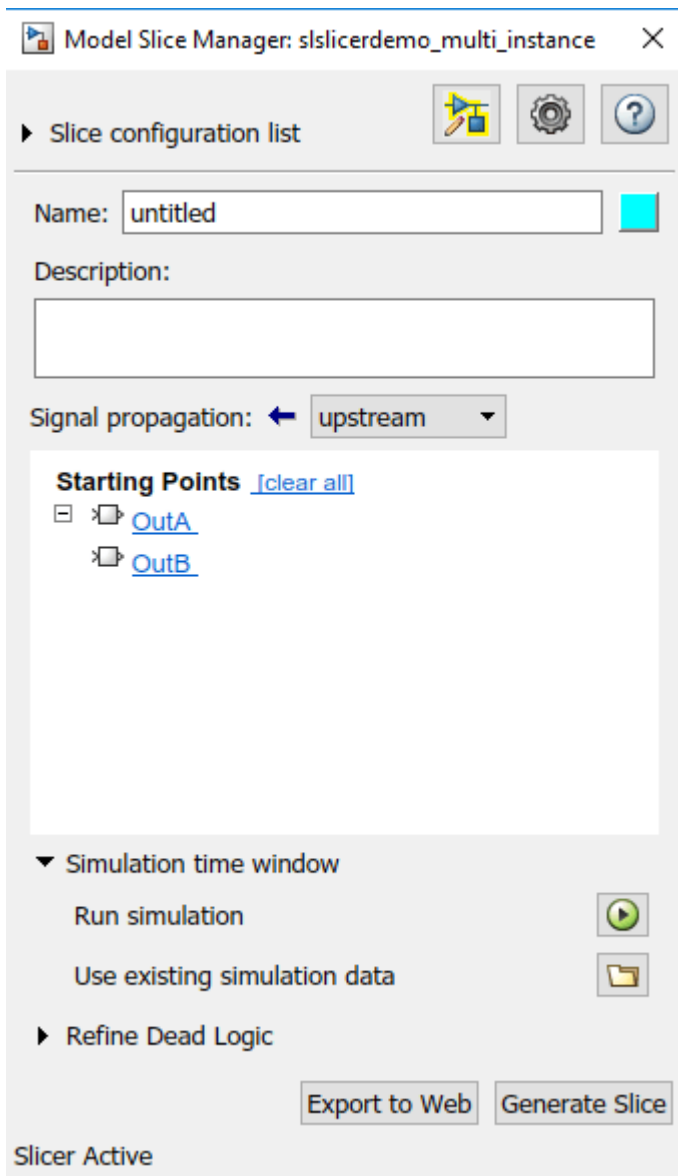
6. In the simulation time window, click **OK**. The model simulation starts.
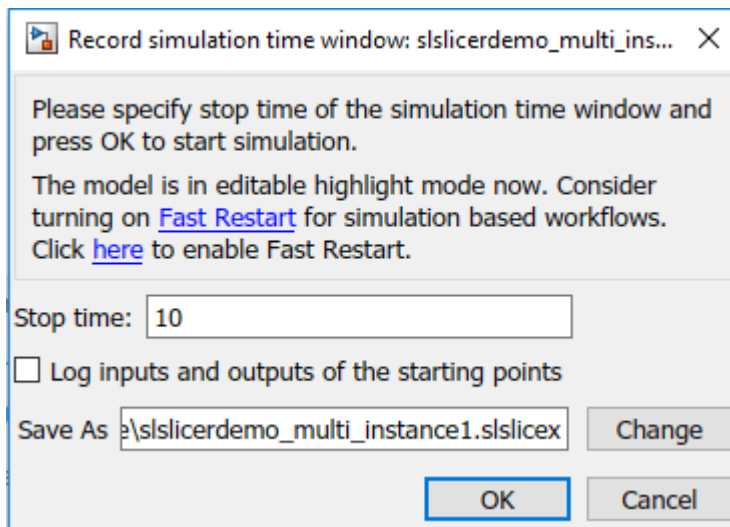
7. The simulated model highlights the upstream dependency of the outports **OutA** and **OutB**.
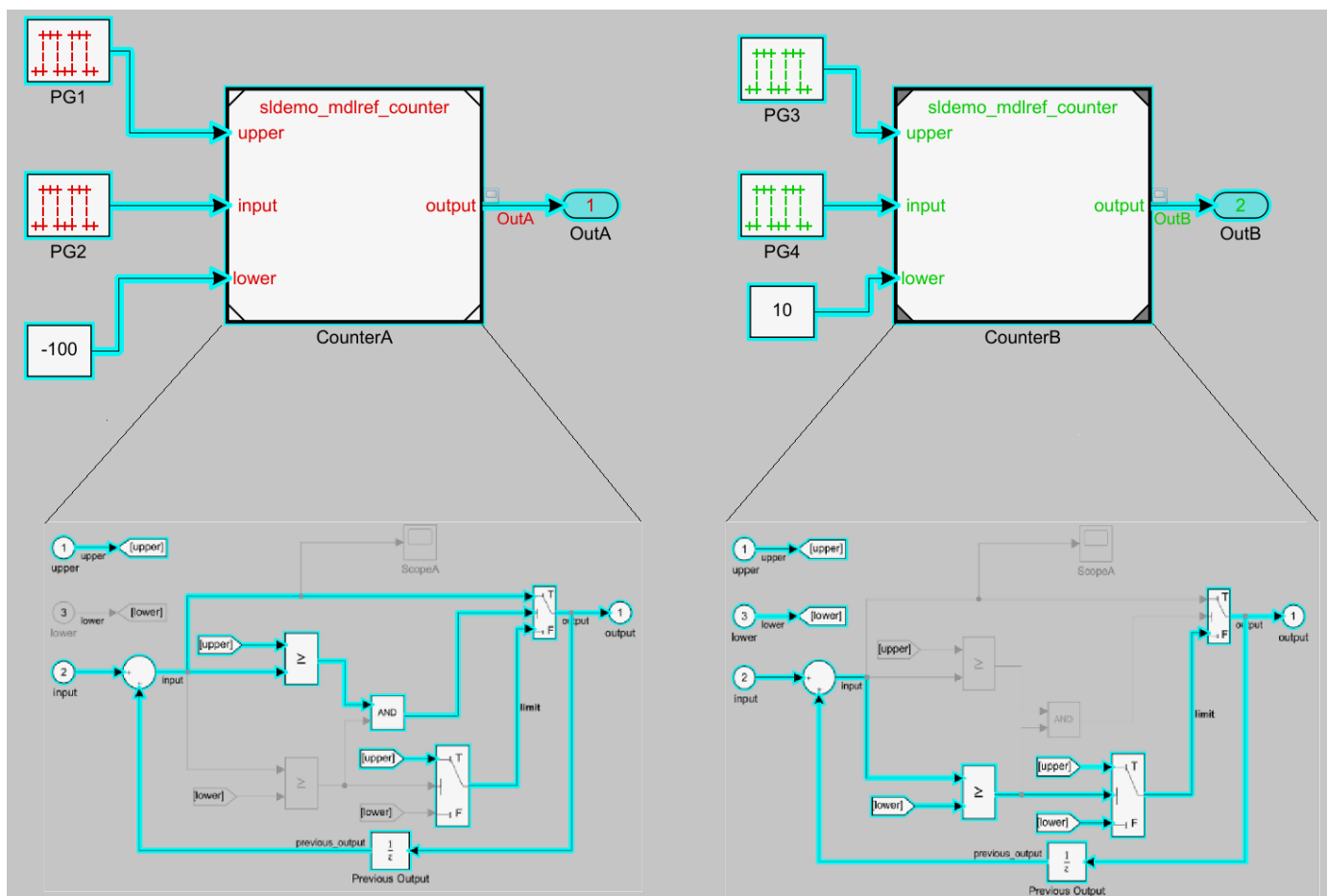
You can notice that the referenced model in both the instances shows different signal propagations highlighted by Simulink Slicer for which the signal travels.

8. To generate the slice, click **Generate Slice**.

More About

*   "Highlight Functional Dependencies" on page 8-2
*   "Model Slicer Considerations and Limitations" on page 8-43

Refine Highlighted Model

After you highlight a model using Model Slicer, you can refine the dependency paths in the highlighted portion of the model. Using Model Slicer, you can refine a highlighted model by including only those blocks used in a portion of a simulation time window, or by excluding blocks or certain inputs of switch blocks. By refining the highlighted portion of your model, you can include only the relevant parts of your model.

In this section...
"Define a Simulation Time Window" on page 8-12
"Exclude Blocks" on page 8-16
"Exclude Inputs of a Switch Block" on page 8-19

Define a Simulation Time Window

You can refine a highlighted model to include only those blocks used in a portion of a simulation time window. Defining the simulation time window holds some switch blocks constant, and as a result removes inactive inputs.

1   Add the example folder to the search path.

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

2   Open the sldvSliceClimateControlExample model.

    sldvSliceClimateControlExample

3   On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

    When you open the Model Slice Manager, Model Slicer compiles the model. You then configure the model slice properties.

4   In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.

5   Set the slice properties:

    • **Name**: Out1Simulation

    • **Color**: ▢ (cyan)

    • **Signal propagation**: upstream
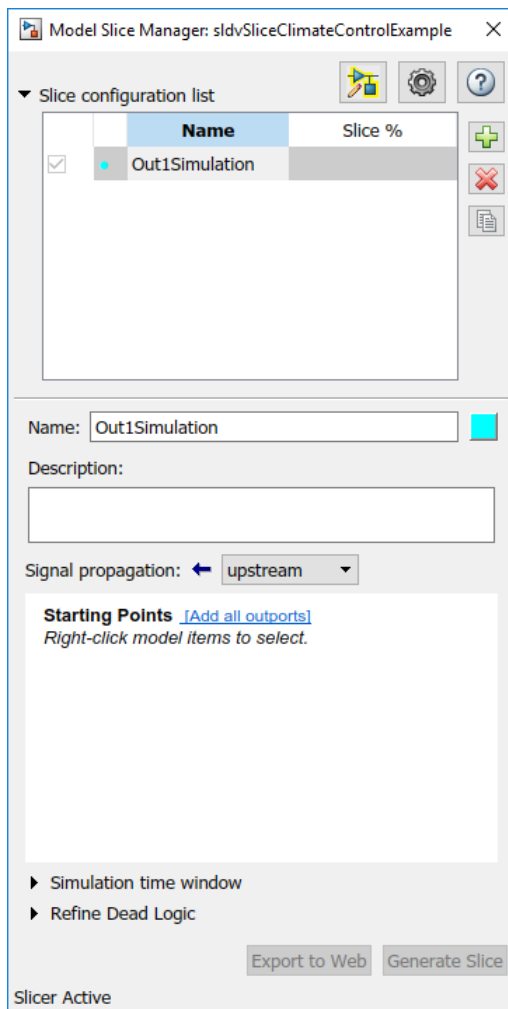
6   In the top level of the model, select the `Out1` block as the slice starting point. Right-click the `Out1` block and select **Model Slicer > Add as Starting Point**.

The model is highlighted.

7   In the Model Slice Manager, select **Simulation time window**.

8   To specify the stop time of the simulation time window, click the run simulation button in the Model Slice Manager.

9   Set the **Stop time** to `10`.

10 Click **OK** to start the simulation.
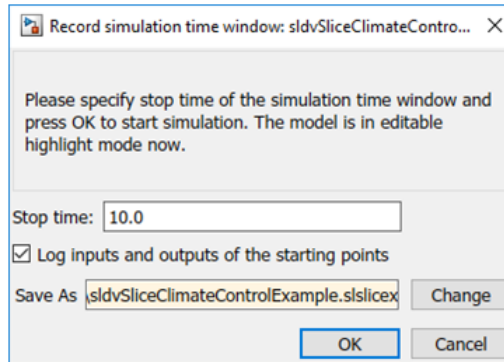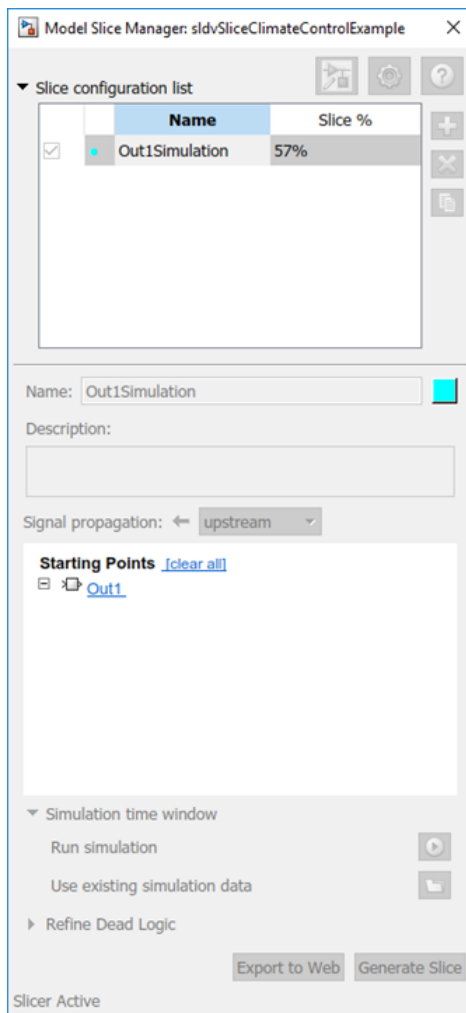
The path is restricted to only those blocks that are active until the stop time that you entered.
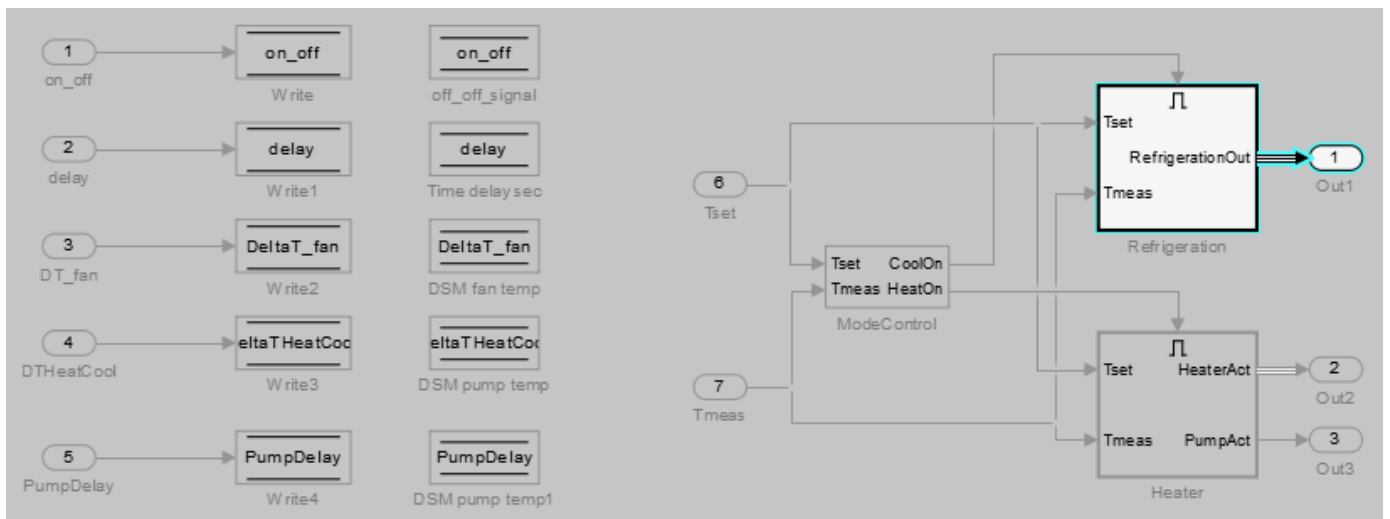
11 To highlight the model for a defined simulation time window, set the **Stop** time to 5. Click **Highlight**.

12 To see how this constraint affects the highlighted portion of the model, open the `Refrigeration` subsystem.

The highlighted portion of the model includes only the input ports of switches that are active in the simulation time window that you specified.
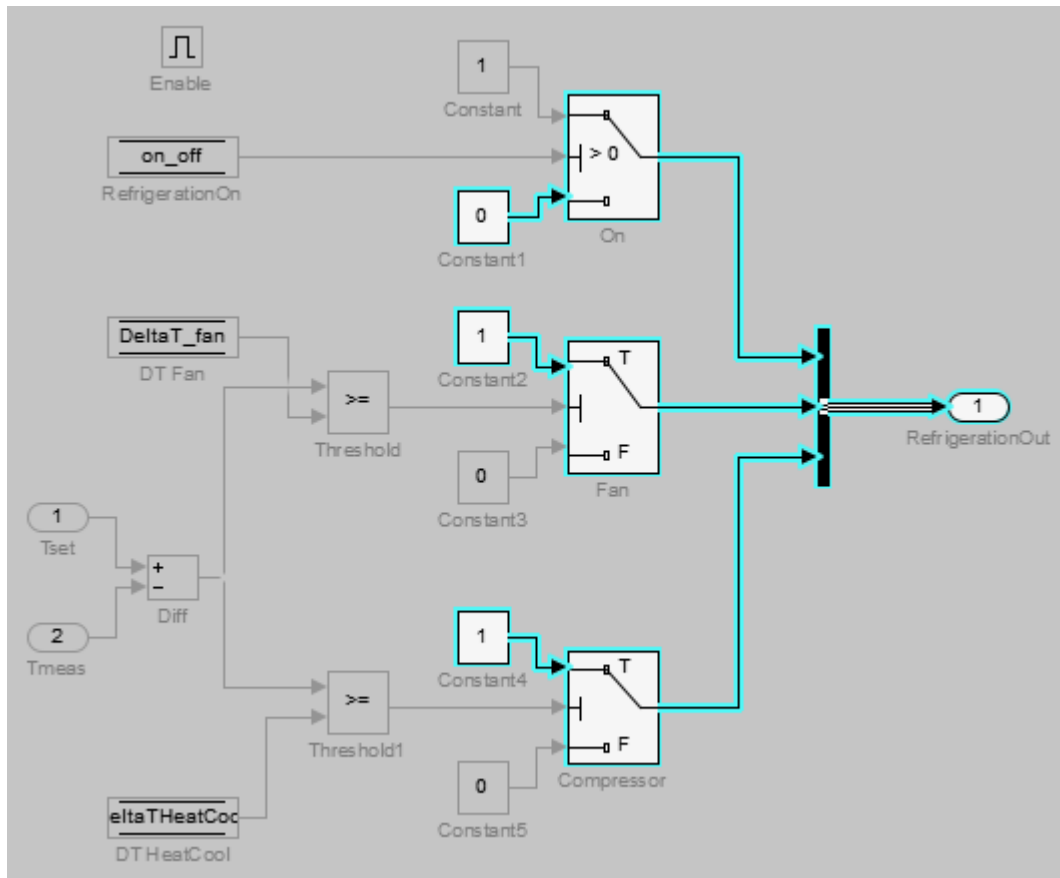
After you refine your highlighted model to include only those blocks used in a portion of a simulation time window, you can then "Create a Simplified Standalone Model" on page 8-28 incorporating the highlighted portion of your model.

Exclude Blocks

You can refine a highlighted model to exclude blocks from the analysis. Excluding a block halts the propagation of dependencies, so that signals and model items beyond the excluded block in the analysis direction are ignored.

Exclusion points are useful for viewing a simplified set of model dependencies. For example, control feedback paths create wide dependencies and extensive model highlighting. You can use an exclusion point to restrict the analysis, particularly if your model has feedback paths.

Note Simplified standalone model creation is not supported for highlighted models with exclusion points.

1. In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.

2. To add a new slice configuration, click the add new button ![plus icon].

3. Set the slice properties:

- **Name**: `Out1Excluded`
- **Color**: ■ (red)
- **Signal Propagation**: `upstream`

4   In the top level of the model, select the `Out1` block as the slice starting point. Right-click the `Out1` block and select **Model Slicer > Add as Starting Point**.
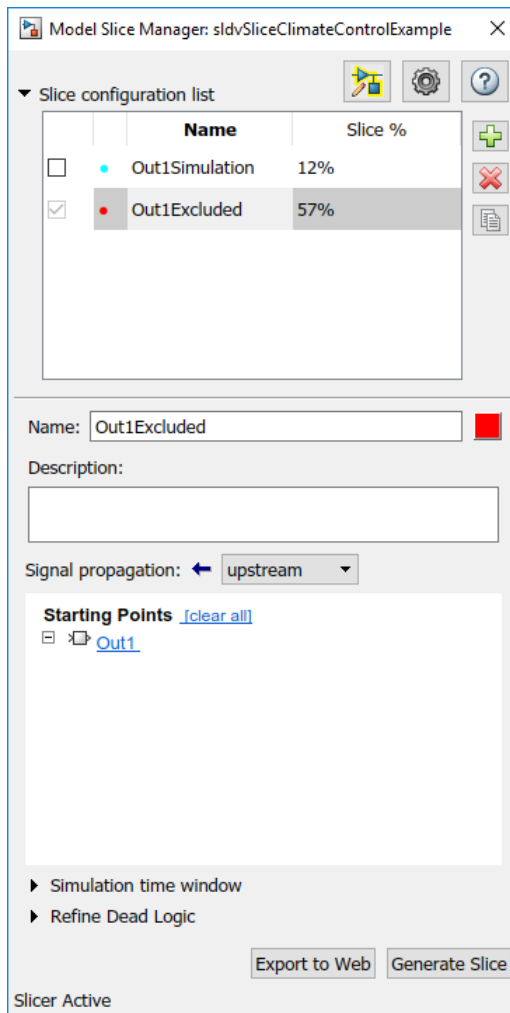
The model is highlighted.

5   To open the subsystem, double-click `Refrigeration`.

6   Right-click the `Fan` switch block, and then select **Model Slicer > Add as Exclusion Point**.

The blocks that are exclusively upstream of the `Fan` switch block are no longer highlighted. The `DT Fan` Data Store Read block is no longer highlighted.
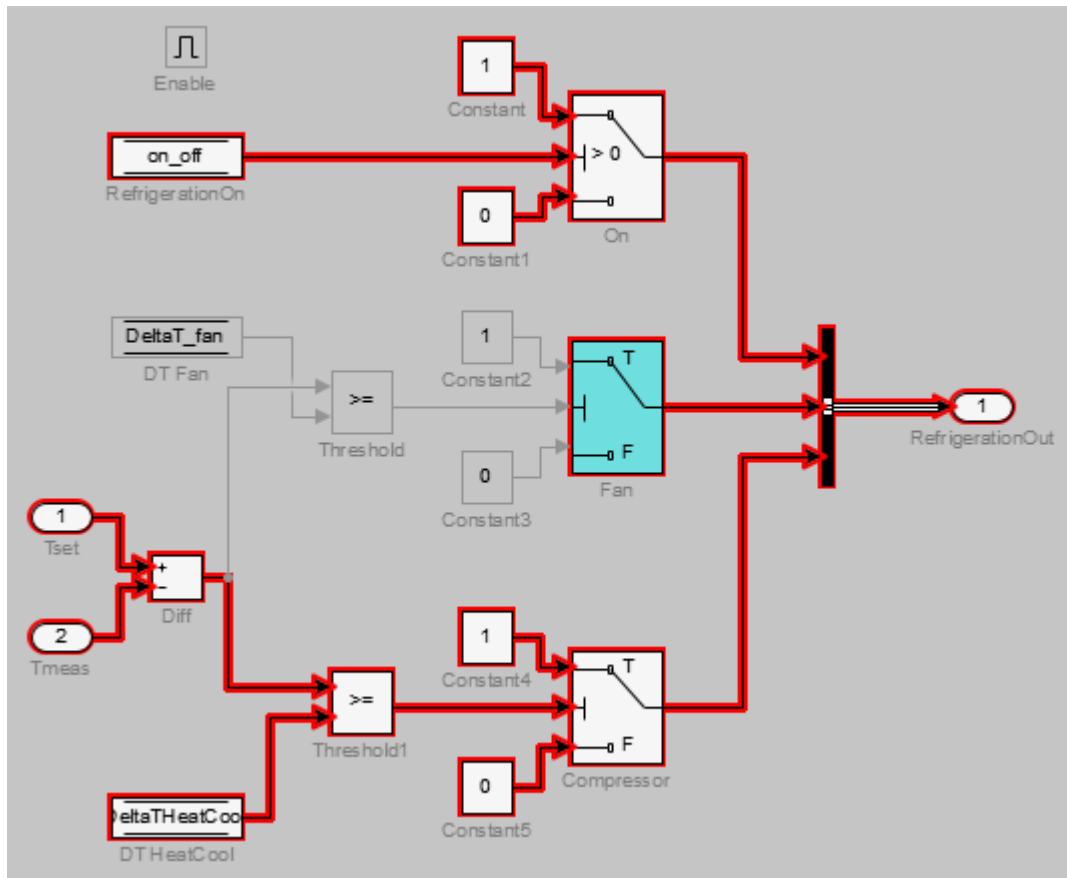
7   To see how this constraint affects the highlighted portion of the model, view the parent system.

     The `DSM fan temp` Data Store Memory block and the `Write2` Data Store Write block are no longer highlighted, because the `DT Fan` Data Store Read in the `Refrigeration` subsystem no longer accesses them.
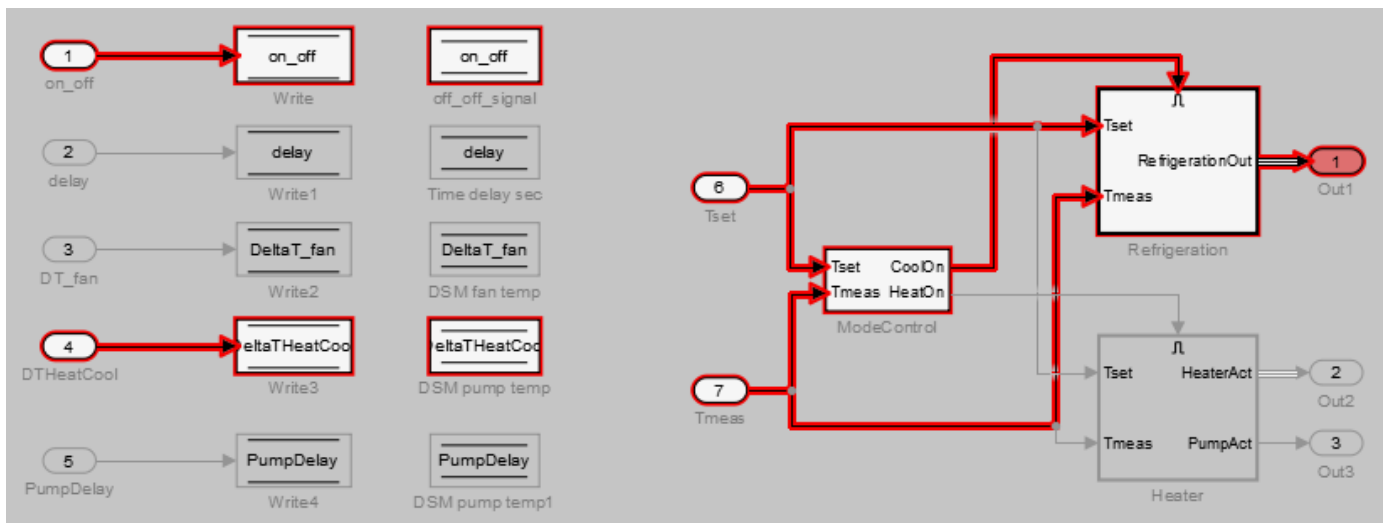
Exclude Inputs of a Switch Block

For complex signal routing, you can constrain the dependency analysis paths to a subset of the available paths through switch blocks. Constraints appear in the Model Slice Manager.

Note Simplified standalone model creation is not supported for highlighted models with constrained switch blocks.

---

1 Double-click `Refrigeration` to open the subsystem.

2 Constrain the `On` switch block:

- Right-click the switch block and select **Model Slicer > Add Constraint**.
- In the Constraints dialog box, select **Port 3**.
- Click **OK**.

The path is restricted to the `Constant1` port on the switch. The blocks that are upstream of **Port 1** and **Port 2** of the constrained switch are no longer highlighted. Only the blocks upstream of **Port 3** are highlighted.
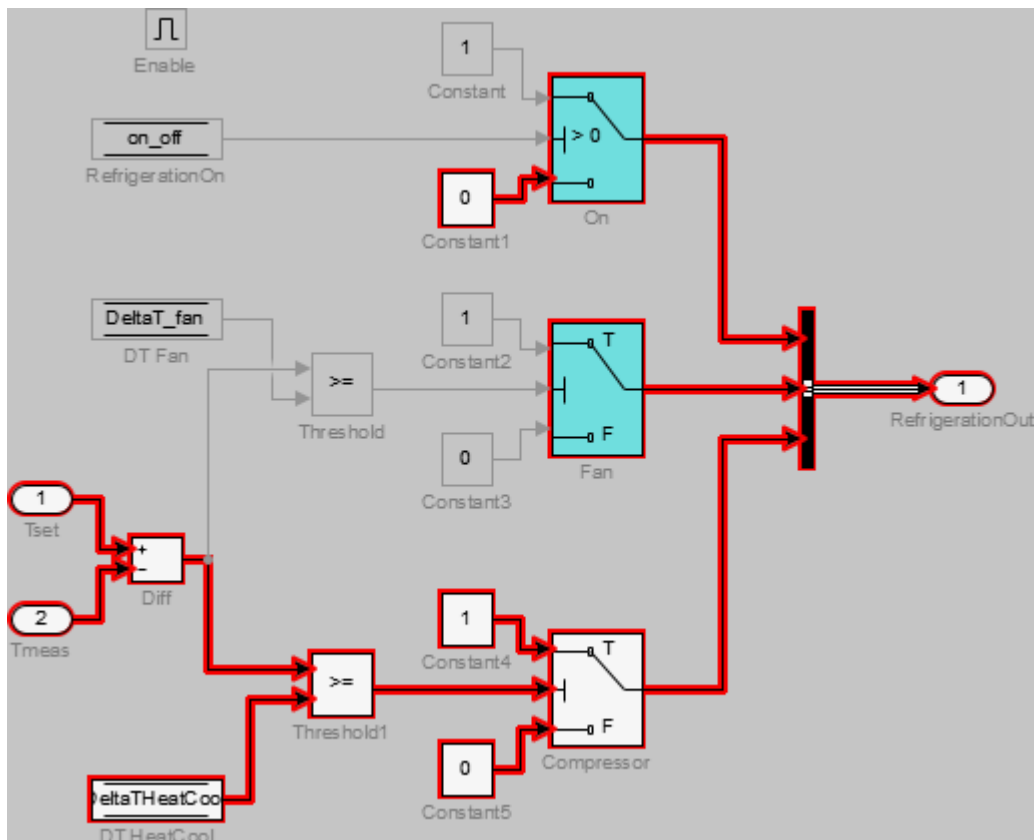
3 To see how this constraint affects the highlighted portion of the model, view the parent system.
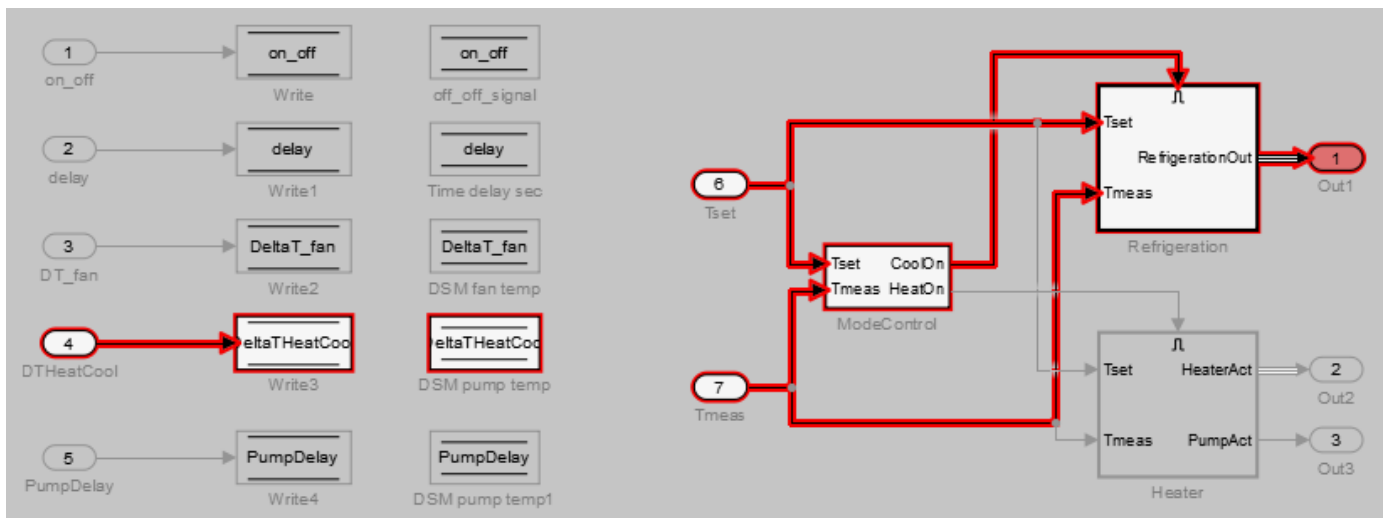
See Also

More About

- "Create a Simplified Standalone Model" on page 8-28

- "Model Slicer Considerations and Limitations" on page 8-43

Refine Dead Logic for Dependency Analysis

To refine the dead logic in your model for dependency analysis, use the Model Slicer. To provide an accurate slice, Model Slicer leverages Simulink Design Verifier dead logic analysis to remove the unreachable paths in the model. Model Slicer identifies the dead logic and refines the model slice for dependency analysis. For more information on Dead logic, see "Dead Logic Detection" (Simulink Design Verifier).

Analyze the Dead Logic

This example shows how to refine the model for dead logic. The `sldvSlicerdemo_dead_logic` model consists of dead logic paths that you refine for dependency analysis.

1. Open the `sldvSlicerdemo_dead_logic` model.

2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```

**Simulink Design Verifier
Cruise Control Test Generation**

Controller

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.

Run
(double-click)

Toggle Speed
Constraint
(double-click)

View Options
(double-click)

Toggle Constraint

Copyright 2006-2018 The MathWorks, Inc.

Open the `Controller` subsystem and add the outport `throt` as the starting point.

The Model Slicer highlights the upstream dependency of the `throt` outport.

2. In the Model Slice Manager, select **Refine Dead Logic**.

3. Click **Get Dead Logic Data**.

4. Specify the **Analysis time** and run the analysis. You can import existing dead logic results from the `sldvData` file or load existing `.slslicex` data for analysis. For more information, see "Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results" on page 8-63.

As the `set` input is equal to `true`, the `False` input to switch is removed for dependency analysis. Similarly, the output of block `OR` is always `true` and removed from the model slice.

See Also

More About

- "Refine Highlighted Model" on page 8-12
- "Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results" on page 8-63

Create a Simplified Standalone Model

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

Note Simplified standalone model creation is not supported for highlighted models with exclusion points or constrained switch blocks. If you want to view the effects of exclusion points or constrained switch blocks on a simplified standalone model, first create the simplified standalone model, and then add exclusion points or constrained switch blocks.

1   Highlight a portion of your model using Model Slicer.

    See "Highlight Functional Dependencies" on page 8-2 and "Refine Highlighted Model" on page 8-12.
2   In the Model Slice Manager, click **Generate slice**.
3   In the **Select File to Write** dialog box, select the save location and enter a model name.

    The simplified standalone model contains the highlighted model items.
4   To remove highlighting from the model, close the Model Slice Manager.

When generating a simplified standalone model from a model highlight, you might need to refine the highlighted model before the simplified standalone model can compile. See the "Model Slicer Considerations and Limitations" on page 8-43 for compilation considerations.

See Also

More About
•   "Configure Model Highlight and Sliced Models" on page 8-40

Highlight Active Time Intervals by Using Activity-Based Time Slicing

Stateflow states and transitions can be active, inactive, or sleeping during model simulation. You can use Model Slicer to constrain model highlighting to only highlight the time intervals in which certain Stateflow "Represent Operating Modes by Using States" (Stateflow) and "Transition Between Operating Modes" (Stateflow) are active. Therefore, you are able to refine your area of interest to only those portions of your model that affect model simulation during the operation of the selected states and transitions. You can also constrain model highlighting to the intersection of the time intervals of two or more states or transitions.

In this section...
"Highlighting the Active Time Intervals of a Stateflow State or Transition" on page 8-29
"Activity-Based Time Slicing Limitations and Considerations" on page 8-35
"Stateflow State and Transition Activity" on page 8-35

Highlighting the Active Time Intervals of a Stateflow State or Transition

The `slslicer_fuelsys_activity_slicing` model contains a fault-tolerant fuel control system. In this tutorial, you use activity-based time slicing to refine a model highlight to only those time intervals in which certain states and transitions are active. You must be familiar with how to "Highlight Functional Dependencies" on page 8-2 by using Model Slicer.

Create a Dynamic Slice Highlight for an Area of Interest

1   Add the example folder to the search path.

   `addpath(fullfile(docroot,'toolbox','simulink','examples'))`
2   Open the `slslicer_fuelsys_activity_slicing` model.

   `open_system('slslicer_fuelsys_activity_slicing')`
3   Open Model Slicer and add the `control logic` Stateflow chart in the fuel rate controller subsystem as a Model Slicer starting point.
4   Highlight the portions of the model that are upstream of the `control logic` Stateflow chart.
5   Simulate the model within a restricted simulation time window (maximum 20 seconds) to highlight only the areas of the model upstream of the starting point and active during the time window of interest.

fuel rate controller
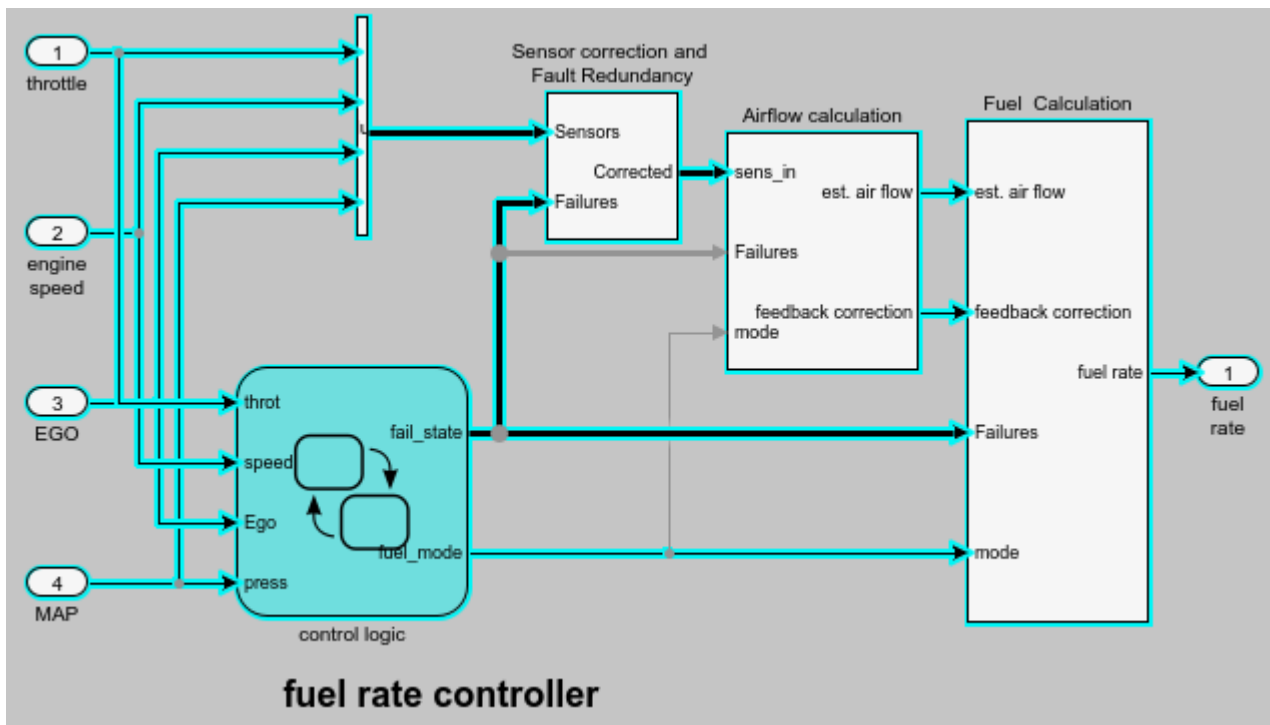
Constrain the Model Highlight to the Active Time Interval of a Stateflow State

1   On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
2   Navigate to the `control logic` Stateflow chart in the `fuel rate controller` subsystem.

    `open_system('slslicer_fuelsys_activity_slicing/fuel rate controller/control logic')`
3   To constrain the model highlight to only those time intervals in which the **Fueling_Mode >
    Running > Low_Emissions > Warmup** state is active, right-click the `Warmup` state and select
    Model Slicer > Constrain to active time intervals for "Warmup".
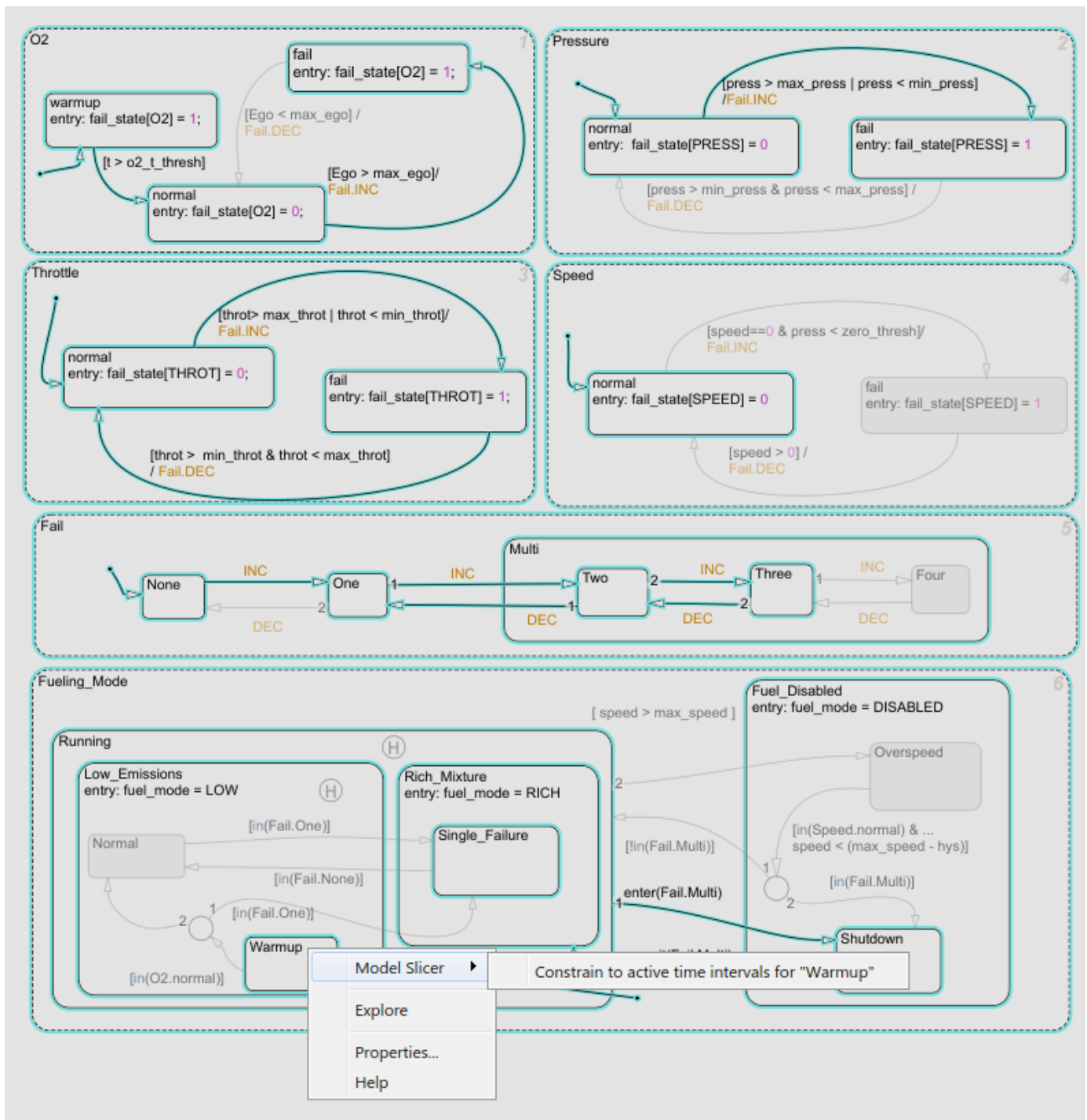
Model Slicer is updated to highlight only those portions of the model that are active during the time intervals in which the `warmup` state is active.

The Model Slice Manager is also updated to show the time interval in which the `warmup` state is active:

Actual simulation time: 0.01 to 3.86 seconds : 1 active interval

The highlight shows a `normal` to `fail` transition in the `Pressure` state, showing that a pressure failure occurred during the time interval in which the `Warmup` state was active.

Constrain the Model Highlight to the Intersection of the Active Time Intervals of a Stateflow State and Transition

1   Clear any time interval constraints from the Model Slice Manager.
2   Constrain the model highlight to only those time intervals in which the **O2 > fail** state is active.
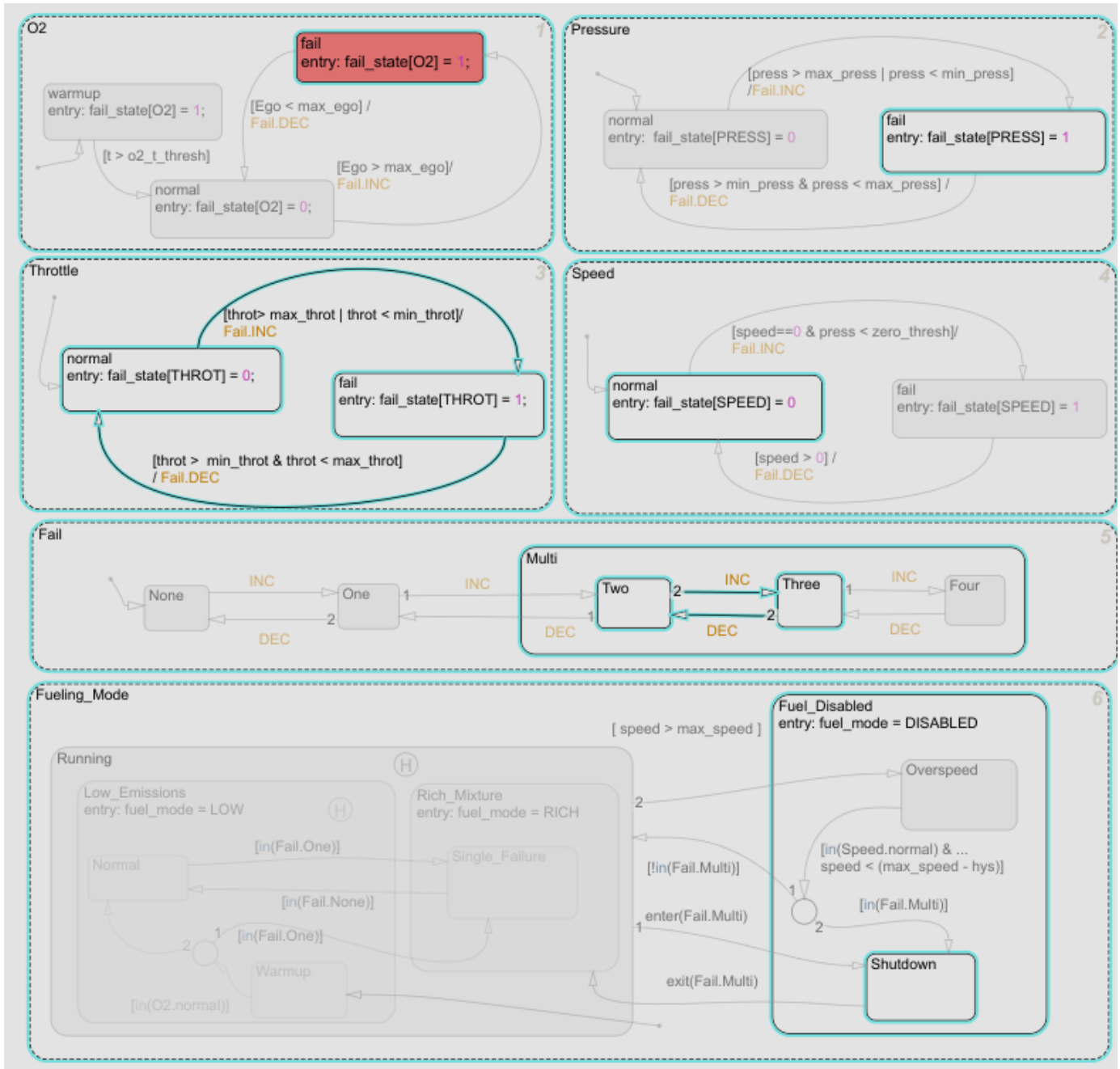
Model Slicer is updated to highlight only those portions of the model that are active during the time intervals in which the **O2 > fail** state is active. The Model Slice Manager is also updated to show the time interval in which the **O2 > fail** state is active:

Actual simulation time: 4.83 to 20 seconds : 1 active interval

3 To constrain the highlighting to the time interval in which the **O2 > fail** state is active and the `normal` to `fail` transition occurs for the `Throttle` chart, right-click the `normal` to `fail` transition and add it as a constraint. Model Slicer is updated to highlight only those portions of the model that are active during the intersection of the time intervals in which the **O2 > fail** state is active and the `normal` to `fail` transition occurs for the `Throttle` chart.
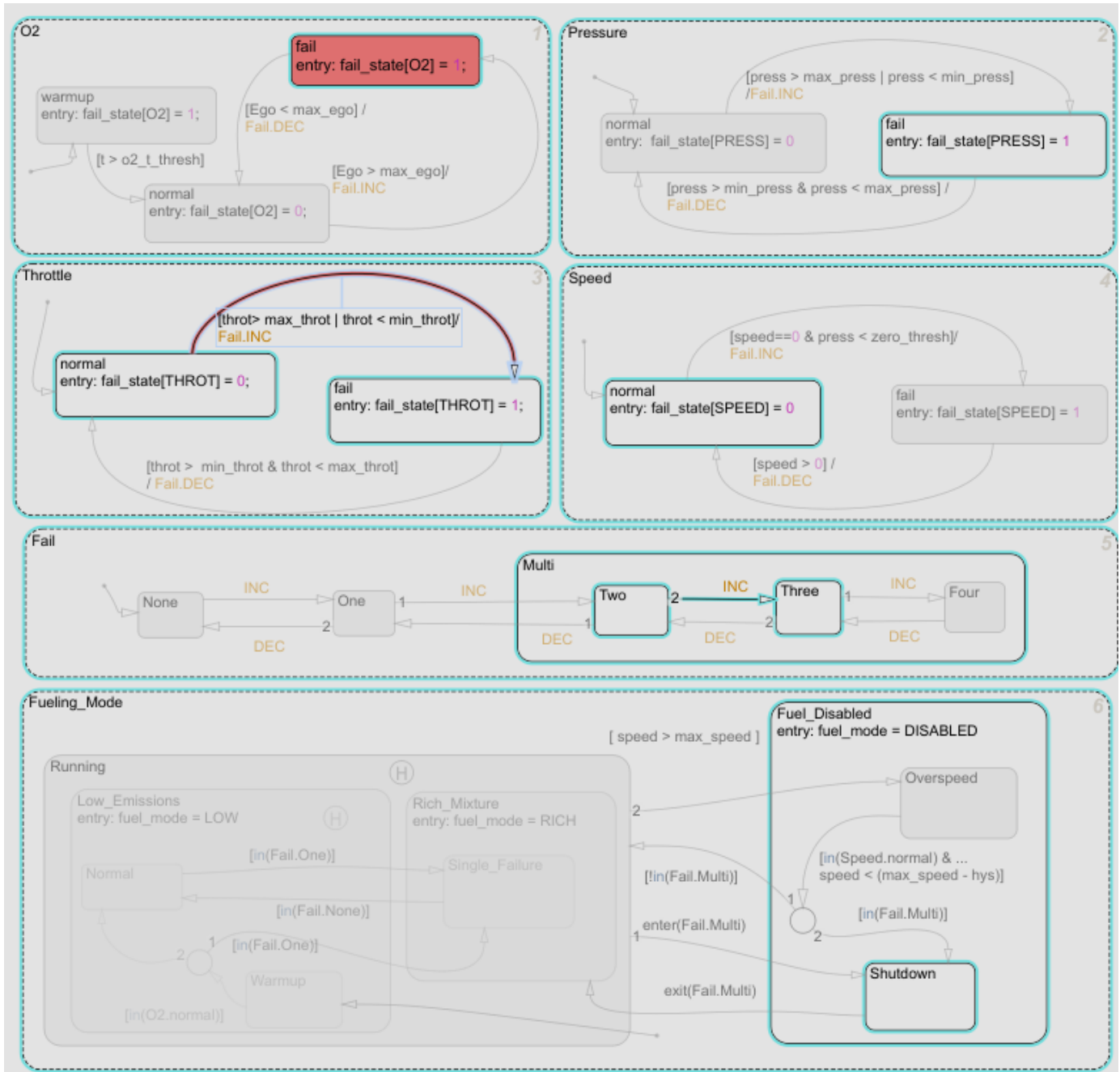
The Model Slice Manager is also updated to show the time interval in which the **O2 > fail** state is active and the `normal` to `fail` transition occurs for the `Throttle` chart:

Actual simulation time: 13.87 to 13.87 seconds : 1 active interval

Activity-Based Time Slicing Limitations and Considerations

For limitations and considerations of activity-based time slicing, see "Model Slicer Considerations and Limitations" on page 8-43.

Stateflow State and Transition Activity

For more information on Stateflow state and transition activity, see "Chart Simulation Semantics" (Stateflow), "Types of Chart Execution" (Stateflow), and "Syntax for States and Transitions" (Stateflow).

See Also

More About

- "Using Model Slicer with Stateflow" on page 8-49
- "Represent Operating Modes by Using States" (Stateflow)
- "Transition Between Operating Modes" (Stateflow)

Simplify a Standalone Model by Inlining Content

You can reduce file dependencies by inlining model content when you generate the sliced model. Inlining brings functional content into the sliced model and can eliminate model references, library links, and variant structures that are often not needed for model refinement or debugging.

If you want to disable inlining for certain block types, open the Model Slice Manager and click the options button ⚙. Select only the block types for which you want to inline content. For information on block-specific inlining behavior, see "Inline Content Options" on page 8-42.

This example demonstrates inlining content of a model referenced by a Model block.

1 Add the path to the example and open the model

```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
open_system('sldvSliceEngineDynamicsExample')
```

2 On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

3 In the model, right-click the MAP outport and select **Model Slicer > Add as Starting Point**. The path is highlighted through the Model block.
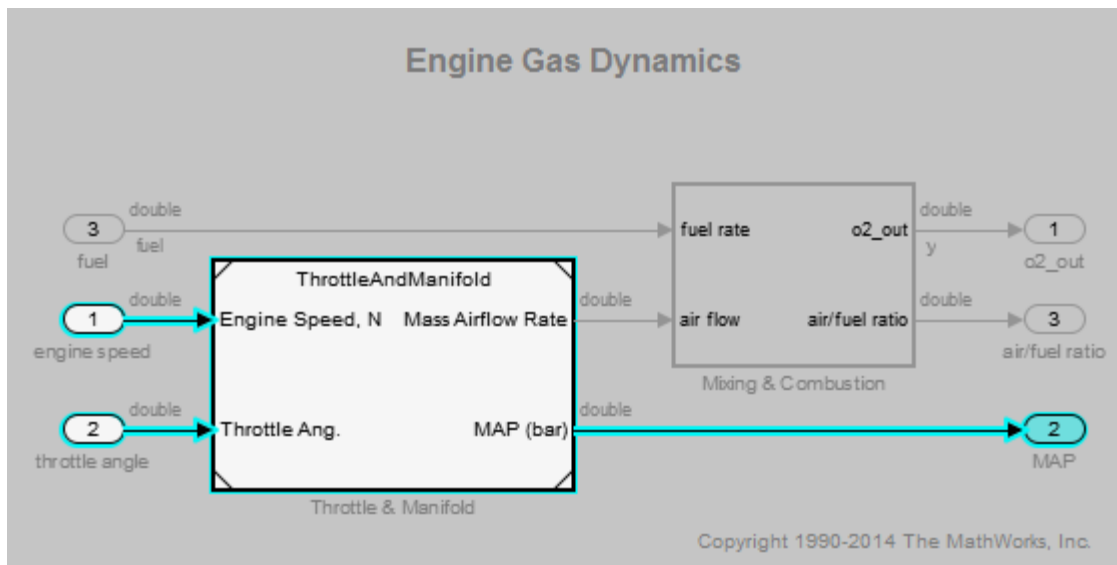
4 Create a sliced model from the highlight. In the Model Slice Manager, click the **Generate slice** button.

5 Enter a file name for the sliced model.

6 The sliced model contains the highlighted model content. The model reference is removed.

Engine Gas Dynamics
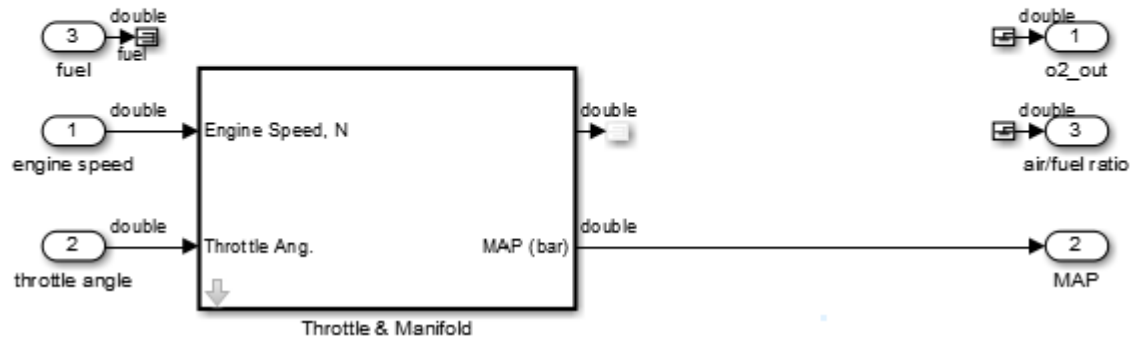
7 Click the arrow to look under the mask of the ThrottleAndManifold subsystem. The content from the referenced model is inlined into the model in the masked subsystem.
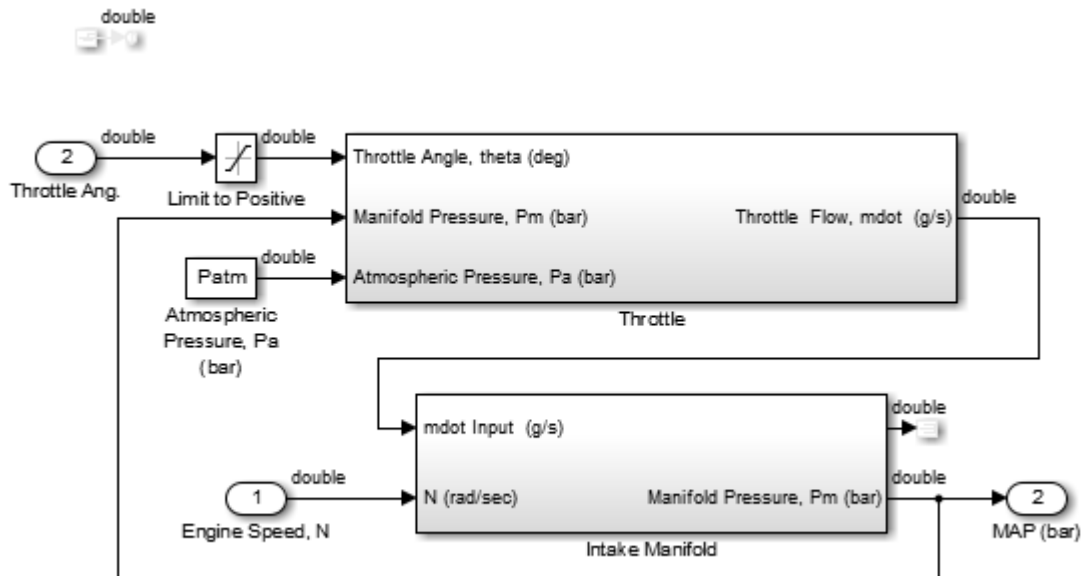
Workflow for Dependency Analysis

In this section...
"Dependency Analysis Workflow" on page 8-38
"Dependency Analysis Objectives" on page 8-38

Model analysis includes determining dependencies of blocks, signals, and model components. For example, to view blocks affecting a subsystem output, or trace a signal path through multiple switches and logic. Determining dependencies can be a lengthy process, particularly for large or complex models. Use Model Slicer as a simple way to understand functional dependencies in large or complex models. You can also use Model Slicer to create simplified standalone models that are easier to understand and analyze, yet retain their original context.

Dependency Analysis Workflow

The dependency analysis workflow identifies the area of interest in your model, generates a sliced model, revises the sliced model, and incorporates those revisions in the main model.
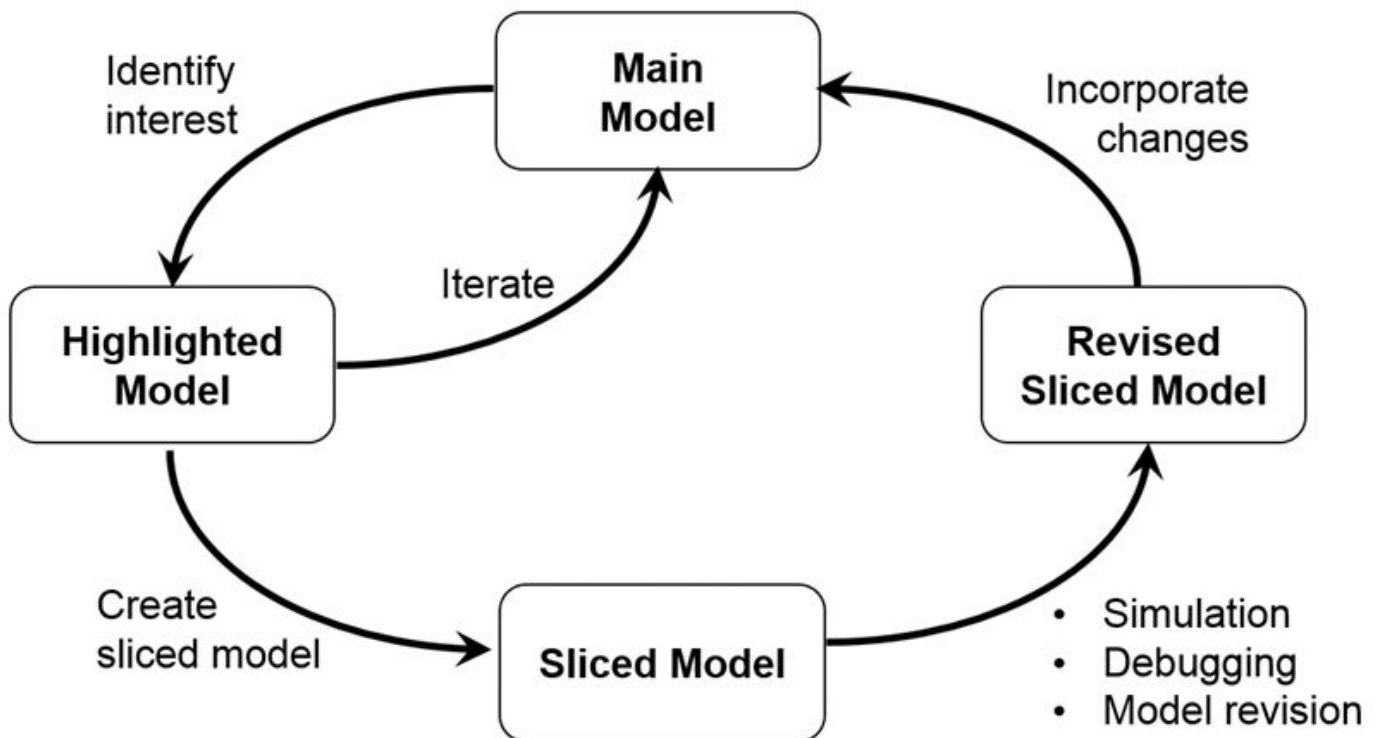
Dependency Analysis Objectives

To identify the area of interest in your model, determine objectives such as:

- What item or items are you analyzing? Analysis begins with at least one starting point.
- In what direction does the analysis propagate? The dependency analysis propagates upstream, downstream, or bidirectionally from the starting points.

- What model items or paths do you want to exclude from analysis?

- What paths do you want to constrain? If your model has switches, you can constrain the switch positions for analysis.

- Is your model a closed-loop system? If so, the highlighted portion of the model can include model dependencies from the feedback loop. Consider excluding blocks from the feedback loop to refine the highlighted portion of the model.

- Do you want to analyze static dependencies, or include simulation effects? Static analysis considers model dependencies for possible simulation paths. Simulation-based analysis highlights only paths active during simulation.

See Also

Related Examples

- "Highlight Functional Dependencies" on page 8-2
- "Refine Highlighted Model" on page 8-12
- "Create a Simplified Standalone Model" on page 8-28

Configure Model Highlight and Sliced Models

In this section...
"Model Slice Manager" on page 8-40
"Model Slicer Options" on page 8-40
"Storage Options" on page 8-40
"Refresh Highlighting Automatically" on page 8-41
"Sliced Model Options" on page 8-41
"Trivial Subsystems" on page 8-41
"Inline Content Options" on page 8-42

Model Slice Manager

Set the properties of your model highlight and standalone sliced model using the Model Slice Manager.

Click the toggle mode button to switch between model edit mode and model highlight mode.

If automatic highlighting is disabled in the slice settings, refresh the model highlight using the

refresh button . Refresh the highlight after changing the slice configuration.

Model Slicer Options

You can customize the slice behavior using the options dialog box, which is accessed with the options

button .

Storage Options

Changes you make to a model slice configuration are saved automatically. You can store the slice configuration in the model SLX file, or in an external SLMS file. Saving the configuration externally can be useful if your SLX file is restricted by a change control system.

To set the storage location, click the options button in the Model Slice Manager and set the location in the **Storage options** pane.

Settings

Store in <model_name>.slx

Saves the model slice configuration in your model's SLX file

Store in external file

Saves the model slice configuration in a separate SLMS file you specify by clicking the **Save As** button. The model slice configuration filename is shown in **File**.

Refresh Highlighting Automatically

Enables automatic refresh of a model highlight after changing the slice configuration.

Settings

on (default)

> Model highlighting refreshes automatically.

off

> Model highlighting must be refreshed manually. Click the refresh button in the Model Slice Manager to refresh the highlight.

Sliced Model Options

You can control what items are retained when you create a sliced model from a model highlight using the options in the **Sliced model options** pane.

Option	On (selected)	Off (cleared)
Retain signal observers	Signal observers, such as scopes, displays, and test condition blocks, are retained in the sliced model.	Signal observers are not retained in the sliced model (default).
Retain root-level inports and outports	Root-level ports are retained in the sliced model (default).	Root-level ports are not retained in the sliced model.
Expand trivial subsystems	Trivial subsystems are expanded in the sliced model and the subsystem boundary is removed (default).	Trivial subsystems are not expanded in the sliced model and the subsystem boundary is retained. See"Trivial Subsystems" on page 8-41.

Trivial Subsystems

If a subsystem has all of these characteristics, Model Slicer considers the subsystem trivial:

- If the subsystem is virtual, it contains three or fewer nonvirtual blocks.
- If the subsystem is atomic, it contains one or fewer nonvirtual blocks.
- The subsystem has two or fewer inports.
- The subsystem has two or fewer outports.
- The active inport or outport blocks of the subsystem have default block parameters.
- The system does not contain Goto Tag Visibility blocks.
- In the Block Properties dialog box, the subsystem **Priority** is empty.
- The data type override parameter (if applicable) is set to use local settings.

Note If you generate a sliced model which does not remove contents of a particular subsystem, the subsystem remains intact in the sliced model.

Inline Content Options

When you create a sliced model from a highlight, model items can be inlined into the sliced model. The **Inline content options** pane controls which model components are inlined in generating a sliced model.

Model Component	Inlining on (selected)	Inlining off (cleared)
Libraries	Model items inside sliced libraries are inlined in the sliced model and the library link is removed. (default)	Model items inside sliced libraries are not inlined in the sliced model and library link remains in place.
Masked subsystems	Model items inside sliced masked subsystems are inlined in the sliced model. (default) The mask is retained in the sliced model.	Model items inside sliced masked subsystems are not inlined in the sliced model and the mask is retained.
Model blocks	Model items are inlined to the sliced model from the model referenced by the Model block. The Model block is removed. (default) **Note** Model Slicer cannot inline model blocks that are not in `Normal` mode.	Model items are not inlined to the sliced model from the model referenced by the Model block. The Model block is retained.
Variants	Model items are inlined to the sliced model from the active variant. Variants are removed. (default)	Model items are not inlined to the sliced model from the variant. The variant is retained.

See Also

Related Examples

Model Slicer Considerations and Limitations

When you work with the Model Slicer, consider these behaviors and limitations:

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

Model Compilation

When you open Model Slice Manager, the model is compiled. To avoid a compilation error, before you open Model Slice Manager, make sure that the model is compilable.

Model Highlighting and Model Editing

When a slice highlight is active, you cannot edit the model. You can switch to model edit mode and preserve the highlights. When you switch back to slice mode, the slice configuration is recomputed and the highlight is updated.

Standalone Sliced Model Generation

Sliced model generation requires one or more starting points for highlighting your model. Sliced model generation is not supported for:

- Forward-propagating (including bidirectional) dependencies
- Constraints
- Exclusion points present in active highlight

Sliced model generation requires a writable working folder in MATLAB.

Sliced Model Considerations

When you generate a sliced model from a model highlight, simplifying your model can change simulation behavior or prevent the sliced model from compiling. For example:

- Model simplification can change the sorted execution order in a sliced model compared to the original model, which can affect the sliced model simulation behavior.
- If you generate a sliced model containing a bus, but not the source signal of that bus, the sliced model can contain unresolved bus elements.
- If you generate a sliced model that inlines a subset of the contents of a masked block, make sure that the subsystem contents resolve to the mask parameters. If the contents and mask do not resolve, it is possible that the sliced model does not compile.
- If the source model uses a bus signal, ensure that the sliced model signals are initialized correctly. Before you create the sliced model, consider including an explicit copy of the bus signal in the source model. For example, you can include a Signal Conversion block with the **Output** option set to `Signal Copy`.
- For solver step sizes set to `auto`, Simulink calculates the maximum time step in part based on the blocks in the model. If the sliced model removes blocks that affect the time step determination, the time step of the sliced model can differ from the source model. The time step difference can cause simulation differences. Consider setting step sizes explicitly to the same values calculated in the source model.

Port Attribute Considerations

You can use blocks that the Model Slicer removes during model simplification to determine compiled attributes, such as inherited sample times, signal dimensions, and data types. The Model Slicer can change sliced model port attributes during model simplification to resolve underspecified model port attributes. If the Model Slicer cannot resolve these inconsistencies, you can resolve some model port attribute inconsistencies by:

- Explicitly specifying attributes in the source model instead of relying on propagation rules.
- Including in the sliced model the blocks that are responsible for the attribute propagation in your source model. Before you slice the model, add these blocks as additional starting points in the source model highlighting.
- Not inlining the model blocks that are responsible for model port attributes into the sliced model. For more information on model items that you can inline into the sliced model, see "Inline Content Options" on page 8-42.

Because of the way Simulink handles model references, you cannot simultaneously compile two models that both contain a model reference to the same model. When you generate a sliced model, the Model Slicer enters the **Slicer Locked (for attribute checking)** mode if these conditions are true:

- The parent model contains a referenced model.
- The highlighted portion of the parent model contains the referenced model.
- The referenced model is not inlined in the sliced model due to one of the following

    - You choose not to inline model blocks in the **Inline content options** pane of the **Model Slicer options**.
    - The Model Slicer cannot inline the referenced model. For more information on model items that Model Slicer cannot inline, see "Inline Content Options" on page 8-42.

To continue refining the highlighted portion of the parent model, you must first activate the slice

highlight mode .

Simulation Time Window Considerations

Depending on the step size of your model and the values that you enter for the start time and stop time of the simulation time window, Model Slicer might alter the actual simulation start time and stop time.

- If you enter a stop or start time that falls between time steps for your model solver, the Model Slicer instead uses a stop or start time that matches the time step previous to the value that you entered. For more information on step sizes in Simulink, see "Compare Solvers".
- 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:

- 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 based slice generation of Function call triggered subsystems and model blocks.

Model Slicer Support Limitations for Simulation Stepper

When using Model Slicer with Simulation Stepper, the slice highlight after a Step Back may not be limited to a single step. The highlight can be influenced by the **Simulation Stepping Options** > **Interval between stored back steps**. For more information, see "Interval between stored back steps".

Model Slicer Support Limitations for Simulink Blocks

The table lists the Model Slicer support limitations for Simulink Blocks.

Block	Limitation
For Each Subsystem block	The simulation impact is ignored for blocks in a For Each subsystem. Therefore, applying a simulation time window returns the same dependency analysis result as a dependency analysis that does not use a simulation time window.
Function Caller block	Model Slicer does not support Function Caller blocks.
MATLAB Function block	Model Slicer assumes that any output depends on all inputs in the upstream direction and any input affects all outputs in the downstream direction.
Merge block	If you generate a slice by using a simulation time window, Merge blocks are removed in the standalone model if only a single path is exercised.
Model block	Model Slicer does not resolve data dependencies generated by global data store memory in Model blocks with **Simulation mode** set to `Accelerator`. Model Slicer does not support function-call root-level Inport blocks. For more information, see Export-Function Models. Model Slicer does not analyze the contents within a reference to a "Reference Protected Models from Third Parties". When you slice a model that contains a protected model reference, the Model Slicer includes the entire model reference in the sliced model.
Resettable Subsystem block	Model Slicer does not support Resettable Subsystem blocks.
S-function block	Model Slicer assumes that any output depends on all inputs in the upstream direction and any input affects all outputs in the downstream direction. Model Slicer does not determine dependencies that result from an S-function block accessing model information dependent on a simulation time window.

Model Slicer Support Limitations for Stateflow

- When you highlight models containing a Stateflow chart or state transition table, Model Slicer assumes that any output from the Chart block or State Transition Table block depends on all inputs to the Chart block or State Transition Table block.

- When you slice a model with a Stateflow chart or a state transition table, Model Slicer does not simplify the chart or table. The chart or table is included in its entirety in the sliced model.

- If you do not "Define a Simulation Time Window" on page 8-12 when you highlight functional dependencies in a Stateflow chart or state transition table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.

- When you highlight functional dependencies in a Stateflow chart or state transition table for a defined simulation time window, Model Slicer does not highlight only the states and transitions that affect the selected starting point. Instead, the Model Slicer highlights elements that are active in the time window that you specify.

- The Model Slicer does not determine dependencies between Stateflow graphical functions and function calls in other Stateflow charts.

- Graphical functions and their contents that were not active during the selected time window can potentially remain highlighted in some cases.

- Entry into states that are preempted due to events can potentially remain highlighted in some cases. For example, after a parent state is entered, an event action can exit the state and preempt entry into the child state. In such a case, the Model Slicer highlights the entry into the child state.

- The Model Slicer does not support:

  - MATLAB Function blocks
  - Simulink functions
  - Truth Table blocks
  - Machine-parented data or events in Stateflow.

  .

Activity-Based Time Slicing Considerations for Stateflow

As measured by the 'Executed Substate' decision coverage, state activity refers to these during/exit actions:

- Entry into a state does not constitute activity.
- The active time interval for a state or transition includes the moment in which the selected state exits and the subsequent state is entered.
- Indirect exits from a state or transition do not constitute activity. For example, if a state C exits because its parent state P exits, state C is not considered active.

For more information on decision coverage for Stateflow charts, see "Decision Coverage for Stateflow Charts" (Simulink Coverage).

When you "Highlight Active Time Intervals by Using Activity-Based Time Slicing" on page 8-29, you can select states and transitions only as activity constraints. You cannot select these Stateflow objects as constraints:

- Parallel states
- Transitions without conditions, such as unlabeled transitions which do not receive decision coverage
- States or transitions within library-linked charts
- XOR states without siblings. For example, if a state P has only one child state C, you cannot select state C as an activity constraints because state P does not receive decision coverage for the executed substate

See Also
"Algebraic Loop Concepts" | "Solver Pane"

Using Model Slicer with Stateflow

In this section...
"Model Slicer Highlighting Behavior for Stateflow Elements" on page 8-49
"Using Model Slicer with Stateflow State Transition Tables" on page 8-49
"Support Limitations for Using Model Slicer with Stateflow" on page 8-49

You can use Model Slicer highlighting to visually verify the logic in your Stateflow charts or tables. After you "Define a Simulation Time Window" on page 8-12, you use Model Slicer to highlight and slice Stateflow elements that are active within the selected time window.

Note If you do not "Define a Simulation Time Window" on page 8-12 when you highlight functional dependencies in a Stateflow chart or table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.

Model Slicer Highlighting Behavior for Stateflow Elements

Model Slicer highlights a Stateflow element if it was executed in the specified time window. Some examples include:

- A chart, if it is activated in the specified a time window.

- A state, if its entry, exit, or during actions are executed in the specified a time window.

- A parent state, if its child state is highlighted in the specified a time window.

- A transition, if it is taken in the specified time window, such as inner, outer, and default. If the conditions of a transition are evaluated, but the transition is not taken, Model Slicer does not highlight the transition.

Using Model Slicer with Stateflow State Transition Tables

Model Slicer does not directly highlight the contents of Stateflow state transition tables. To view highlighted functional dependencies in a state transition table, you must view the auto-generated diagram for the state transition table. For instructions on how to view the auto-generated diagram for the state transition table, see "Generate Diagrams from State Transition Tables" (Stateflow).

Support Limitations for Using Model Slicer with Stateflow

For support limitations when you use Model Slicer with Stateflow, see "Model Slicer Support Limitations for Stateflow" on page 8-47.

See Also

More About

- "Highlight Functional Dependencies" on page 8-2
- "Refine Highlighted Model" on page 8-12
- "Chart Simulation Semantics" (Stateflow)

Isolating Dependencies of an Actuator Subsystem

This example demonstrates highlighting model items that a subsystem depends on. It also demonstrates generating a standalone model slice from the model highlight.

In this section...
"Choose Starting Points and Direction" on page 8-51
"View Precedents and Generate Model Slice" on page 8-52

Choose Starting Points and Direction

1   Open the `f14` example model.

     `f14`

2   On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
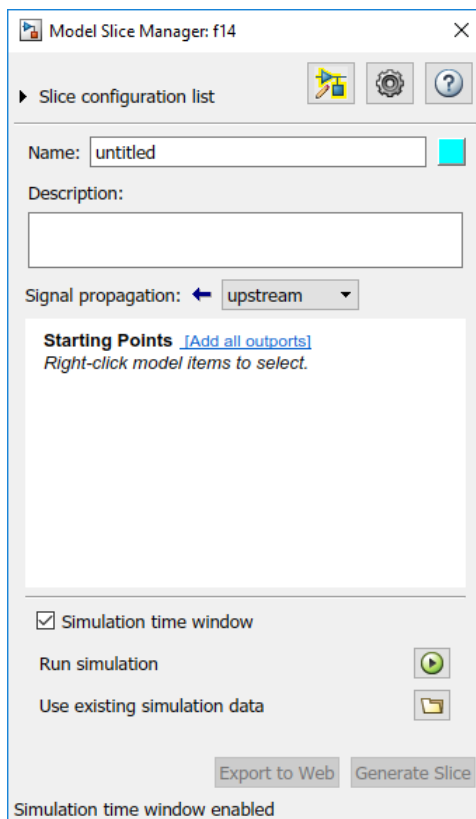
3   In the Model Slice Manager, click the arrow to expand the **Slice configuration list** list. Set the slice properties:

- **Name**: `Actuator_slice`

- To the right of **Name**, click the colored square to set the highlight color. Choose magenta from the palette.

- **Signal Propagation**: `upstream`.

4 Add the `Actuator Model` subsystem as a starting point. In the model, right-click the `Actuator Model` subsystem and select **Model Slicer > Add as Starting Point**.
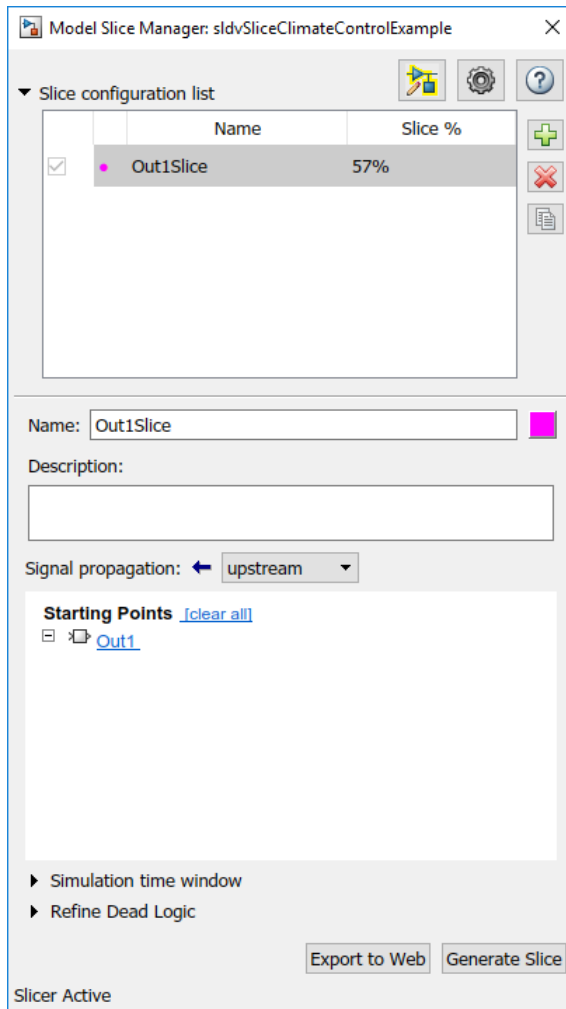
View Precedents and Generate Model Slice

1 The model highlights the upstream dependencies of the `Actuator Model` subsystem.

F-14 Flight Control

Copyright 1990-2014 The MathWorks, Inc.
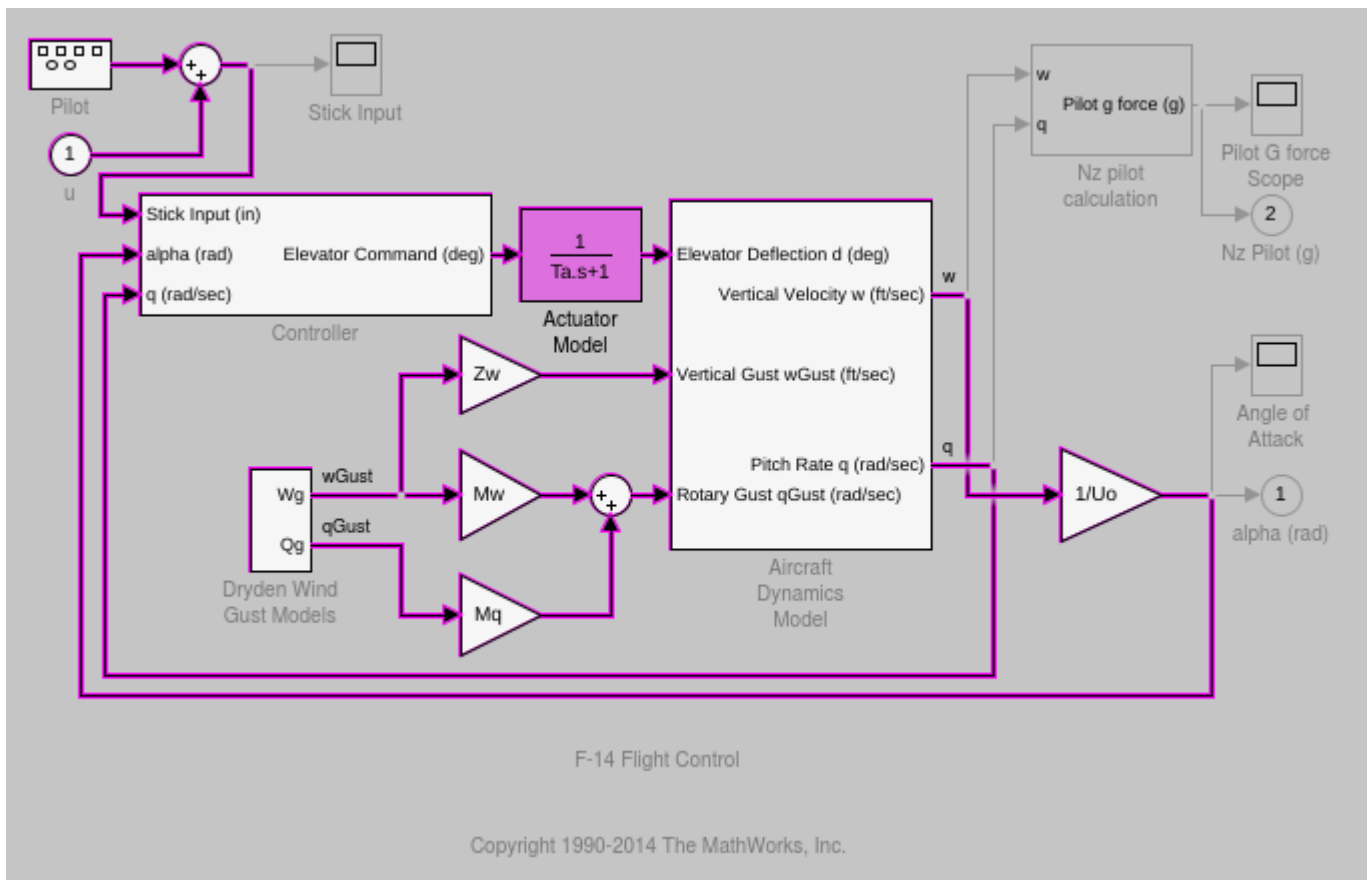
Trace the following dependency path. `Aircraft Dynamics Model` is highlighted via the `Pitch Rate q` signal, which is an input to `Controller`, the output of which feeds `Actuator Model`.

2 Generate a standalone model containing the highlighted model items:

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

b In the **Select File to Write** dialog box, select the save location and enter `actuator_slice_model`.

c Click **Save**.

3 The sliced model contains the highlighted model items.
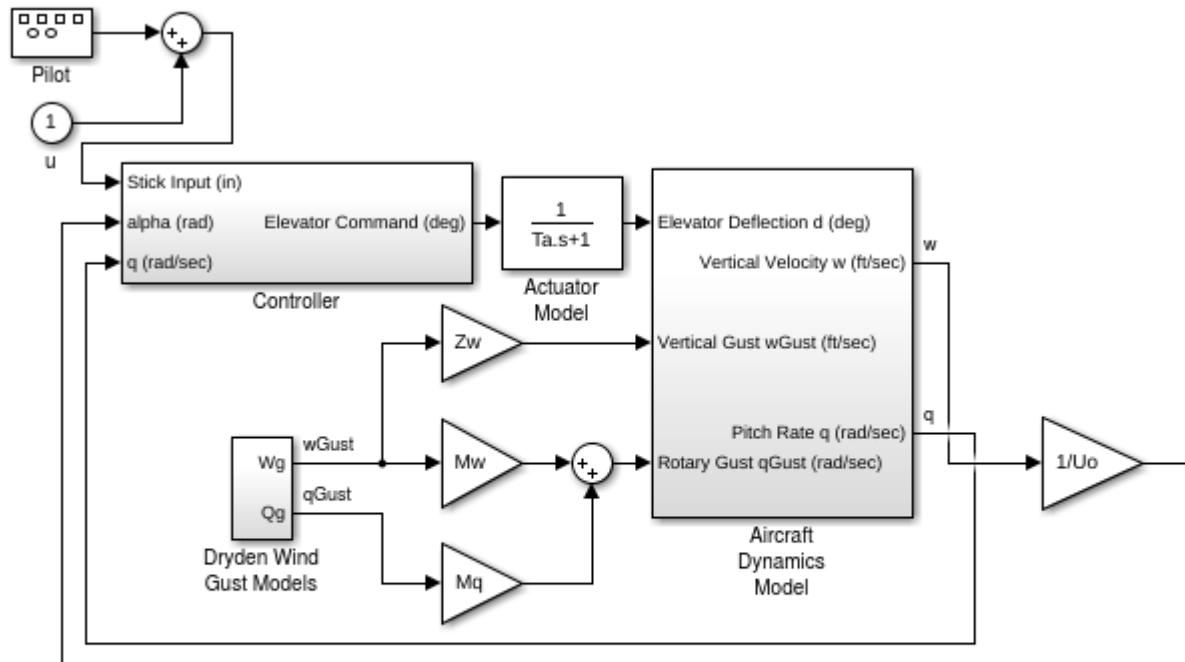
F-14 Flight Control

Copyright 1990-2014 The MathWorks, Inc.

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

Isolate Model Components for Functional Testing

You can create a standalone model for the model designed using "Design Model Architecture". The model slice isolates the model components and relevant signals for debugging and refinement.

Isolate Subsystems for Functional Testing

To debug and refine a subsystem of your model, create a standalone model. The standalone model isolates the subsystem and relevant signals. You can observe the subsystem behavior without simulating the entire source model.

Note You cannot slice virtual subsystems. To isolate a virtual subsystem, first convert it to an atomic subsystem.

Isolate a Subsystem with Simulation-Based Inputs

To observe the simulation behavior of a subsystem, include logged signal inputs in the standalone model. When you configure the model slice, specify a simulation time window. For large models, observing subsystem behavior in a separate model can save time compared to compiling and running the entire source model.

This example shows how to include simulation effects for the Controller subsystem of a cruise control system.
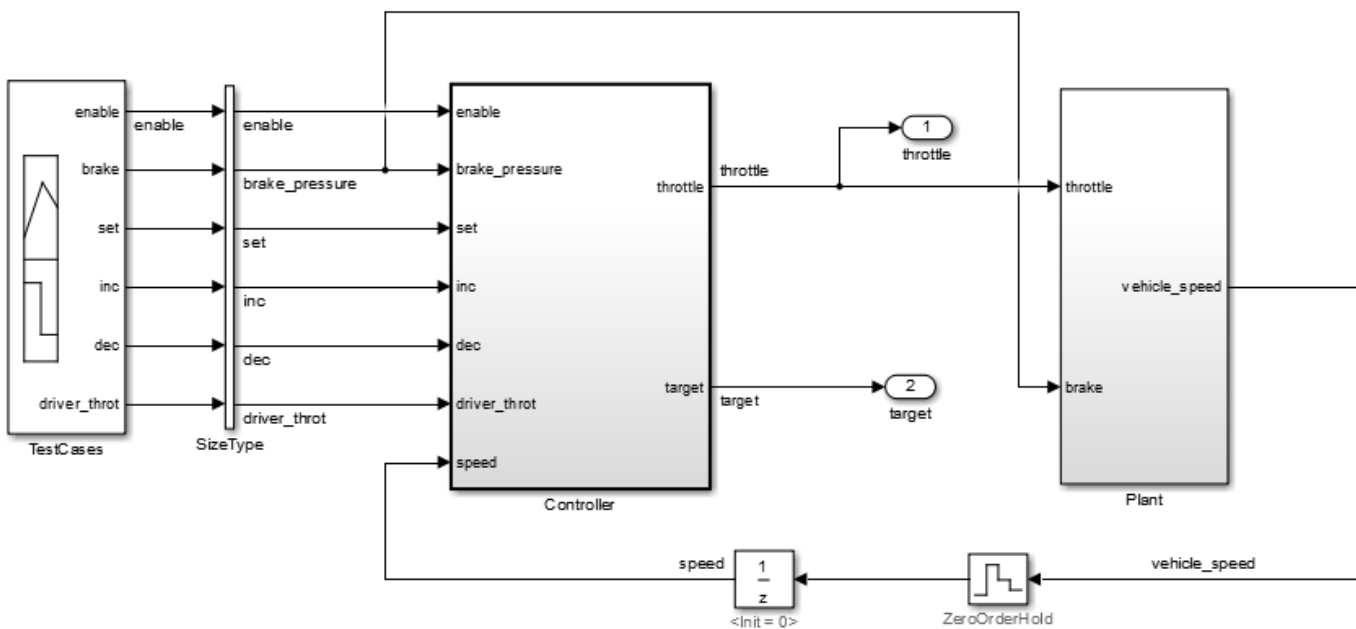
1   To open the Model Slice Manager, on the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

2 To select the starting point for dependency analysis, right-click a block, signal, or a port, and select **Model Slicer > Add as Starting point**.

3 To isolate the subsystem in the sliced model, right-click the subsystem, and select **Model Slicer > Slice component**.

In the example model, selecting **Slice component** for the Controller subsystem limits the dependency analysis to the path between the starting point (the throttle outport) and the Controller subsystem.
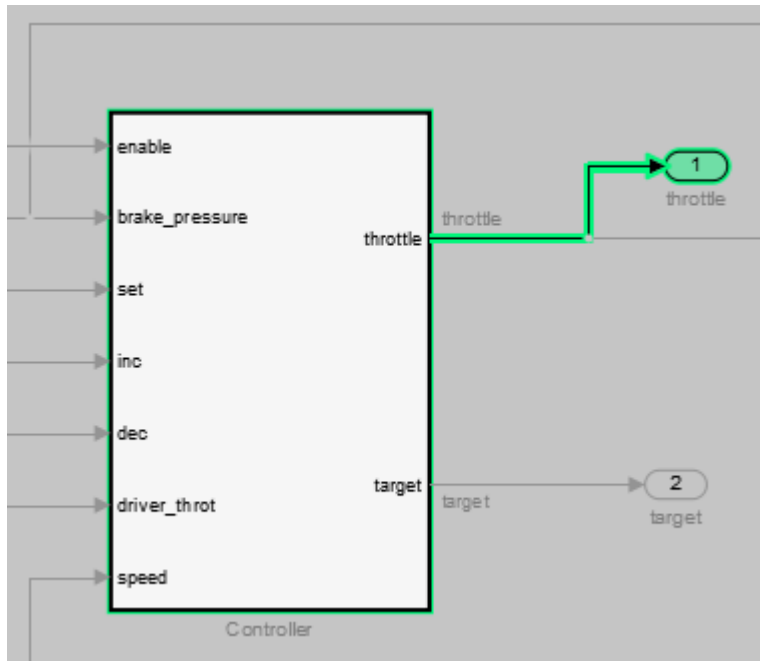
4 To specify the simulation time window:

a In the Model Slice Manager, select **Simulation time window**.

b Click the run simulation button .

c Enter the simulation stop time, and click **OK**.
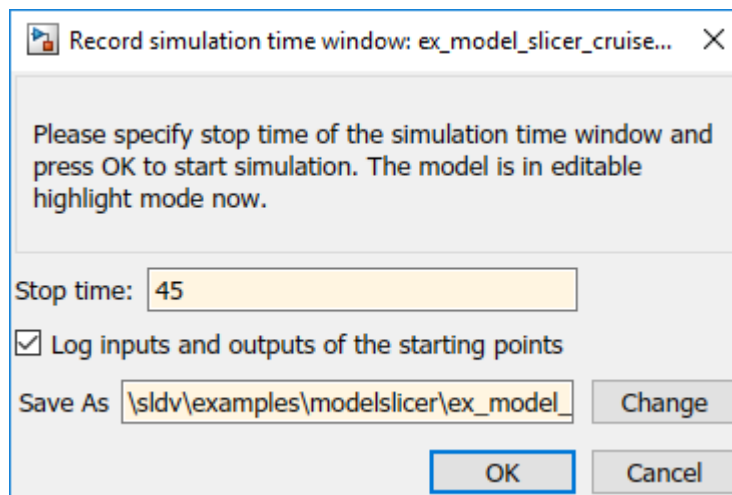
The Model slicer analyzes the model dependencies for the simulation interval.

5 To extract the subsystem and logged signals, click **Generate slice**. Enter a file name for the sliced model.

Based on the dependency analysis, a Signal Builder block supplies the signal inputs to the subsystem.

In the sliced model shown, the sliced model Signal Builder block contains one test case representing the signal inputs to the Controller subsystem for simulation time 0–45 seconds.

Isolate Referenced Model for Functional Testing

To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

This example shows how to slice the referenced model controller of a fault-tolerant fuel control system for functional testing. To create a simplified open-loop referenced model for debugging and refinement, you generate a slice of the referenced controller.

Step 1: Open the Model

The fault-tolerant fuel control system model contains a referenced model controller `fuel_rate_control`.

`open_system('sldvSlicerdemo_fuelsys');`

Fault-Tolerant Fuel Control System

Copyright 1990-2017 The MathWorks, Inc.

Step 2: Slice the Referenced Model

To analyze the `fuel_rate_control` referenced model, you slice it to create a standalone open-loop model. To open the Model Slice Manager, select **Apps > Model Verification, Validation, and Test > Model Slicer**, or right-click the `fuel_rate_control` model and select **Model Slicer > Slice component**. When you open the Model Slice Manager, the Model Slicer compiles the model. You then configure the model slice properties.

Note: The simulation mode of the `sldvSlicerdemo_fuelsys` model is `Accelerator` mode. When you slice the referenced model, the software configures the simulation mode to `Normal` mode and sets it back to its original simulation mode while exiting the Model Slicer.

Step 3: Select Starting Point

Open the `fuel_rate_control` model, right-click the `fuel-rate` port, and select **Model Slicer > Add as starting point**. The Model Slicer highlights the upstream constructs that affect the `fuel_rate`.

Step 4: Generate Slice

a. In the Model Slice Manager dialog box, select the **Simulation time window**.

b. Click **Run simulation**.

c. For the **Stop time**, enter 20. Click **OK**.

d. Click **Generate Slice**. The software simulates the sliced referenced model by using the inputs of the close-loop sldvSlicerdemo_fuelsys model.

For the sliced model, in the Signal Builder window, one test case is displayed that represents the signals input to the referenced model for simulation time 0–20 seconds.

See Also

"Model Slicer Considerations and Limitations" on page 8-43 | "Highlight Functional Dependencies" on page 8-2

Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results

When you run simulation or refine dead logic, Model Slicer saves your simulation results at the default location `<current_folder>\modelslicer\<model_name>\<model_name>.slslicex`. For large or complex models, the simulation time can be lengthy. To refine the highlighted slice, you can use the existing Model Slicer simulation data or dead logic results.

If you want to highlight functional dependencies in the model again at another time, you can use the existing `.slslicex` simulation time window data without needing to resimulate the model. Model Slicer then uses the existing simulation data to highlight the model.

1   Open the Simulink model.

2   To open the Model Slice Manager, On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

3   Select **Simulation time window**.

4   Click **Use existing simulation data** 📁.

5   Navigate to the existing `.slslicex` data and click **Open**.

To refine the dead logic for dependency analysis, you can import the existing Simulink Design Verifier data file or use the existing `.slslicex` dead logic results. For more information see, "Dead Logic Detection" (Simulink Design Verifier) and "Simulink Design Verifier Data Files" (Simulink Design Verifier).

1   In Model Slice Manager, select **Refine Dead Logic** and click **Get Dead Logic Data**.

2   To import the Simulink Design Verifier data file, click **Browse for SLDV data file** 📁.

 To load the existing dead logic results, click **Browse for existing dead logic results** 📁.

3   Navigate to the existing data and click **Open**.
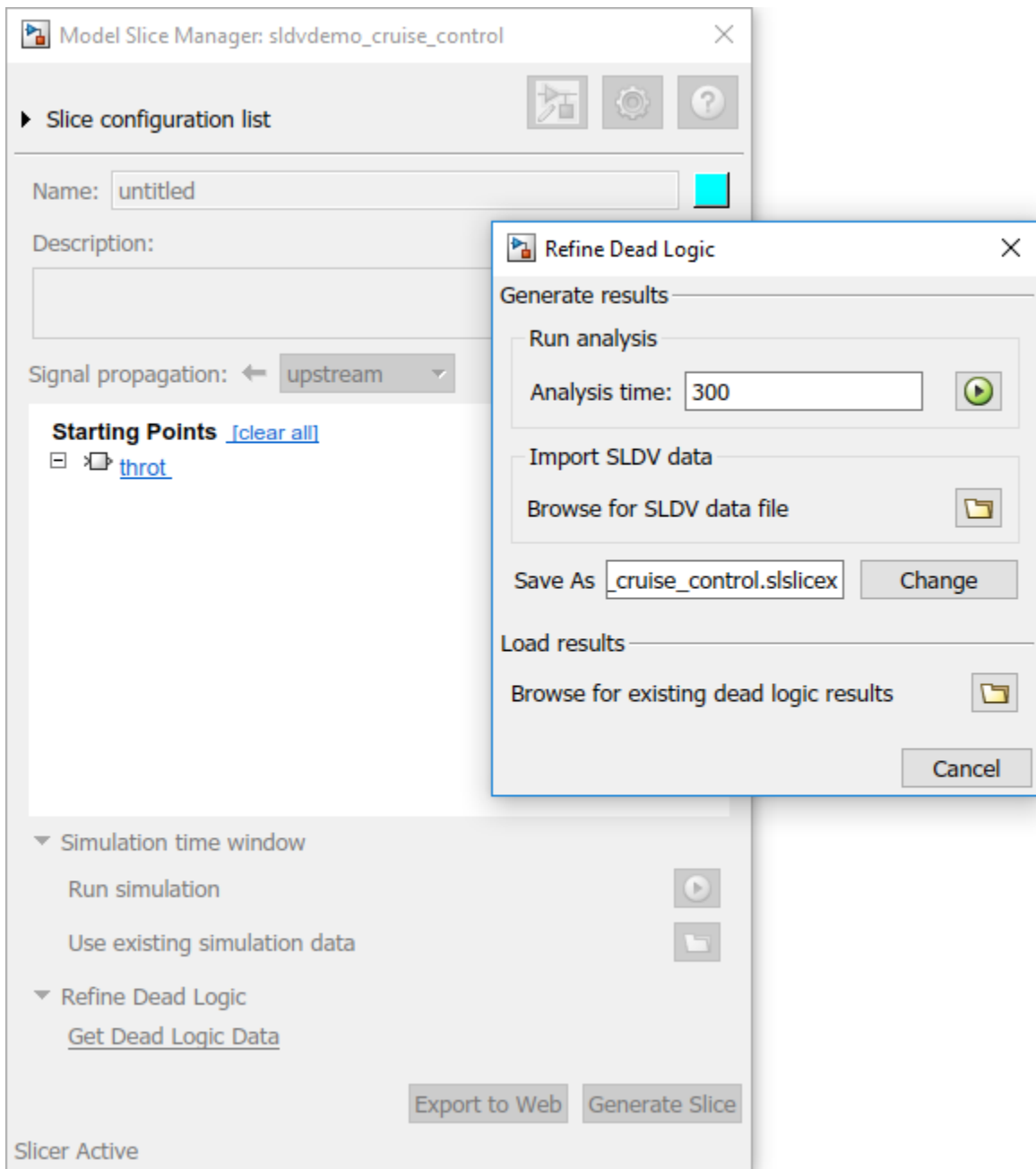
See Also

More About

- "Highlight Functional Dependencies" on page 8-2
- "Configure Model Highlight and Sliced Models" on page 8-40
- "Refine Dead Logic for Dependency Analysis" on page 8-22

Simplification of Variant Systems

In this section...
"Use the Variant Reducer to Simplify Variant Systems" on page 8-65
"Use Model Slicer to Simplify Variant Systems" on page 8-65

If your model contains "Variant Systems", 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, you can reduce the model from the Variant Manager:

1 Open a model containing at least one valid variant configuration.

2 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".

Use Model Slicer to Simplify Variant Systems

After you Add and Validate Variant Configurations, you can use Model Slicer to create a simplified, standalone model containing only the active variant configuration. When you "Highlight Functional Dependencies" on page 8-2 in a model containing variant systems, only active variant choices are highlighted. When you "Create a Simplified Standalone Model" on page 8-28 from a model highlight that includes variant systems, Model Slicer removes the variant systems and replaces them with the active variant configurations.

For instructions on how to change the active variant configuration and how to set default variant choices, see "Working with Variant Choices".

See Also

More About

- "Create a Simple Variant Model"
- "Define, Configure, and Activate Variant Choices in a Variant Subsystem Block"
- "Introduction to Variant Controls"
- "Reduce Models Containing Variant Blocks"

Programmatically Resolve Unexpected Behavior in a Model with Model Slicer

In this example, you evaluate a Simulink® model, detect unexpected behavior, and use Model Slicer to programmatically isolate and resolve the unexpected behavior. When you plan to reuse your API commands and extend their use to other models, a programmatic approach is useful.

Prerequisites

Be familiar with the behavior and purpose of Model Slicer and the functionality of the Model Slicer API. "Highlight Functional Dependencies" on page 8-2 outlines how to use Model Slicer user interface to explore models. The `slslicer`, `slsliceoptions`, 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')
```
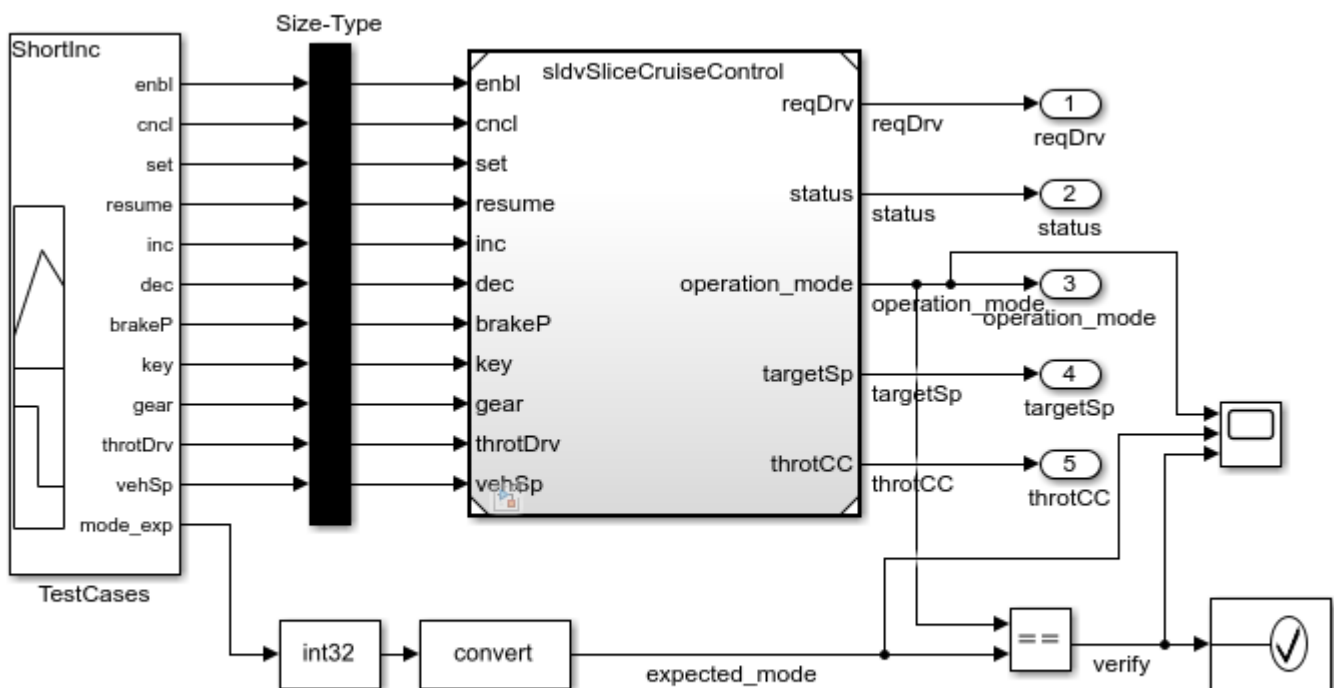

Copyright 2015 The MathWorks, Inc.

Note: The Assertion block is set to **Stop simulation when assertion fails** when the actual operation mode is not the same as the expected operation mode.

The TestCases block contains several test inputs for sldvSliceCruiseControl.

2. In the TestCases Signal Builder click the **Run all** button to run all of the included test cases. You receive an error during the ResumeWO test case. The Assertion block halted simulation at 27 seconds, because the actual operation mode was not the same as the expected operation mode. Click **OK** to close this error message.

3. In the sldvSliceCruiseControlHarness model, double-click the Assertion block, clear **Enable assertion**, and click **OK**.

```
set_param('sldvSliceCruiseControlHarness/Assertion','Enabled','off')
```

4. Set the **Active Group** of the *TestCases* Signal Builder block to the test case containing the error and run the simulation again.

```
signalbuilder('sldvSliceCruiseControlHarness/TestCases', 'ACTIVEGROUP', 12)
sim('sldvSliceCruiseControlHarness')
```

The Scope block in the model contains three signals:

- *operation_mode* - displays the actual operation mode of the subsystem.
- *expected_mode* - displays the expected operation mode of the subsystem that the test case provides.
- *verify* - displays a Boolean value comparing the operation mode and the expected mode.

The scope shows a disparity between the expected operation mode and the actual operation mode beginning at time **27**. Now that you know the outport displaying the unexpected behavior and the time window containing the unexpected behavior, use Model Slicer to isolate and analyze the unexpected behavior.

Isolate the Area of the Model Responsible for Unexpected Behavior

1. Create a Model Slicer configuration object for the model using `slslicer`. The Command Window displays the slice properties for this Model Slicer configuration.

```
obj = slslicer('sldvSliceCruiseControlHarness')


obj =

  SLSlicer with properties:

        Configuration: [1x1 SLSlicerAPI.SLSlicerConfig]
         ActiveConfig: 1
      DisplayedConfig: []
        StorageOptions: [1x1 struct]
       AnalysisOptions: [1x1 struct]
          SliceOptions: [1x1 struct]
         InlineOptions: [1x1 struct]

   Contents of active configuration:
                 Name: 'untitled'
          Description: ''
```

```
                Color: [0 1 1]
    SignalPropagation: 'upstream'
        StartingPoint: [1x0 struct]
       ExclusionPoint: [1x0 struct]
           Constraint: [1x0 struct]
       SliceComponent: [1x0 struct]
        UseTimeWindow: 0
         CoverageFile: ''
         UseDeadLogic: 0
        DeadLogicFile: ''
```

2. Activate the slice highlighting mode of Model Slicer to compile the model and prepare it for dependency analysis.

```
activate(obj)
```

```
Consider turning on <a href="matlab:helpview(fullfile(docroot, 'simulink/ug/fast-restart-workflow
```

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')
```

```
slicedModel =

    'sldvSliceCruiseControlHarness_sliced'
```
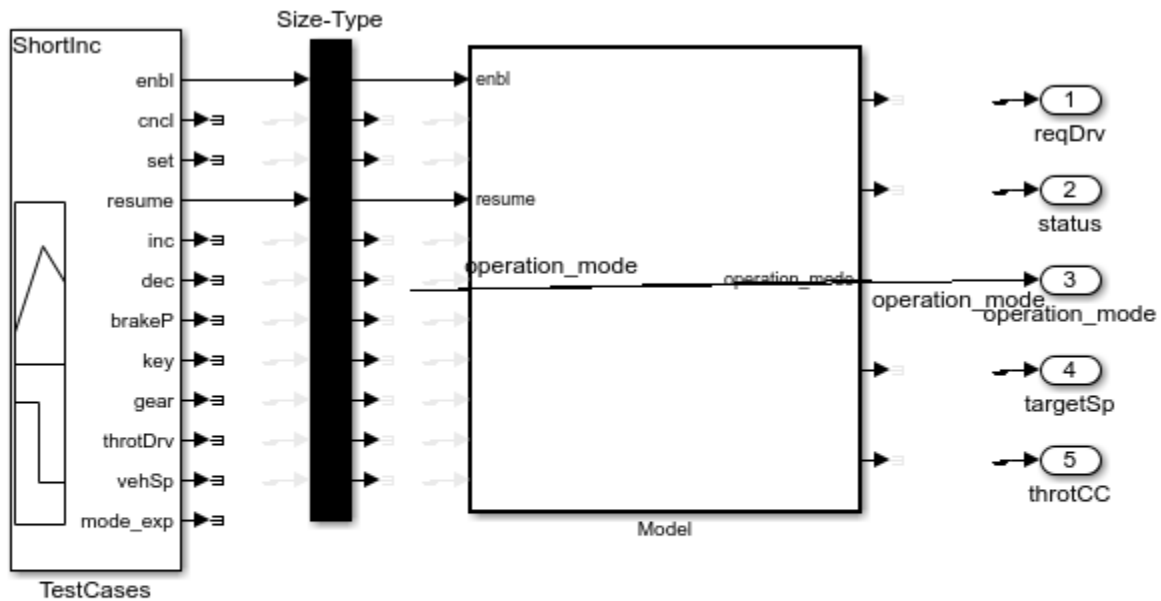
The sliced model **sldvSliceCruiseControlHarness_sliced** now contains a simplified version of the source model **sldvSliceCruiseControlHarness**. The simplified standalone model contains only those parts of the model that are upstream of the specified starting point and active during the time window 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,
```
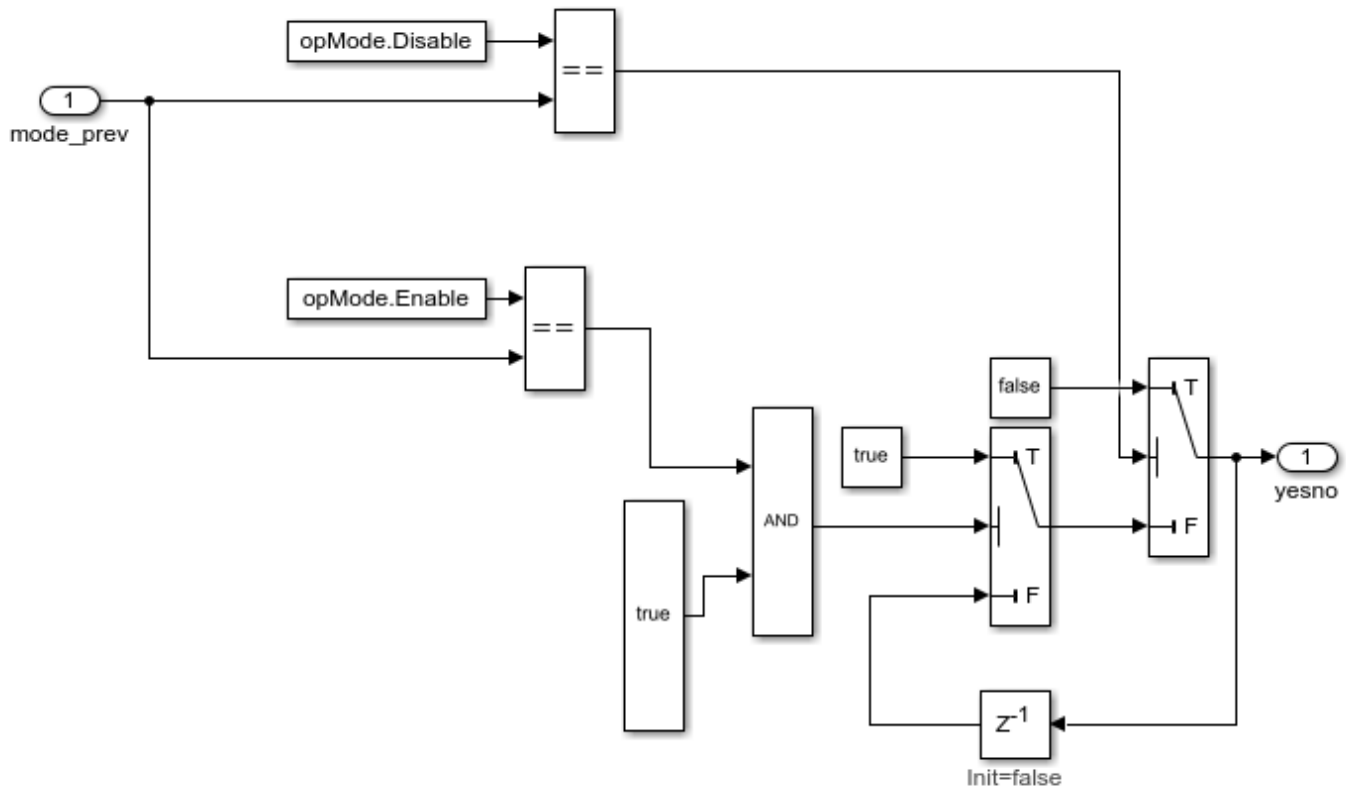
The *AND* Logical Operator block in this subsystem has a truncated *true* constant attached to its second input port. This *true* constant indicates that the second input port is always *true* during the restricted time window for this sliced model, causing the cruise control system not to enter the "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
hilite_system(h)

h =

   1.2150e+03
```
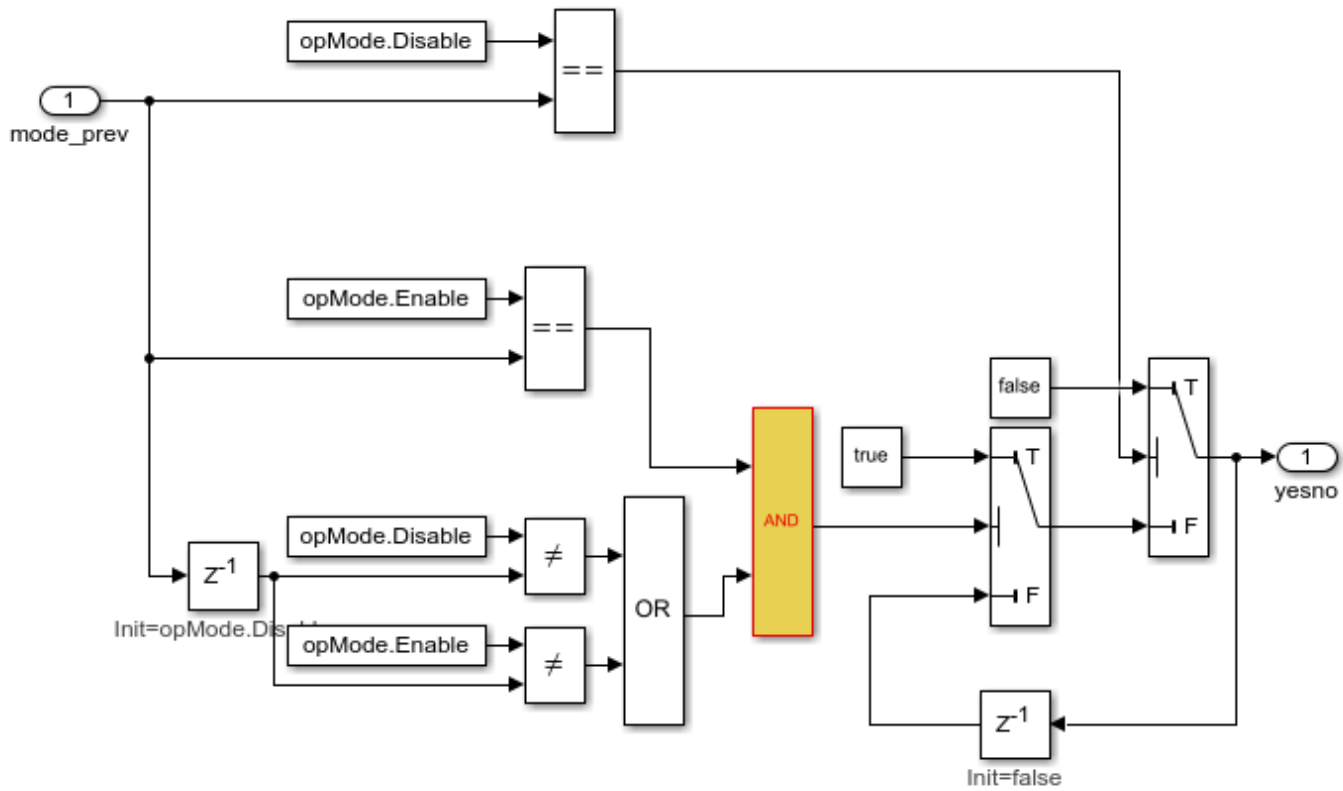
The *OR* Logical Operator block in this subsystem is always *true* in the current configuration. Changing the *OR* Logical Operator block to an *AND* Logical Operator block rectifies this error.

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')
```
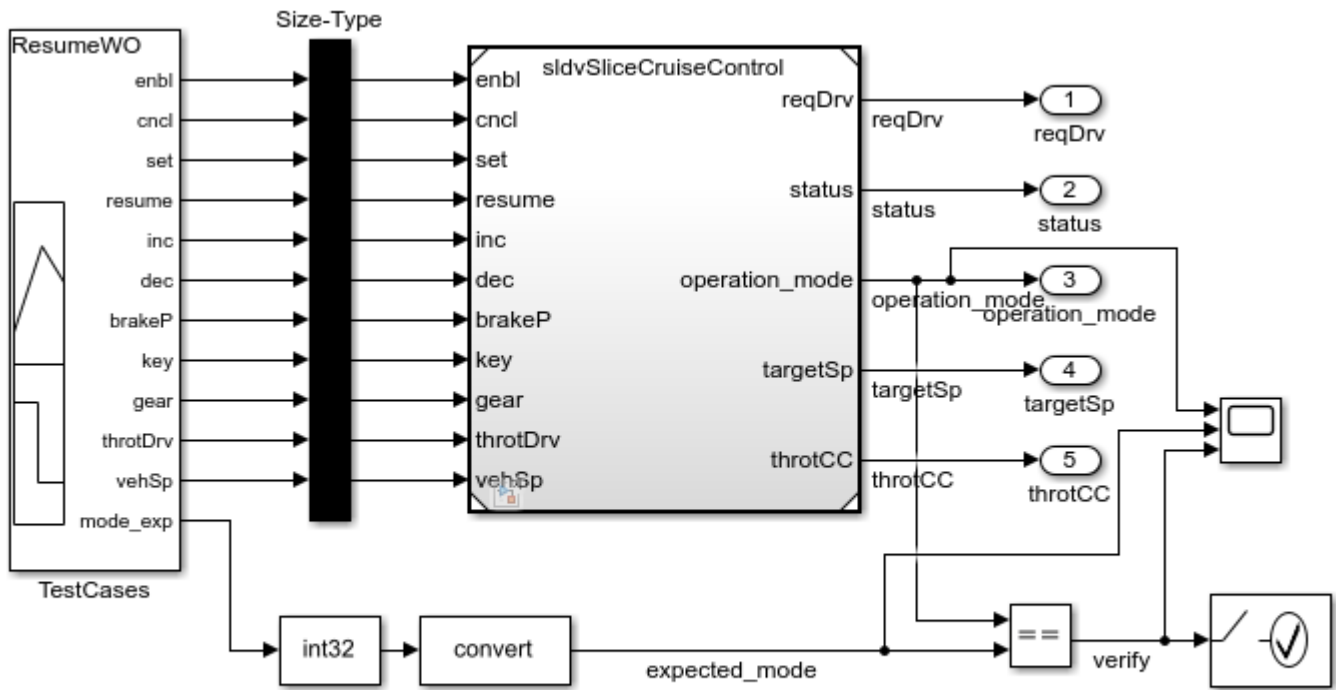
Copyright 2015 The MathWorks, Inc.
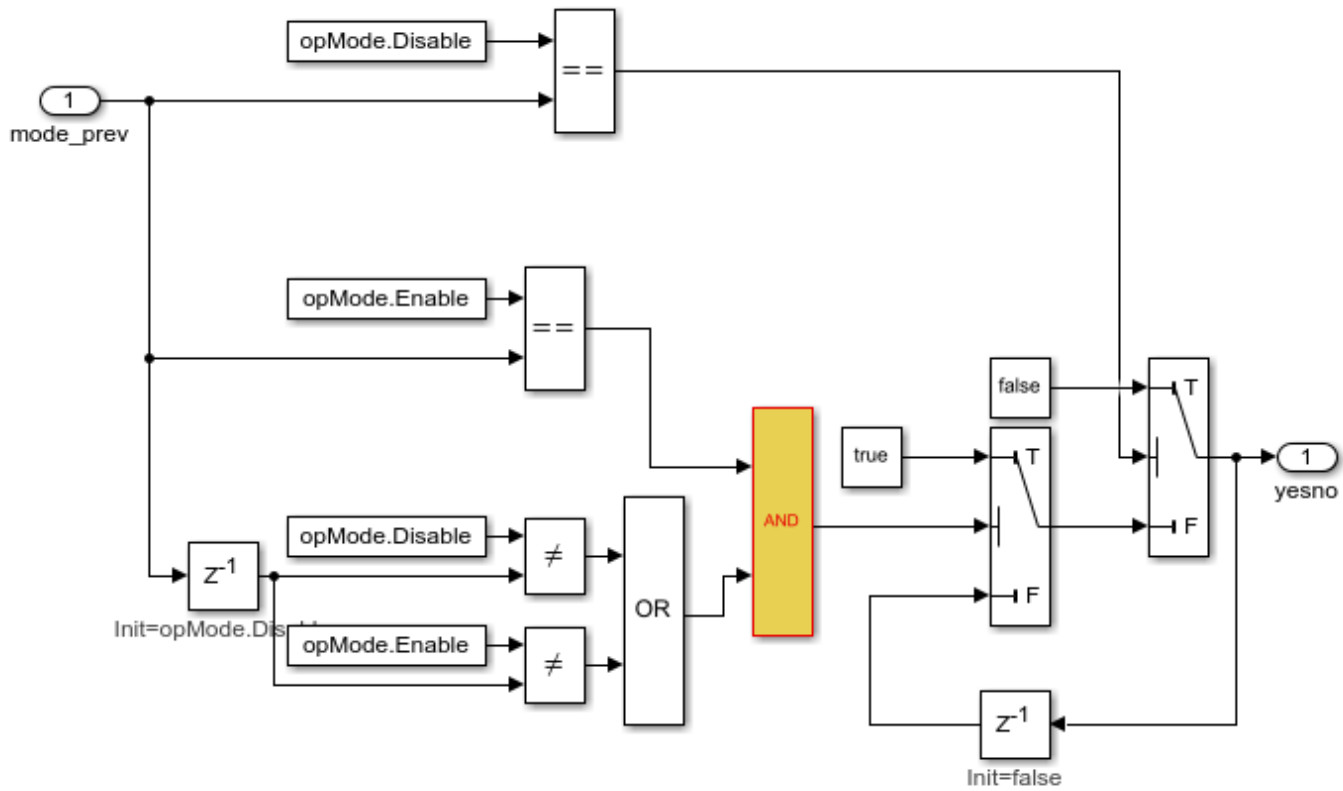
5. Update the model reference in the test harness to refer to the newly saved model.

```
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/Log
```

7. Simulate the test harness for 45 seconds with the fixed model to confirm the corrected behavior.

```
sim('sldvSliceCruiseControlHarness_fixed')
```

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
```

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".
- Select simulation time window — Define simulation time window by using data cursors in the graphical plot or by defining the **Start** and **Stop** time in the Inspector.
- Highlight — Compute a slice for the defined simulation time window. See "Highlight Functional Dependencies" on page 8-2.

Investigate Highlighted Model Slice by Using Model Slicer Data Inspector

This example shows how to investigate and refine the highlighted model slice by using the Model Slicer Data Inspector.

In the fault-tolerant fuel control system, the `control_logic` controls the fueling mode of the engine. In this example, you slice the `fuel_rate_control` referenced model. Then, investigate the effect of `fuel_rate_ratio` on the `Fueling_mode` of the engine. For more information, see "Modeling a Fault-Tolerant Fuel Control System".

Step 1: Start the Model Slice Manager

To start the Model Slice Manager, open the `fuel_rate_control` model, and select **Apps > Model Verification, Validation, and Test > Model Slicer**.

```
open_system('sldvSlicerdemo_fuelsys');
```

Fault-Tolerant Fuel Control System

Copyright 1990-2017 The MathWorks, Inc.

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**.

Step 3: Inspect signals

To open the Model Slicer Data Inspector, click **Inspect Signals**.

The logged input and output signals appear in the Model Slicer Data Inspector. When you open the Model Slicer Data Inspector, Model Slicer saves the existing Simulation Data Inspector session as MLDATX-file in the current working directory.

You can select the time window by dragging the data cursors to a specific location or by specifying the **Start** and **Stop** time in the navigation pane. To highlight the model for the defined simulation time window, Click **Highlight**.

To investigate the `Fueling_mode`, open the `control_logic` Stateflow™ chart, available in the `fuel_rate_control` referenced model. Select the time window for 13–15 seconds and click **Highlight**. For the defined simulation time window, the `Low_Emissions` fueling mode is active and highlighted.

Select the data cursor for the time window 6–7.5 seconds, with `0 fuel_cal:1`. Click **Highlight**. In the `control_logic` model, the `Fuel_Disabled` state is highlighted. The engine is in `Shutdown` mode.

See Also

"Highlight Functional Dependencies" on page 8-2 | "Refine Highlighted Model" on page 8-12

Debug Slice Simulation by Using Fast Restart Mode

Perform multiple slicer simulations and streamline model debugging workflows by using Model Slicer in fast restart mode. For more information, see "Get Started with Fast Restart".

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".

Simulate and Debug a Test Case in a Model Slice

This example shows how the fast restart mode performs slicer simulations with different test case inputs, without recompiling the model. You can simulate a sliced harness model with a test case input and highlight the dependency analysis in the model.

Analyze the highlighted slice by stepping through the time steps. You use the simulation stepper to analyze how the slice changes at each time step.

1 Open the `sldvdemo_cruise_control` model.

```
open_system('sldvdemo_cruise_control');
```

2 Set `sldvoptions` parameters and analyze the model by using the specified options.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';              % Perform test-generation analysis
opts.ModelCoverageObjectives = 'MCDC';     % Specify type of model coverage
opts.SaveHarnessModel = 'on';              % Save harness as model file
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
```

After the analysis, the software opens a harness model `sldvdemo_cruise_control_harness` and saves it in the default location `<current_folder>\sldv_output \sldvdemo_cruise_control\sldvdemo_cruise_control_harness.slx`. For more information, see "Simulink Design Verifier Harness Models" (Simulink Design Verifier).
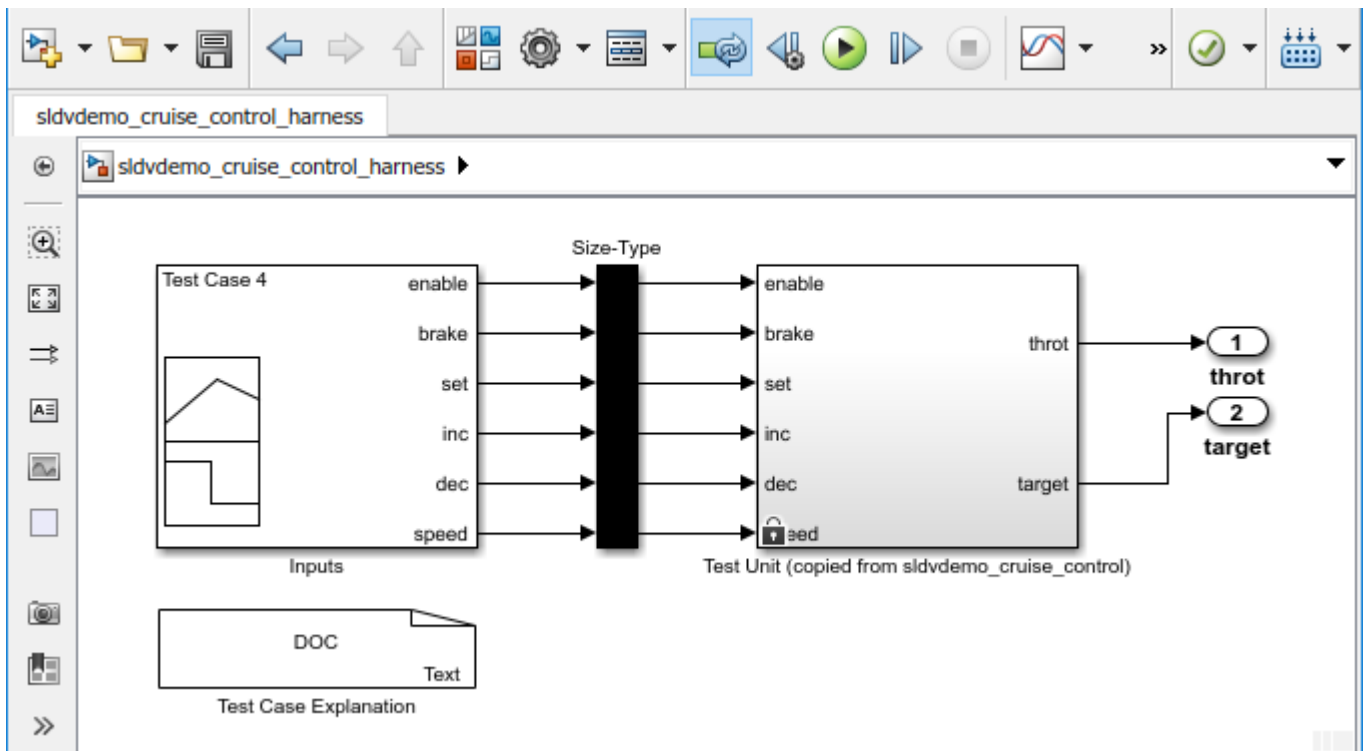
3

To enable the fast restart mode, click **Enable Fast Restart** button .

4 On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**. Model Slicer compiles the model.

Optionally, you can enable fast restart after opening the Model Slice Manager. Select **Simulation time window** and click the run simulation button . To enable fast restart, in the Record simulation time window, click the **here** link.
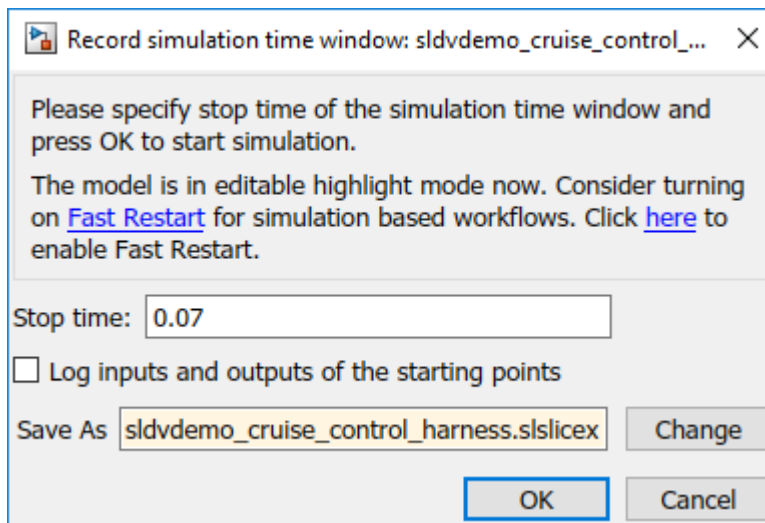
5 To add **Starting Points**, in the Model Slice Manager, click **Add all outports.**.

The `throt` and `target` outports are added as the **Starting Points**.

6 You can simulate a test case and analyze the highlighted dependencies in the slice.

    a In the Signal Builder block, select `Test Case 4`.

    b
To simulate the test case, click **Start simulation** button, .

Optionally, you can simulate the model by using the **Run** button in the Simulink editor. You can also simulate by using the **Simulation time window** in the Model Slice Manager.

The slice shows the highlighted dependencies for the `Test Case 4` inputs.
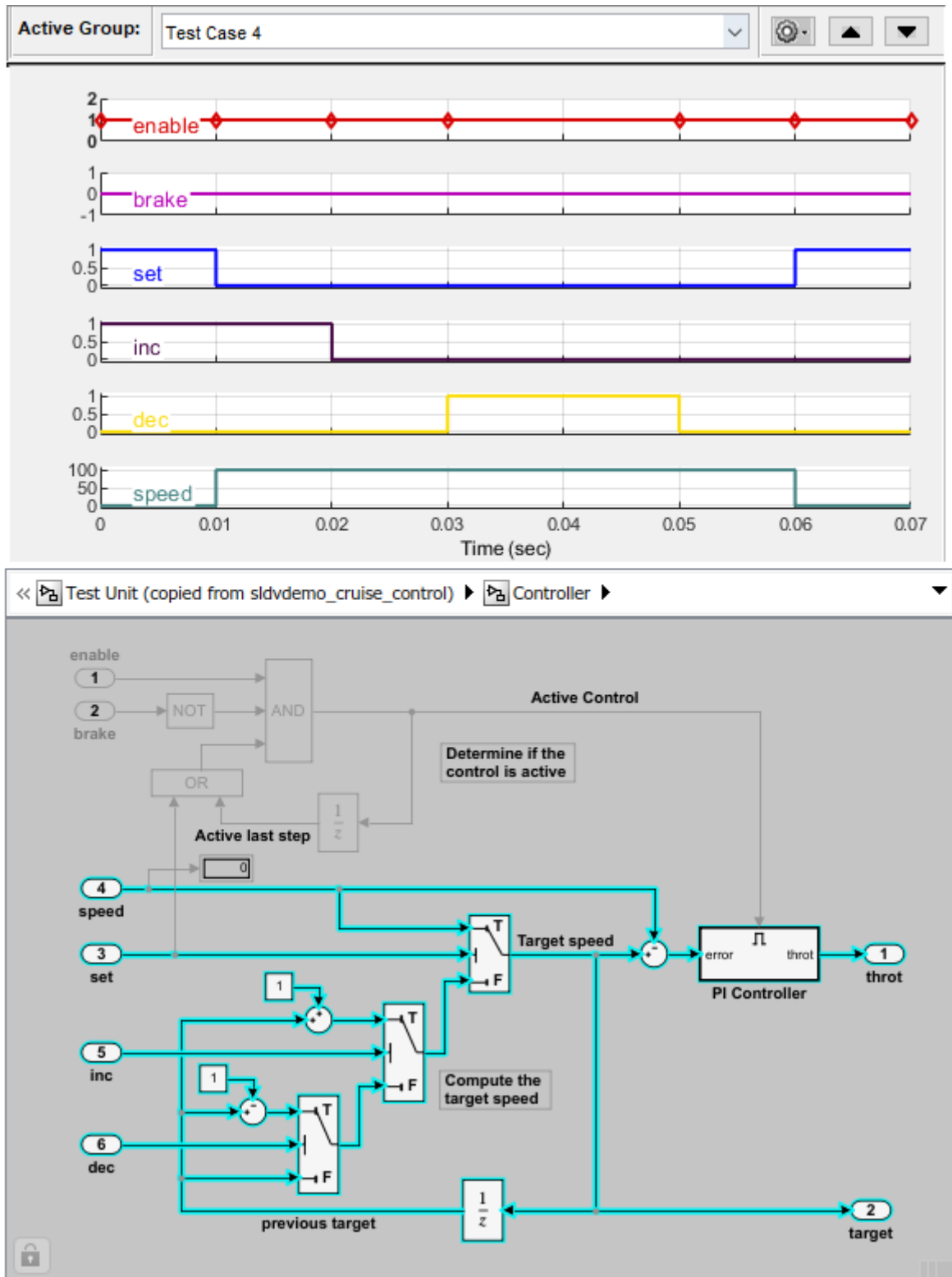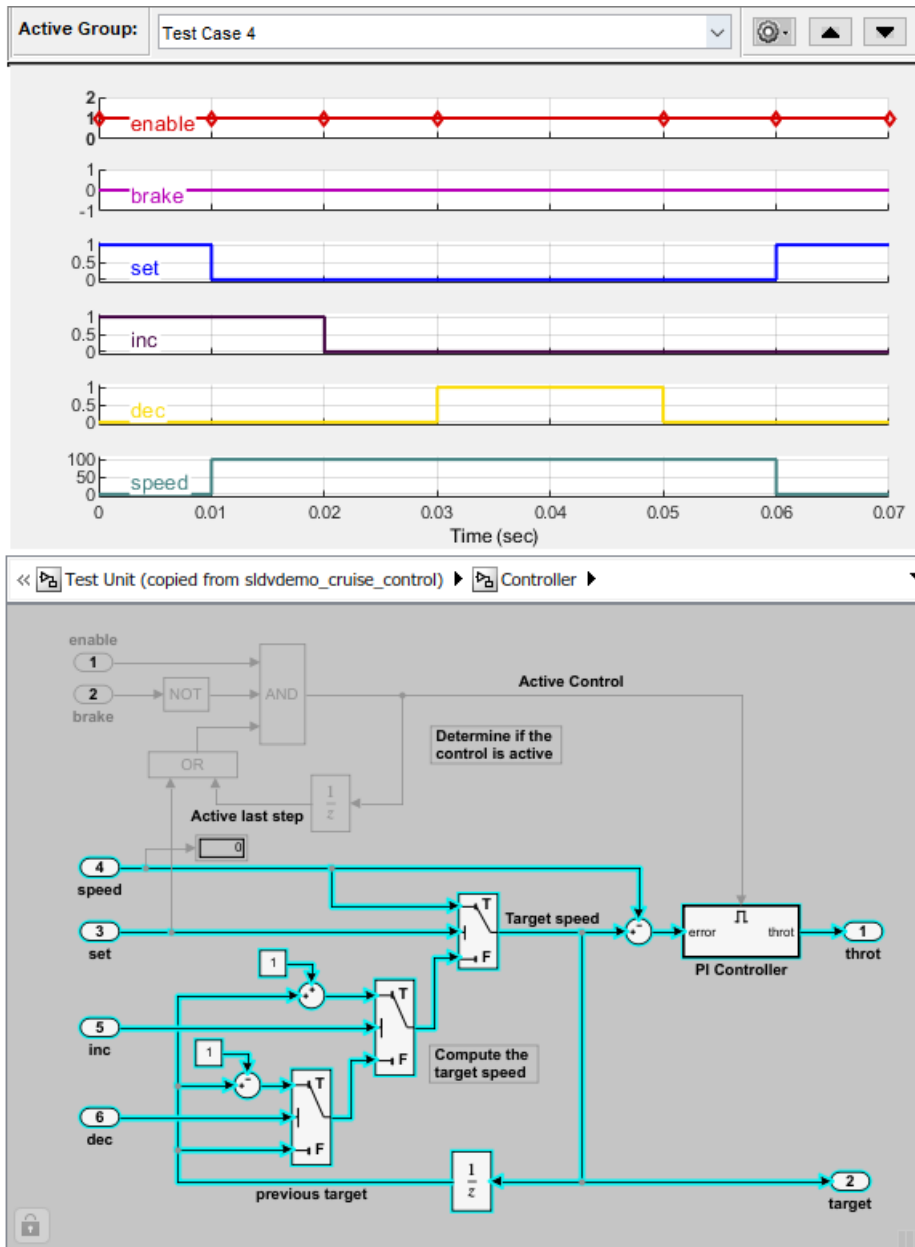
You can simulate a slice for different test case inputs and analyze the dependency analysis.

7   Debug a slicer simulation by using a simulation stepper. For more information see, "Simulation Stepper Access".
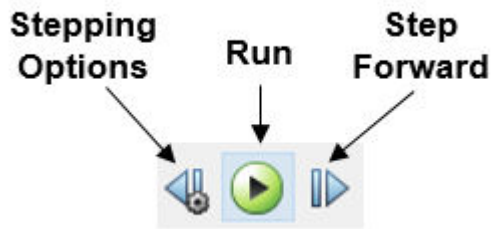
a To debug the simulation for the test case, in the Simulink Editor for the `sldvdemo_cruise_control_harness` model, click **Step Forward** button. You can view the signal values and the highlighted slice at each time step. For more information, see "Simulation Stepping Options". The signal values and the dependencies at T=0.010 appears.
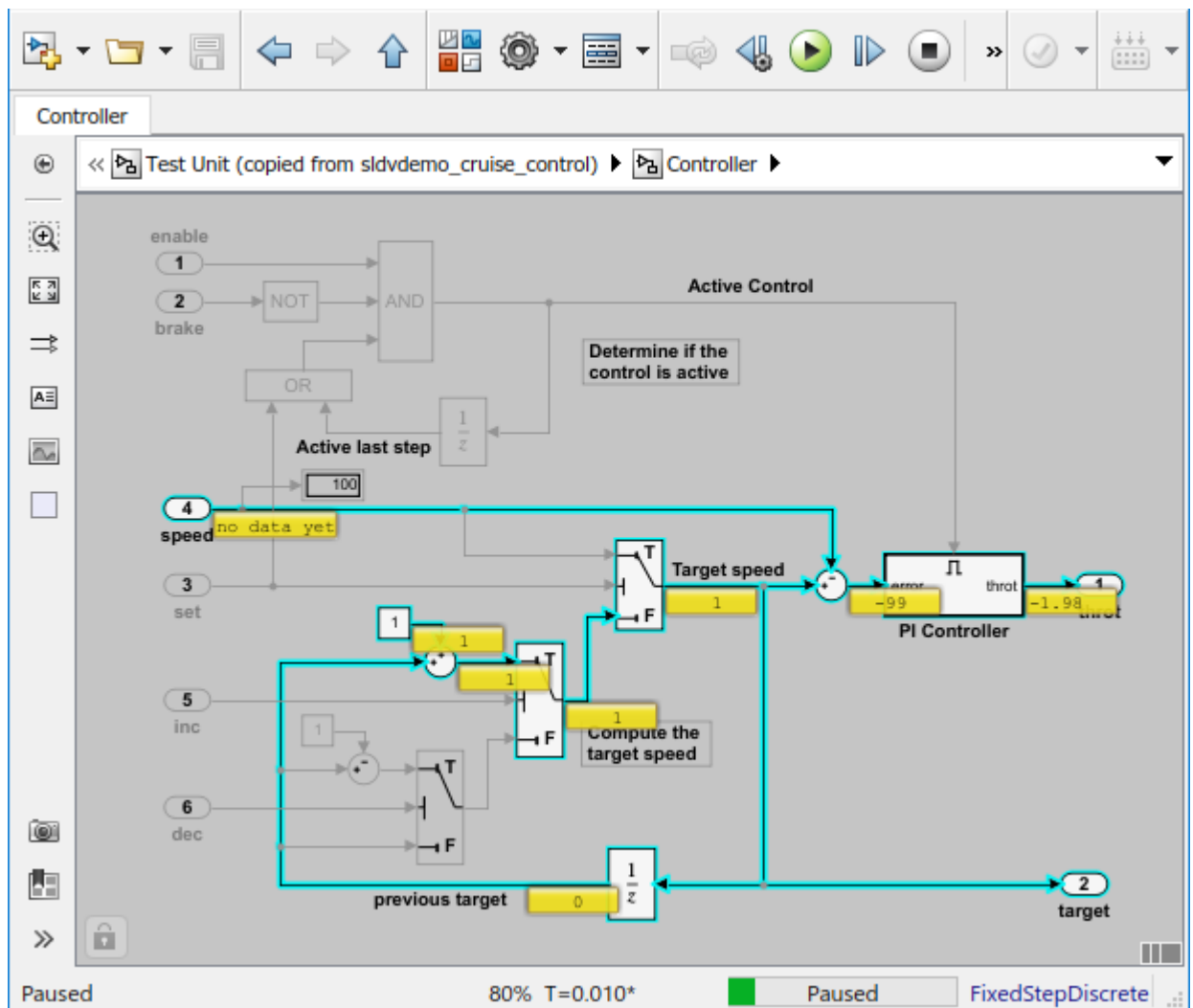
b To debug the slice at T=0.030, step forward and view the signal values and the highlighted slice.
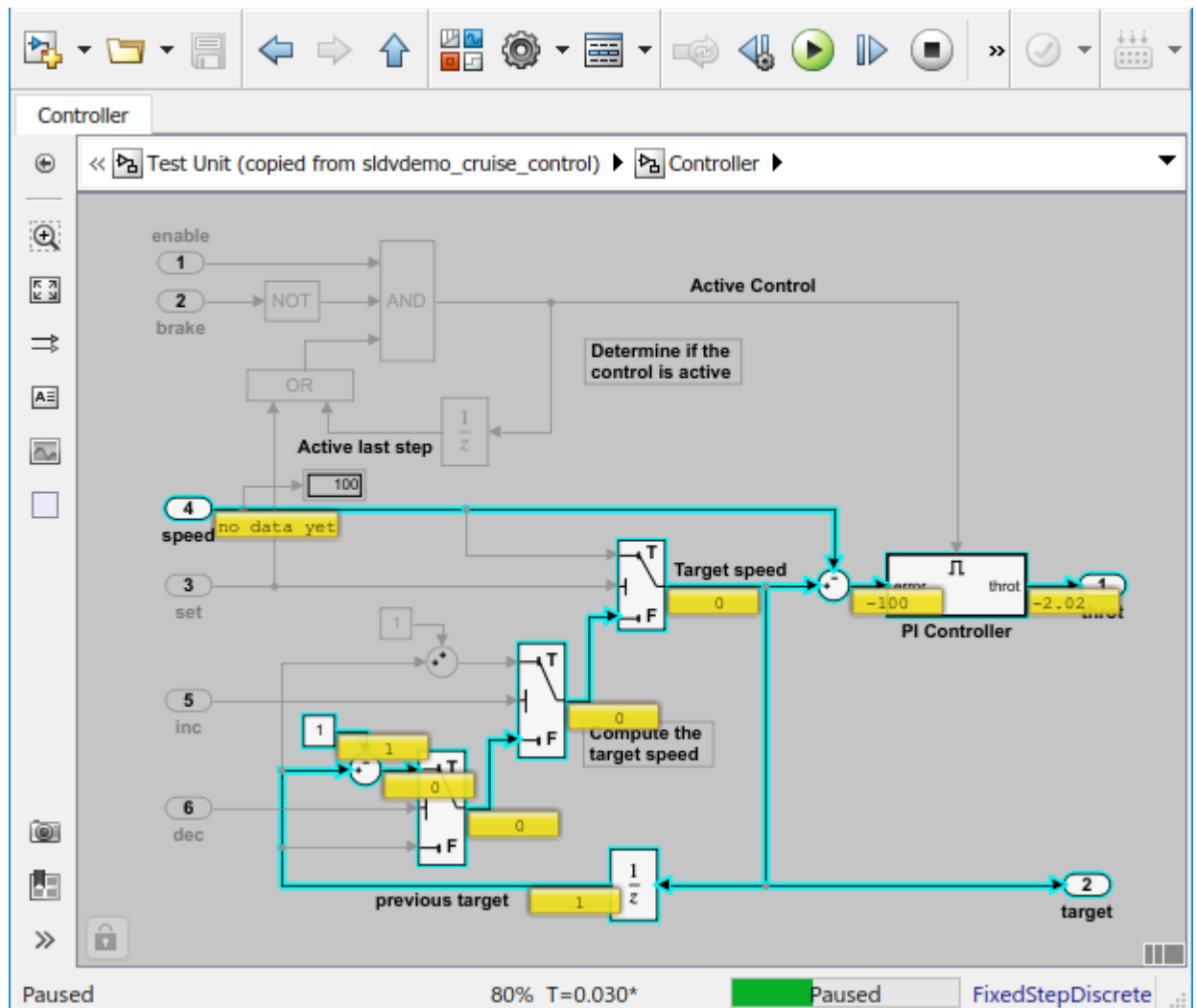
8 To complete the simulation stepping, click the **Run** button.

See Also

More About
- "Highlight Functional Dependencies" on page 8-2
- "Simulation Stepper"
- "Get Started with Fast Restart"

Isolate Referenced Model for Functional Testing

To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

This example shows how to slice the referenced model controller of a fault-tolerant fuel control system for functional testing. To create a simplified open-loop referenced model for debugging and refinement, you generate a slice of the referenced controller.

Step 1: Open the Model

The fault-tolerant fuel control system model contains a referenced model controller `fuel_rate_control`.

`open_system('sldvSlicerdemo_fuelsys');`

Step 2: Slice the Referenced Model

To analyze the `fuel_rate_control` referenced model, you slice it to create a standalone open-loop model. To open the Model Slice Manager, select **Apps > Model Verification, Validation, and Test > Model Slicer**, or right-click the `fuel_rate_control` model and select **Model Slicer > Slice component**. When you open the Model Slice Manager, the Model Slicer compiles the model. You then configure the model slice properties.

Note: The simulation mode of the `sldvSlicerdemo_fuelsys` model is `Accelerator` mode. When you slice the referenced model, the software configures the simulation mode to `Normal` mode and sets it back to its original simulation mode while exiting the Model Slicer.
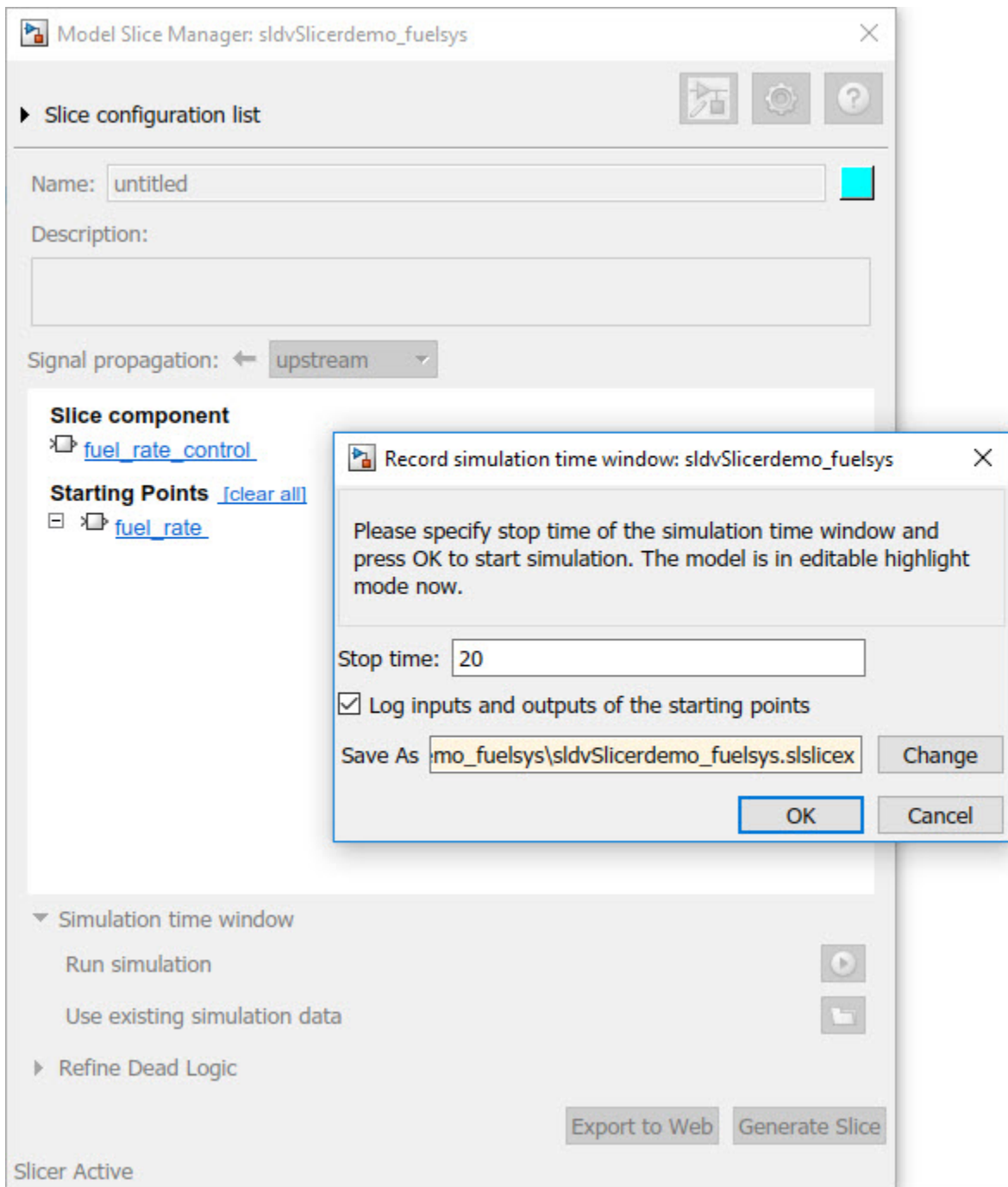
Step 3: Select Starting Point

Open the `fuel_rate_control` model, right-click the `fuel-rate` port, and select **Model Slicer >
Add as starting point**. The Model Slicer highlights the upstream constructs that affect the
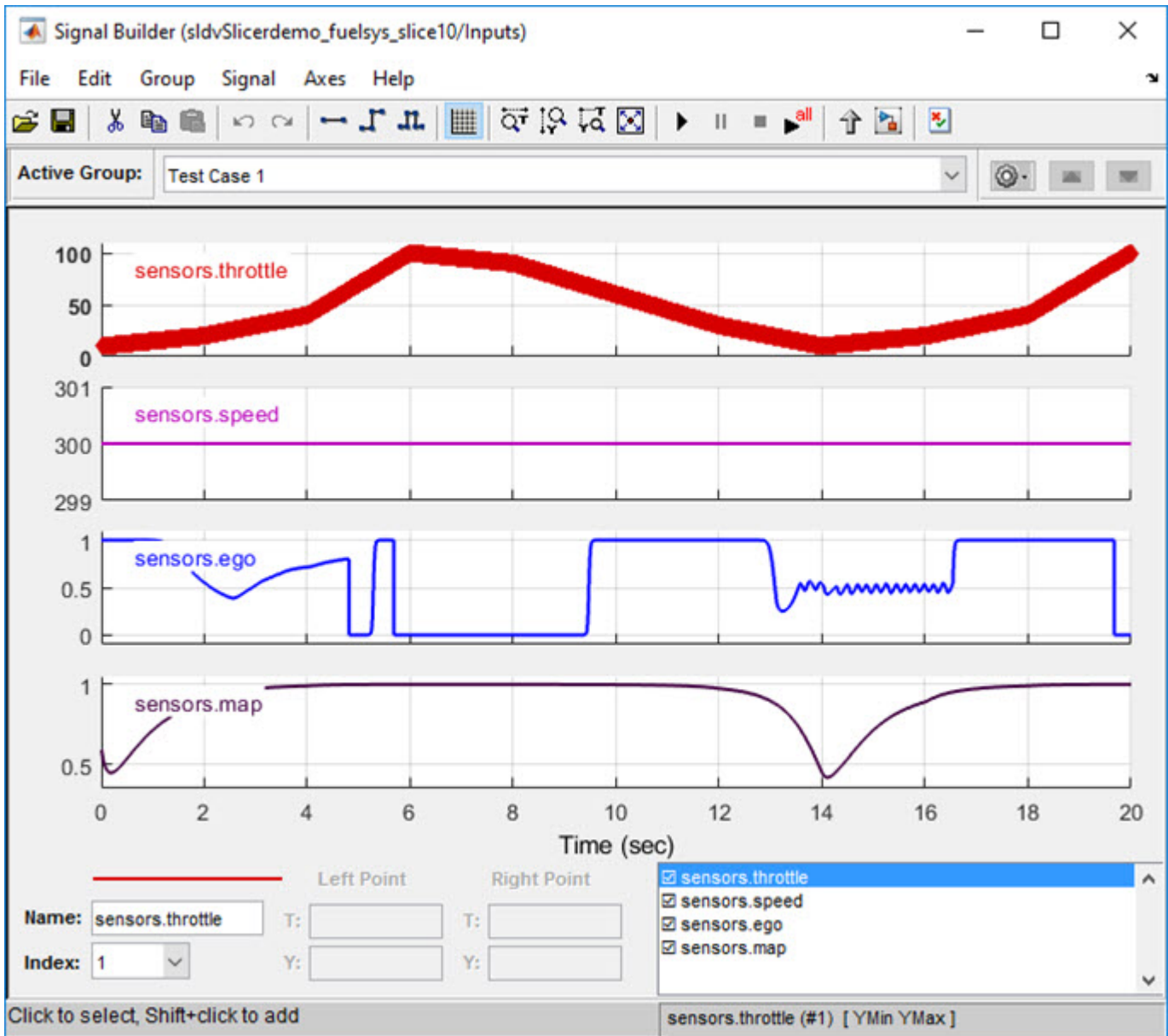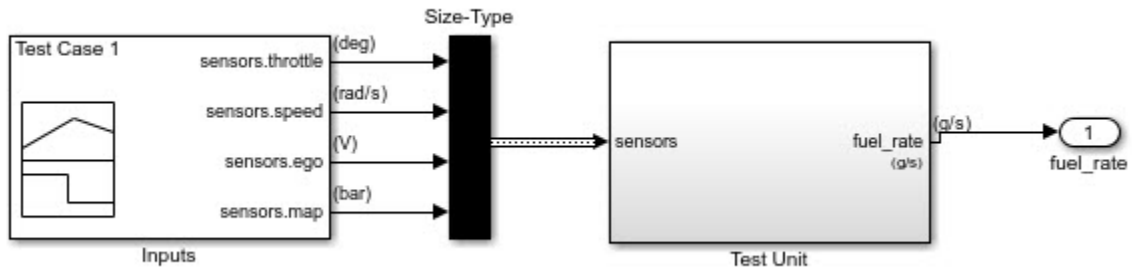`fuel_rate`.

Step 4: Generate Slice

a. In the Model Slice Manager dialog box, select the **Simulation time window**.

b. Click **Run simulation**.

c. For the **Stop time**, enter `20`. Click **OK**.

d. Click **Generate Slice**. The software simulates the sliced referenced model by using the inputs of
the close-loop `sldvSlicerdemo_fuelsys` model.

For the sliced model, in the Signal Builder window, one test case is displayed that represents the signals input to the referenced model for simulation time 0–20 seconds.
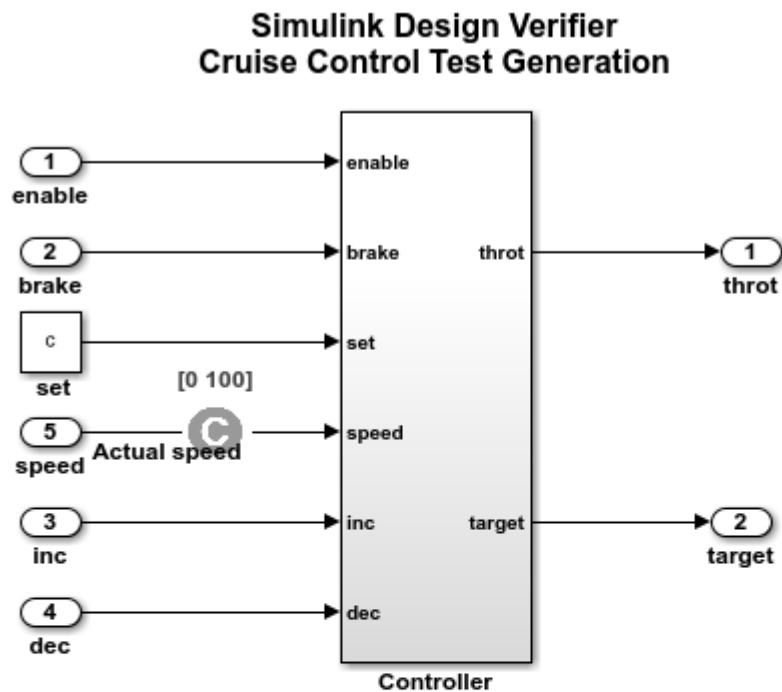
Analyze the Dead Logic

This example shows how to refine the model for dead logic. The `sldvSlicerdemo_dead_logic` model consists of dead logic paths that you refine for dependency analysis.

1. Open the `sldvSlicerdemo_dead_logic` model.

2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```

Simulink Design Verifier
Cruise Control Test Generation

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.
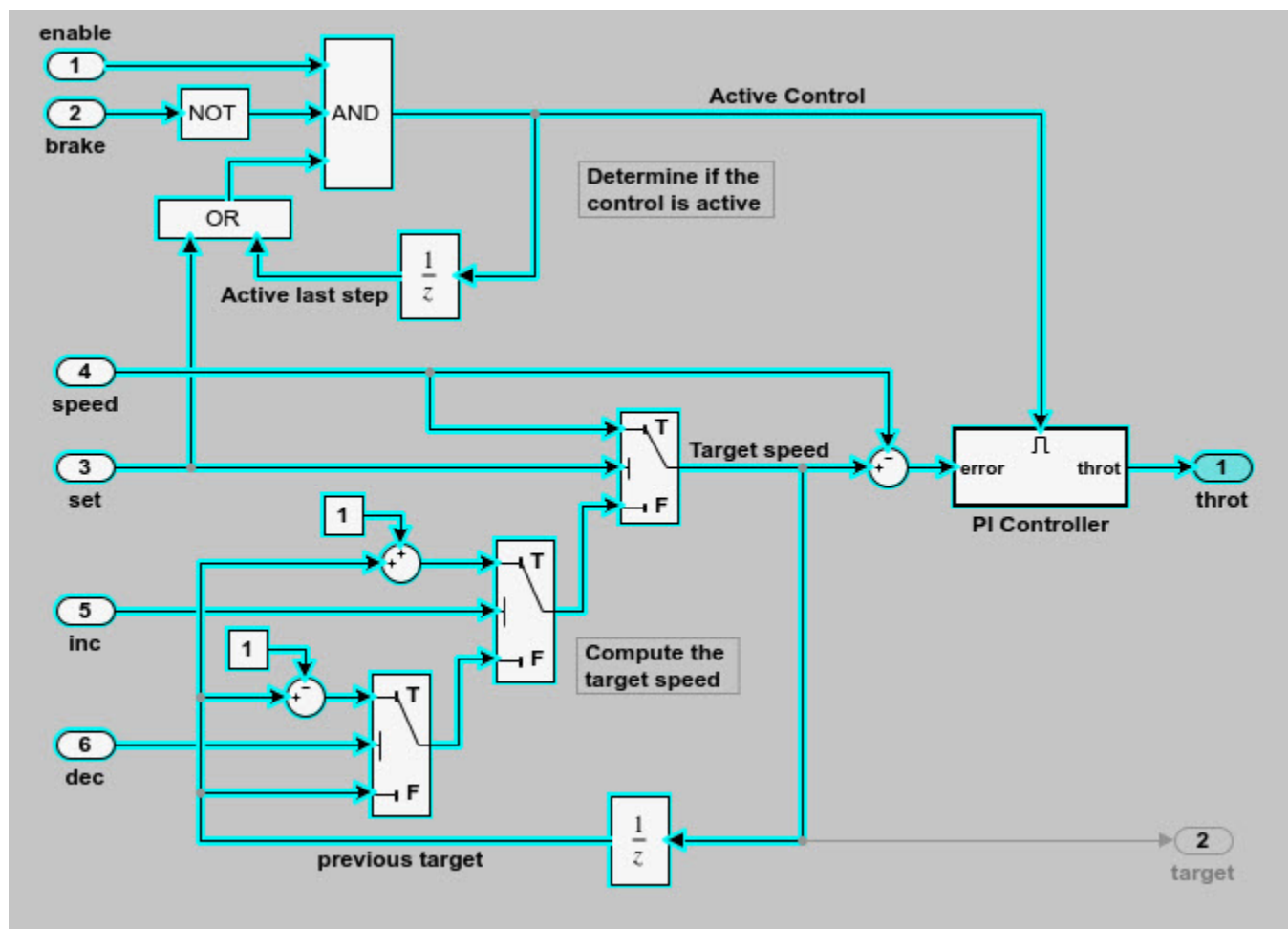
Run
(double-click)

Toggle Speed
Constraint
(double-click)

View Options
(double-click)

Toggle Constraint
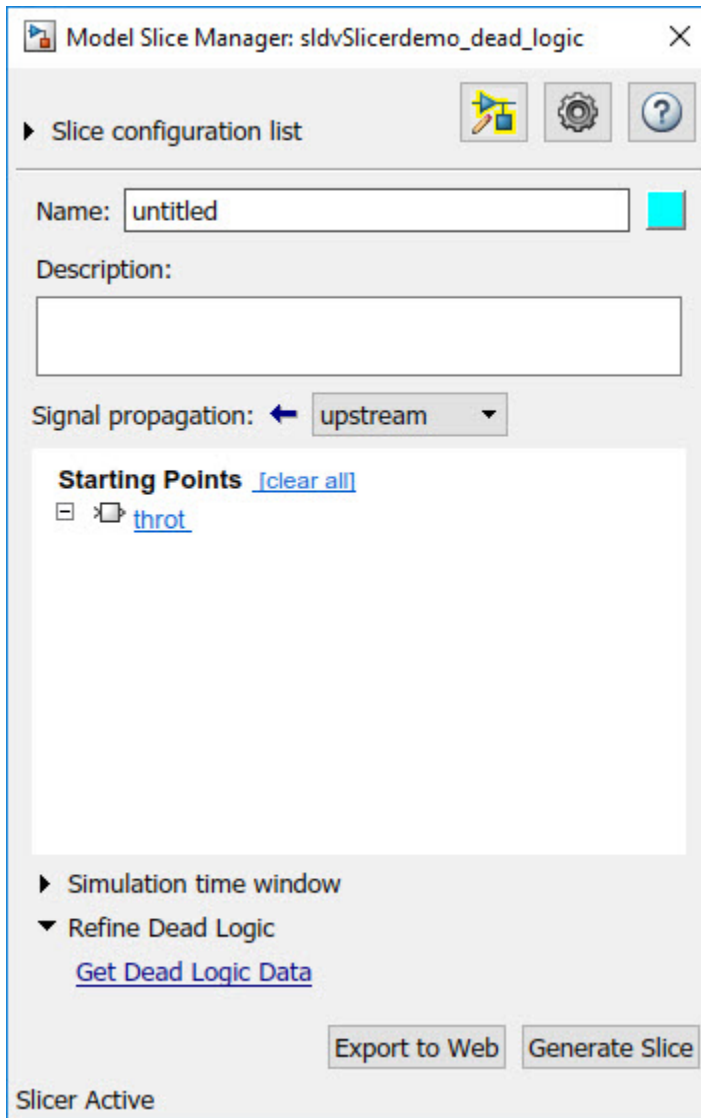
Copyright 2006-2018 The MathWorks, Inc.

Open the `Controller` subsystem and add the outport `throt` as the starting point.
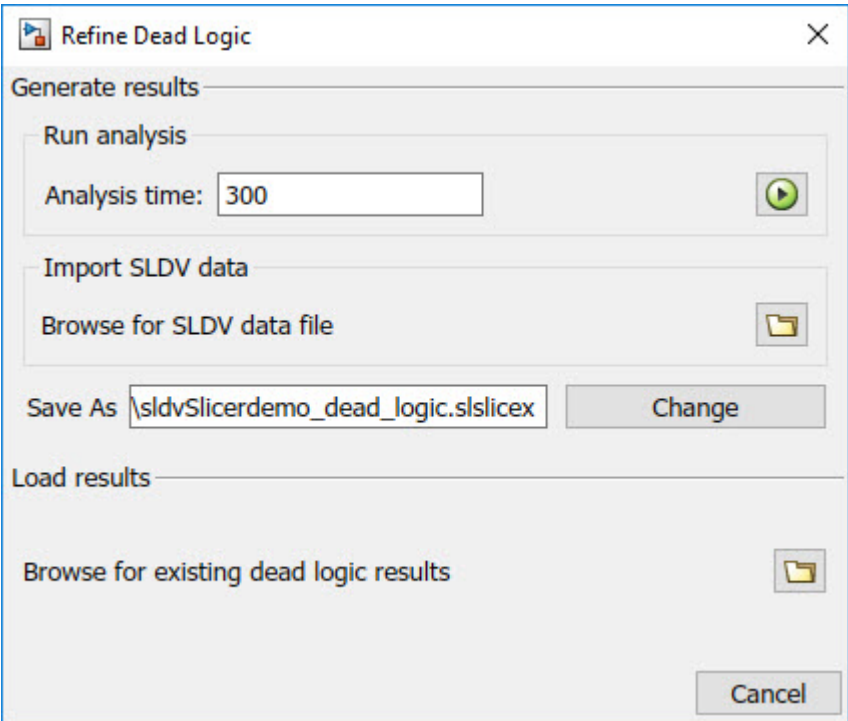
The Model Slicer highlights the upstream dependency of the `throt` outport.
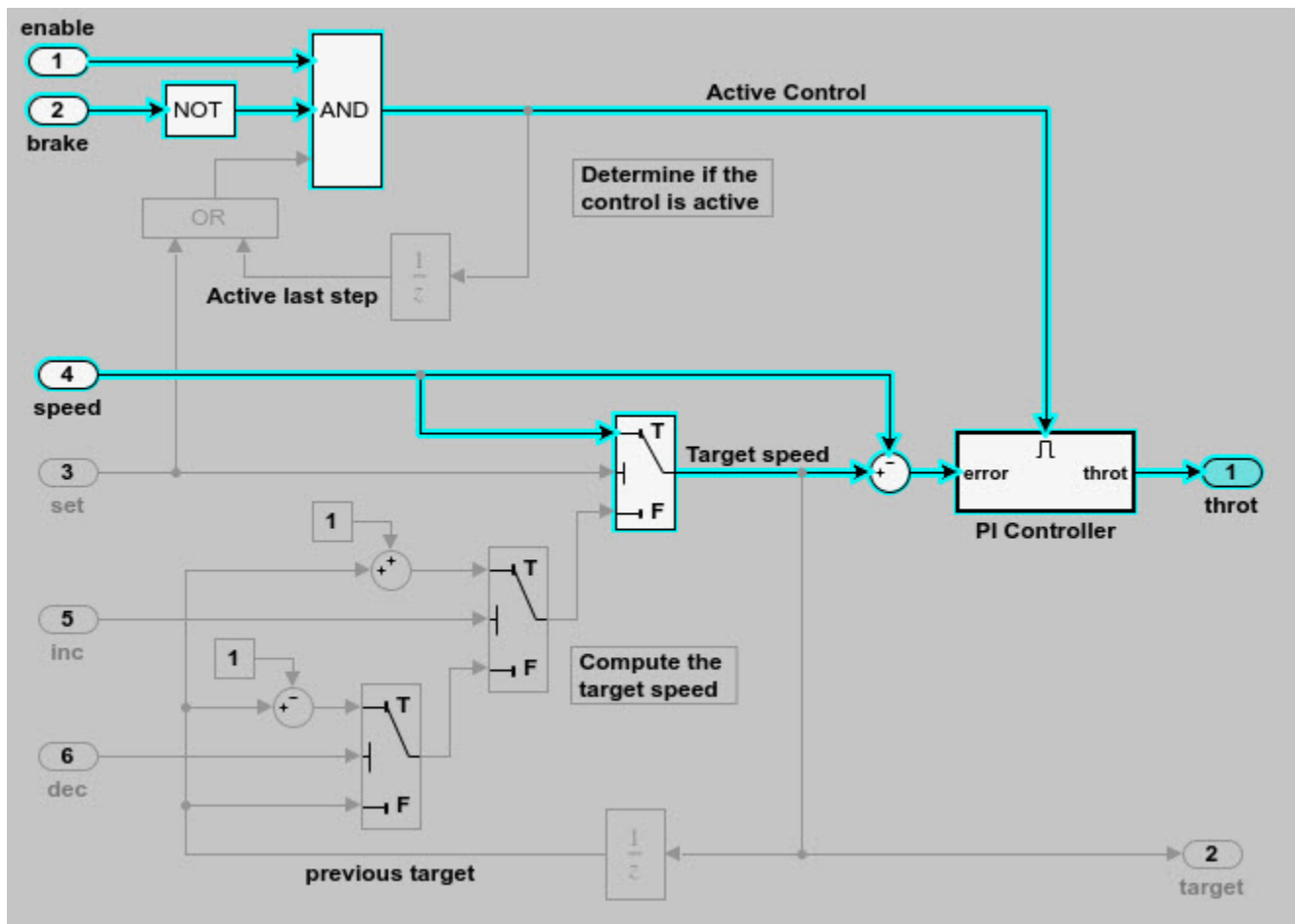
2. In the Model Slice Manager, select **Refine Dead Logic**.

3. Click **Get Dead Logic Data**.

4. Specify the **Analysis time** and run the analysis. You can import existing dead logic results from the `sldvData` file or load existing `.slslicex` data for analysis. For more information, see "Refine Highlighted Model by Using Existing .slslicex or Dead Logic Results" on page 8-63.

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.

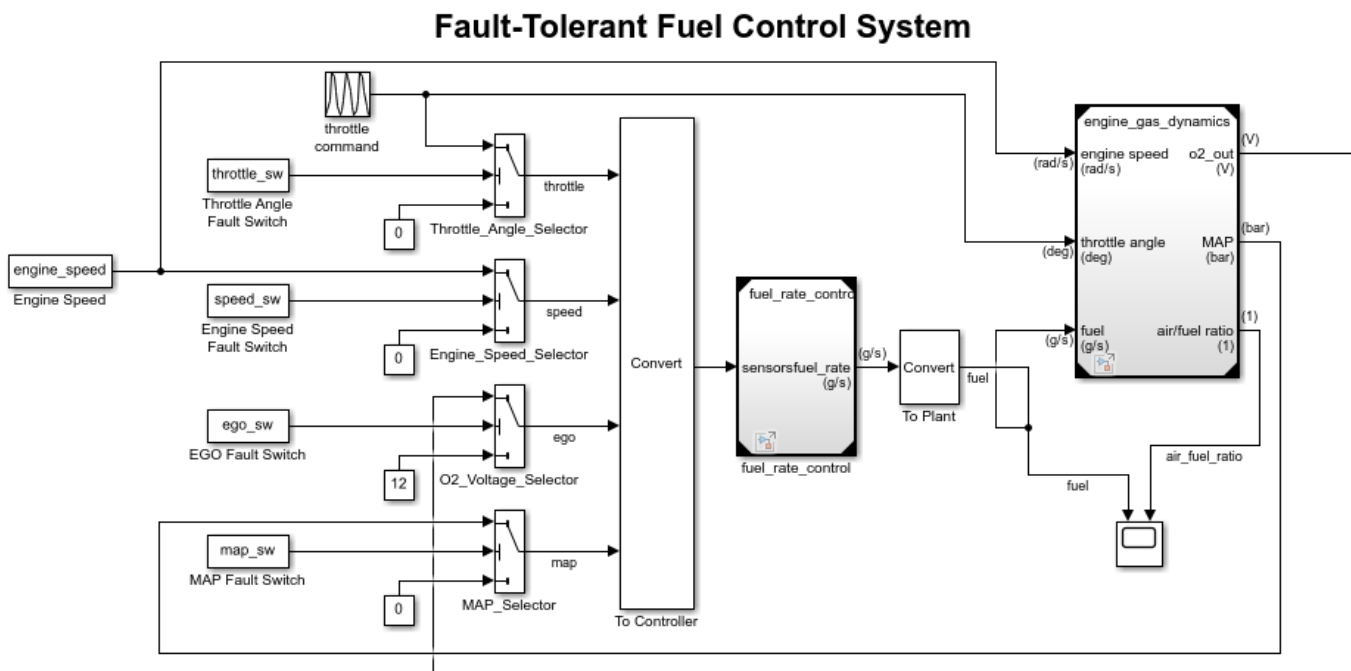# Investigate Highlighted Model Slice by Using Model Slicer Data Inspector

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".

Step 1: Start the Model Slice Manager

To start the Model Slice Manager, open the `fuel_rate_control` model, and select **Apps > Model Verification, Validation, and Test > Model Slicer**.

```
open_system('sldvSlicerdemo_fuelsys');
```
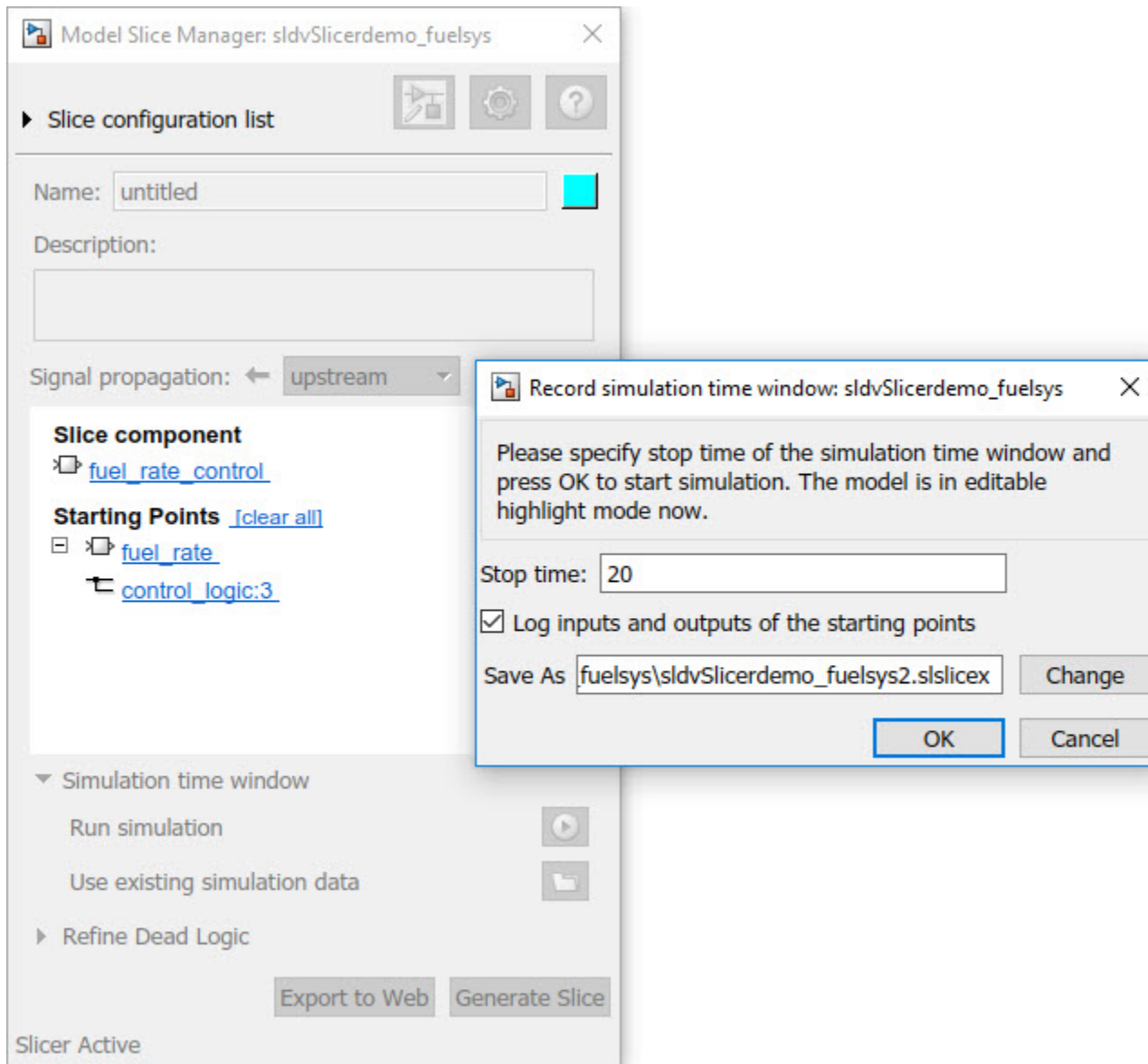
Fault-Tolerant Fuel Control System
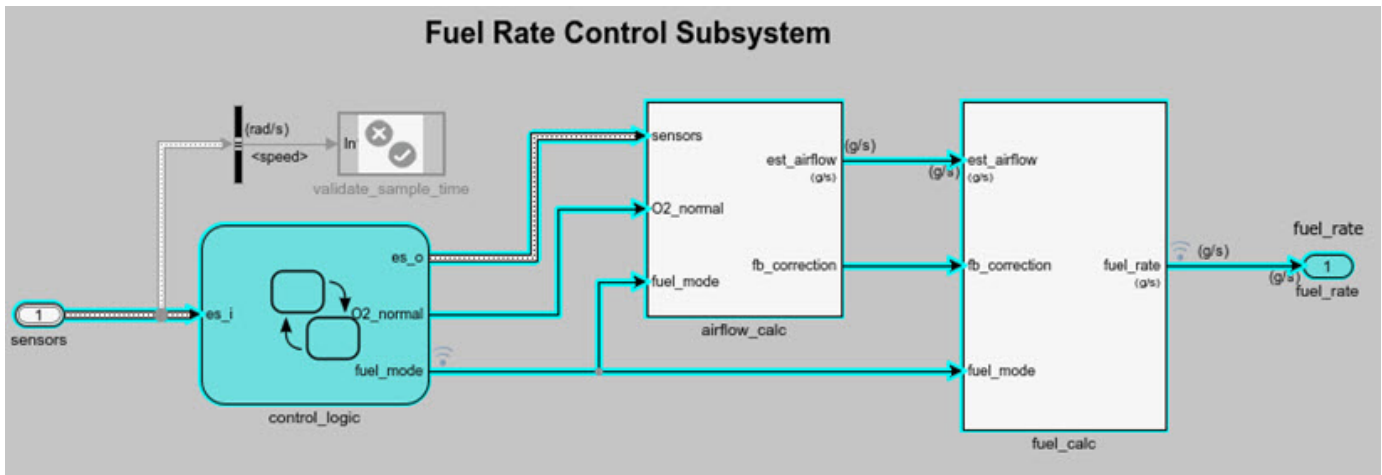
Copyright 1990-2017 The MathWorks, Inc.

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.

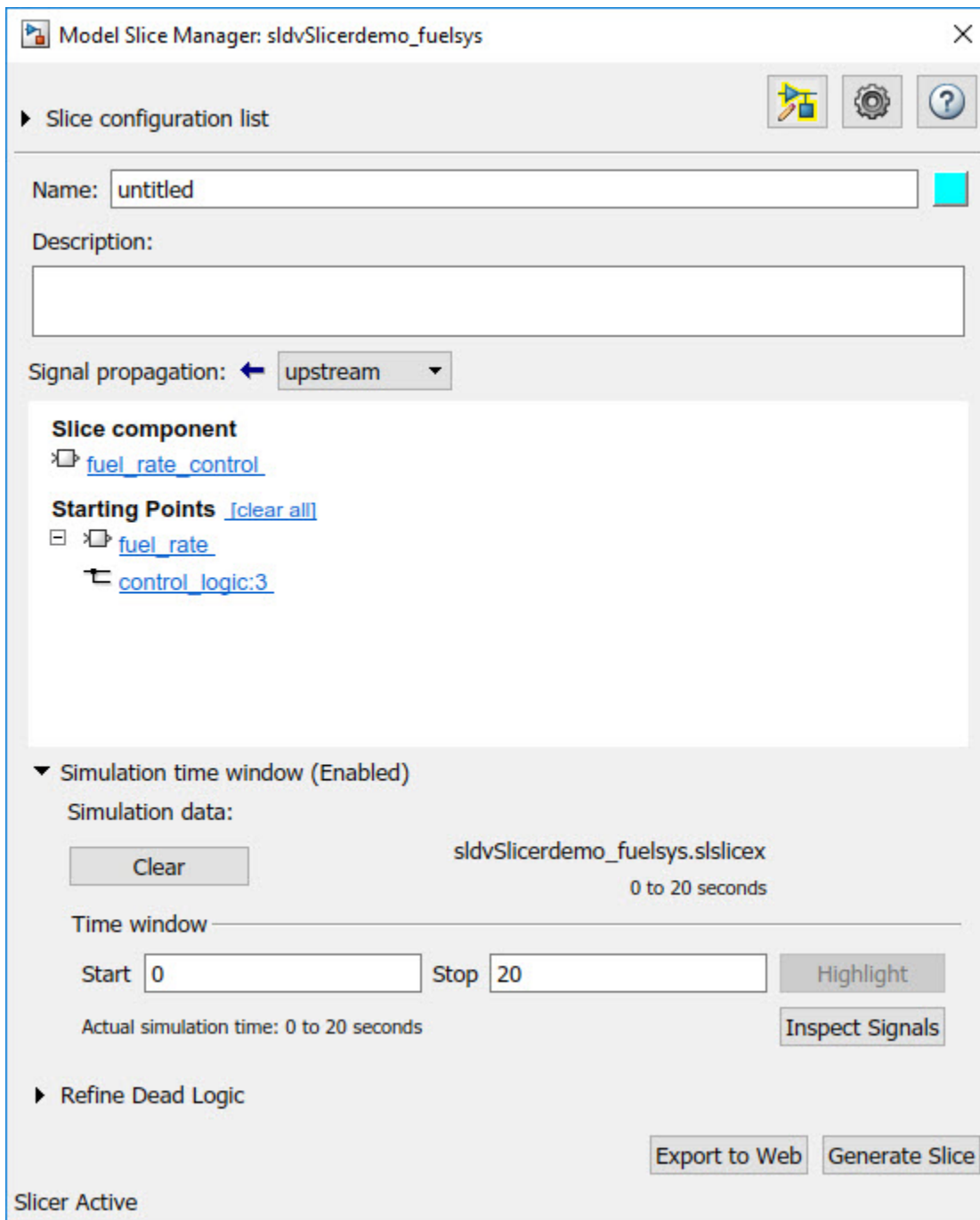c. Select the **Log inputs and outputs of the starting points**.

d. Click **OK**.
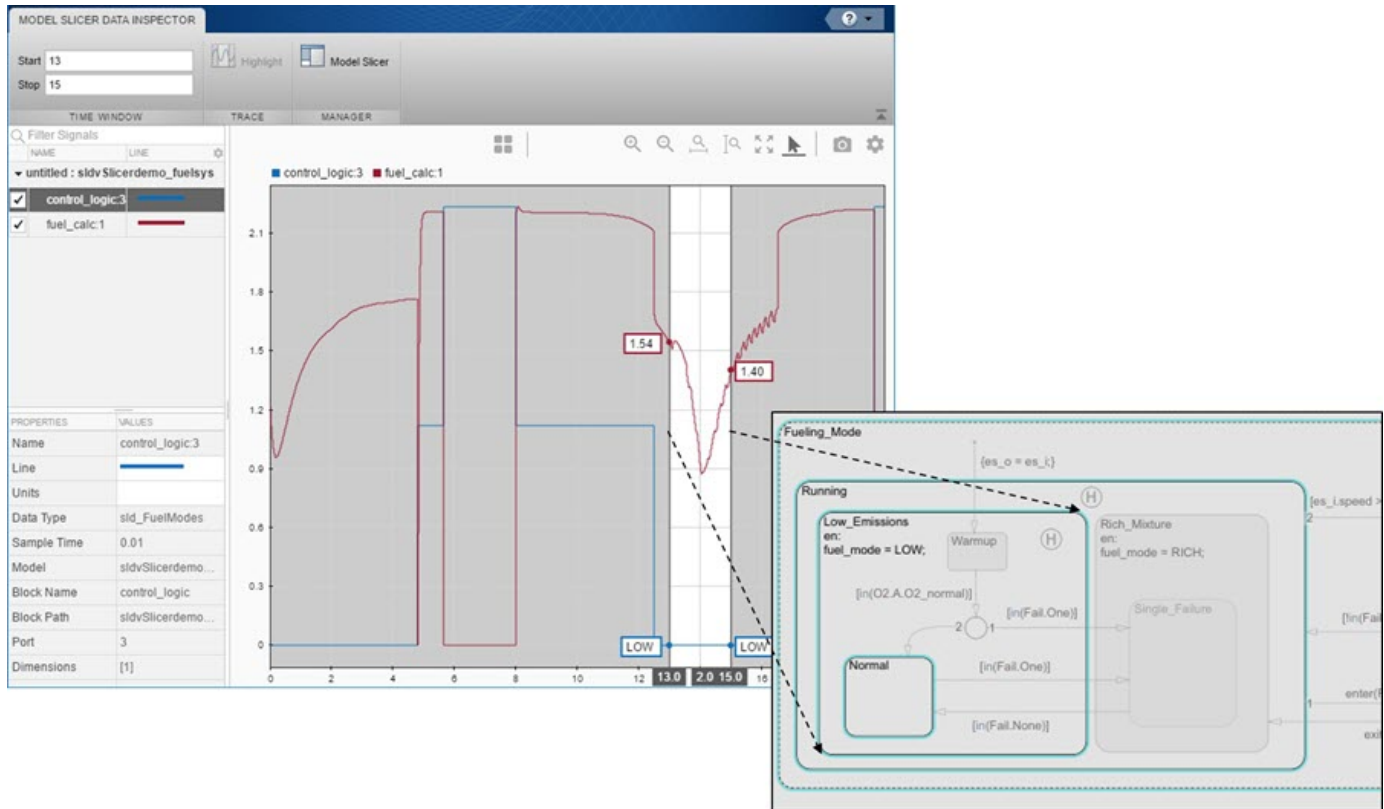
Step 3: Inspect signals

To open the Model Slicer Data Inspector, click **Inspect Signals**.

The logged input and output signals appear in the Model Slicer Data Inspector. When you open the Model Slicer Data Inspector, Model Slicer saves the existing Simulation Data Inspector session as MLDATX-file in the current working directory.

You can select the time window by dragging the data cursors to a specific location or by specifying the **Start** and **Stop** time in the navigation pane. To highlight the model for the defined simulation time window, Click **Highlight**.

To investigate the `Fueling_mode`, open the `control_logic` Stateflow™ chart, available in the `fuel_rate_control` referenced model. Select the time window for 13–15 seconds and click **Highlight**. For the defined simulation time window, the `Low_Emissions` fueling mode is active and highlighted.

Select the data cursor for the time window 6–7.5 seconds, with `0 fuel_cal:1`. Click **Highlight**. In the `control_logic` model, the `Fuel_Disabled` state is highlighted. The engine is in `Shutdown` mode.

Programmatically Generate I/O Dependency Matrix

This example shows how to programmatically generate a dependency matrix that shows the relationship between root level inports and outports.

To create the dependency matrix:

1. Open model slcheckSliceCruiseControl.

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

2. Create a SysDependencyTabulator object.

```
obj=SysDependencyTabulator(model);
```

3. Initialise a model handle.

```
sysH = get_param(model, 'handle');
```

4. Use **tabulateDependencies** method to create a dependency matrix for the model handle.

```
T1 = obj.tabulateDependencies(sysH)
```

T1=*5×11 table*

	enbl	cncl	set	resume	inc	dec	brakeP	key	gear	thr
reqDrv	1	1	1	1	1	1	0	0	0	
status	1	1	1	1	1	1	1	1	1	
operation_mode	1	1	1	1	1	1	1	1	1	
targetSp	1	1	1	1	1	1	1	1	1	
throtCC	1	1	1	1	1	1	1	1	1	

5. Initialise a subsystem handle.

```
subsystemPath = [model '/CruiseControlMode'];
sysH = get_param(subsystemPath, 'handle');
```

6. Use **tabulateDependencies** method to create a dependency matrix for the subsystem handle.

```
T2 = obj.tabulateDependencies(sysH)
```

T2=*2×5 table*

	reqDrv	brakeP	vehSp	key	gear
status	1	1	1	1	1
mode	1	1	1	1	1

7. Delete the SysDependencyTabulator object.

```
delete(obj);
```

8. Close the model.

```
close_system(model);
```

Observe Impact of Simulink Parameters Using Model Slicer

Use Model Slicer to observe the impact a parameter has on a model.

This example demonstrates the ability of Model Slicer to display the parameters that affect a block (Option 1), and blocks that are affected by a parameter (Option 2) using the methods of the `SLSlicerAPI.ParameterDependence` class `parametersAffectingBlock`, and `blocksAffectedByParameter` repectively.

Open Model and Initialize ParameterDependence Class

1. Open the model `sldvSliceCruiseControl`.

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

2. Create an object of the `ParameterDependence` Class.

```
slicerObj = slslicer(model);
pd = slicerObj.parameterDependence;
```

Option 1: Find Parameters Affecting a Block

1. View the parameters that affect the `Switch3` block in the `DriverSwRequest` subsystem by entering:

```
params = parametersAffectingBlock(pd, 'sldvSliceCruiseControl/DriverSwRequest/Switch3')
```

```
params=1×49 object
  1x49 VariableUsage array with properties:

    Name
    Source
    SourceType
    Users
```

You can see that there are 49 parameters that affect the `Switch3` block. To view the details of individual parameters, explore each element of the array:

```
params(1)
```

```
ans =
  VariableUsage with properties:

          Name: 'CountValue'
        Source: 'sldvSliceCruiseControl/DriverSwRequest/decrement/counter'
    SourceType: 'mask workspace'
         Users: {'sldvSliceCruiseControl/DriverSwRequest/decrement/counter/Constant'}
```

Option 2: Get Blocks Affected By a Parameter

1. To observe the impact of a parameter, create a `Simulink.VariableUsage` object for that parameter.

```
param = Simulink.VariableUsage('CountValue','sldvSliceCruiseControl/DriverSwRequest/decrement/cou
```

2. To view all the blocks affected by `param`:

```
affectedBlocks = blocksAffectedByParameter(pd, param)
```

affectedBlocks = *1×153*
10³ ×

    0.0130   0.0280   0.0290   0.0720   0.0730   0.0740   0.0840   0.5110   0.5130   0.5

You can further refine the blocks affected using the same options supported by `find_system`.

```
affectedOutports = blocksAffectedByParameter(pd, param, 'blockType', 'Outport')
```

affectedOutports = *1×5*
10³ ×

    1.8770   1.8760   1.8780   1.8790   1.8800

Optional Step: Highlight Result on Model by Using Model Slicer

You can view the active section of the analyzed model by using the Model Slicer highlighting.

```
slicerObj.highlight(slicerObj.ActiveConfig);
```

Clean Up

Model Slicer maintains the model in compiled state after analysis. To close the model, terminate the `slicerObj` object.

```
slicerObj.terminate;
```