

Simulink® Check™

User's Guide



MATLAB® & SIMULINK®

R2018a



How to Contact MathWorks



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



Phone: 508-647-7000



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

Simulink® Check™ User's Guide

© COPYRIGHT 2004–2018 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2017	Online only	New for Version 4.0 (Release 2017b)
March 2018	Online only	Revised for Version 4.1 (Release 2018a)

1	Getting Started	
	Simulink Check Product Description	1-2
	Key Features	1-2
	Check for Standards Compliance in Your Model	1-3
	Detect and Fix Model Advisor Check Violations by Using Edit- Time Checking	1-3
	Detect Model Advisor Check Violations Interactively	1-5
	Collect Model Metric Data by Using the Metrics Dashboard	1-9
	Refactor Model with Clone Detection and Model Transformer Tools	1-13
	Identify and Replace Clones with Links to Library Blocks ...	1-13
	Replace Qualifying Modeling Patterns with Variant Blocks ..	1-16

2	Verification and Validation	
	Test Model Against Requirements and Report Results	2-2
	Requirements Overview	2-2
	Test a Cruise Control Safety Requirement	2-2
	Analyze a Model for Standards Compliance and Design Errors	2-6
	Standards and Analysis Overview	2-6
	Check Model for Style Guideline Violations and Design Errors	2-6

Perform Functional Testing and Analyze Test Coverage	2-9
Functional Testing and Coverage Analysis Overview	2-9
Incrementally Increase Test Coverage Using Test Case Generation	2-9
Analyze Code and Test Software-in-the-Loop	2-13
Code Analysis and Testing Software-in-the-Loop Overview . .	2-13
Analyze Code for Defects, Metrics, and MISRA C:2012	2-13
Module Verification and Testing Processor-in-the-Loop	2-22
Module Verification and Testing Processor-in-the- Loop Overview	2-22
Test a Model in Real Time	2-23
Real-Time Testing and Testing Production Models Overview	2-23

Checking Systems Interactively

3

Check for Compliance Using the Model Advisor and Edit-Time Checking	3-2
Check Your Model Interactively	3-2
Check Your Model While You Edit	3-3
Transform Model to Variant System	3-8
Example Model	3-8
Perform Variant Transform on Example Model	3-10
Model Transformation Limitations	3-12
Enable Component Reuse by Using Clone Detection	3-14
Exact Clones Versus Similar Clones	3-14
Identify Exact and Similar Clones	3-14
Save and View Clone Detection Reports	3-19
Additional Information	3-19
Improve Model Readability by Eliminating Local Data Store Blocks	3-21
Example Model	3-21
Replace Data Store Blocks	3-23

Limitations	3-25
Limit Model Checks	3-26
What Is a Model Advisor Exclusion?	3-26
Save Model Advisor Exclusions in a Model File	3-27
Save Model Advisor Exclusions in Exclusion File	3-27
Create Model Advisor Exclusions	3-28
Review Model Advisor Exclusions	3-29
Manage Exclusions	3-30
Edit-Time Exclusions	3-32
Limit Model Checks By Excluding Gain and Output Blocks	3-35
Model Checks for DO-178C/DO-331 Standard Compliance . .	3-39
Model Checks for IEC 61508, IEC 62304, ISO 26262, and EN 50128 Standard Compliance	3-47
Model Checks for MathWorks Automotive Advisory Board (MAAB) Guideline Compliance	3-55
Model Checks for Japan MATLAB Automotive Advisory Board (JMAAB) Guideline Compliance	3-61
Set MAAB and JMAAB Checks to Check Under Masks or Follow Links	3-69
Control Whether Checks Look Under Masks or Follow Links	3-69
Model Checks for MISRA C:2012 Compliance	3-70
Model Checks for Secure Coding (CERT C, CWE, and ISO/IEC TS 17961 Standards)	3-71
Model Checks for Requirements Links	3-72
Generate Model Advisor Reports in Adobe PDF and Microsoft Word Formats	3-73
Modify Default Template	3-73

Check Systems Programmatically

4

Checking Systems Programmatically	4-2
Find Check IDs	4-3
Create a Function for Checking Multiple Systems	4-5
Check Multiple Systems in Parallel	4-7
Create a Function for Checking Multiple Systems in Parallel	4-8
Archive and View Results	4-10
Archive Results	4-10
View Results in Command Window	4-10
View Results in Model Advisor Command-Line Summary Report	4-11
View Results in Model Advisor GUI	4-12
View Model Advisor Report	4-13
Archive and View Model Advisor Run Results	4-14

Model Metrics

5

Collect and Explore Metric Data by Using the Metrics Dashboard	5-2
Size	5-3
Modeling Guideline Compliance	5-4
Architecture	5-5
Dashboard Limitations	5-7
Collect Model Metrics Using the Model Advisor	5-8
Create a Custom Model Metric	5-10
Create Model Metric for Nonvirtual Block Count	5-10
Limitations	5-16

Collect Model Metrics Programmatically	5-18
Example Model	5-18
Collect Metrics	5-18
Access Results	5-19
Display and Store Results	5-19
Limitations	5-20
Model Metric Data Aggregation	5-22
How Model Metric Aggregation Works	5-22
Access Aggregated Metric Data	5-24

Overview of Customizing the Model Advisor

6

Model Advisor Customization	6-2
Requirements for Customizing the Model Advisor	6-2

Create Model Advisor Checks

7

Create Model Advisor Checks Workflow	7-2
Customization File Overview	7-3
Common Utilities for Creating Checks	7-5
Create and Add Custom Checks - Basic Examples	7-6
Add Custom Check to By Product Folder	7-6
Create Customized Pass/Fail Check	7-7
Create Customized Pass/Fail Check with Fix Action	7-10
Create Check for Model Configuration Parameters	7-15
Create Data File for Diagnostics Pane Configuration Parameter Check	7-15
Create Check for Diagnostics Pane Model Configuration Parameters	7-18
Data File for Configuration Parameter Check	7-20

Define Checks for Supported or Unsupported Blocks and Parameters	7-28
Example	7-28
Create Block Parameter Constraints	7-29
Create Model Advisor Checks from Constraints	7-32
Register Checks	7-35
Create sl_customization Function	7-35
Register Checks	7-35
Define Startup and Post-Execution Actions Using Process Callback Functions	7-37
Process Callback Function Arguments	7-37
Process Callback Function	7-38
Tips for Using the Process Callback Function in a sl_customization File	7-39
Define Custom Checks	7-40
About Custom Checks	7-40
Contents of Check Definitions	7-40
Display and Enable Checks	7-41
Define Where Custom Checks Appear	7-42
Check Definition Function	7-43
Define Check Input Parameters	7-43
Define Model Advisor Result Explorer Views	7-45
Define Check Actions	7-46
Create Callback Functions and Results	7-48
About Callback Functions	7-48
Informational Check Callback Function	7-49
Simple Check Callback Function	7-50
Detailed Check Callback Function	7-51
Check Callback Function with Hyperlinked Results	7-52
Action Callback Function	7-55
Check With Subchecks and Actions	7-56
Basic Check with Pass/Fail Status	7-58
Exclude Blocks From Custom Checks	7-62
Format Check Results	7-65
Format Results	7-65
Format Text	7-65
Format Lists	7-66

Format Tables	7-66
Format Paragraphs	7-68
Formatted Output	7-68
Format Linebreaks	7-69
Format Images	7-69

Create Custom Configurations by Organizing Checks and Folders

8

Create Custom Configurations	8-2
Create Configurations by Organizing Checks and Folders	8-3
Create Procedural-Based Configurations	8-0
Organize Checks and Folders Using the Model Advisor	
Configuration Editor	8-5
Overview of the Model Advisor Configuration Editor	8-5
Start the Model Advisor Configuration Editor	8-8
Organize Checks and Folders Using the Model Advisor	
Configuration Editor	8-9
Organize Customization File Checks and Folders	8-11
Customization File Overview	8-11
Register Tasks and Folders	8-12
Define Custom Tasks	8-13
Define Custom Folders	8-14
Customization Example	8-16
Verify and Use Custom Configurations	8-17
Update the Environment to Include Your sl_customization	
File	8-17
Verify Custom Configurations	8-17
Customize Model Advisor Check for Nondefault Block	
Attributes	8-19
Automatically Fix Display of Nondefault Block Parameters	8-20

Create Procedural-Based Model Advisor Configurations

9

Create Procedures	9-2
What Is a Procedure?	9-2
Create Procedures Using the Procedures API	9-2
Define Procedures	9-2
Create Procedural-Based Configurations	9-5
Overview of Procedural-Based Configurations	9-5
Create a Procedural-Based Configuration	9-6

Deploy Custom Configurations

10

Overview of Deploying Custom Configurations	10-2
About Deploying Custom Configurations	10-2
Deploying Custom Configurations Workflow	10-2
How to Deploy Custom Configurations	10-3
Manually Load and Set the Default Configuration	10-4
Automatically Load and Set the Default Configuration	10-5

Getting Started

- “Simulink Check Product Description” on page 1-2
- “Check for Standards Compliance in Your Model” on page 1-3
- “Collect Model Metric Data by Using the Metrics Dashboard” on page 1-9
- “Refactor Model with Clone Detection and Model Transformer Tools” on page 1-13

Simulink Check Product Description

Verify compliance with style guidelines and modeling standards

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

Simulink Check provides metrics such as size and complexity that you can use to evaluate your model's architecture and compliance to standards. A consolidated metrics dashboard lets you assess design status and quality. Automatic model refactoring lets you replace duplicate design elements, reduce design complexity, and identify reusable content.

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

Key Features

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

Check for Standards Compliance in Your Model

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

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

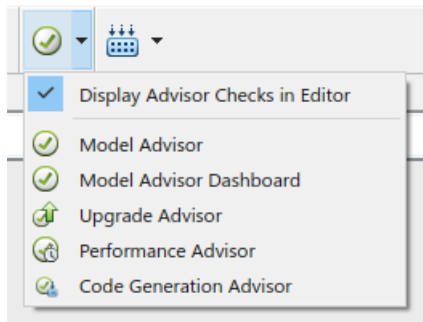
Detect and Fix Model Advisor Check Violations by Using Edit-Time Checking

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

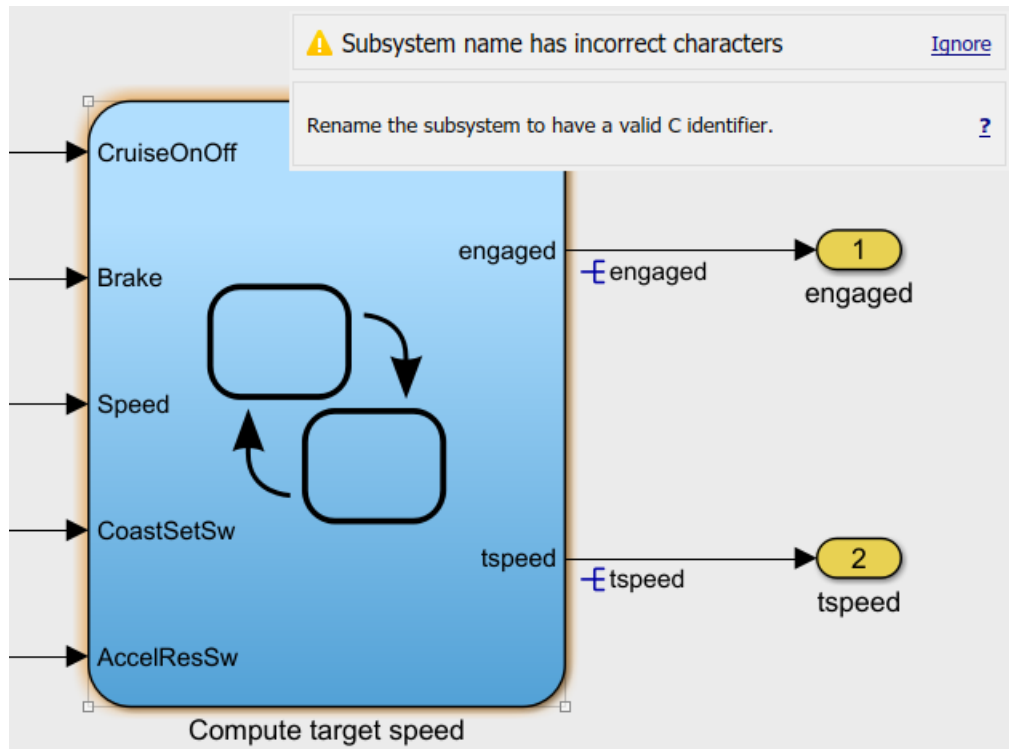
- 1 Create a copy of the example project in a working folder. At the command line, enter:


```
slVerificationCruiseStart
```
- 2 Open the model. At the command line, enter:


```
open_system simulinkCruiseErrorAndStandardsExample
```
- 3 In the model window, turn on edit-time checking by selecting **Analysis > Model Advisor > Display Advisor Checks in Editor**. Alternatively, on the model editor toolbar, select **Display Advisor Checks in Editor** from the **Model Advisor** menu.

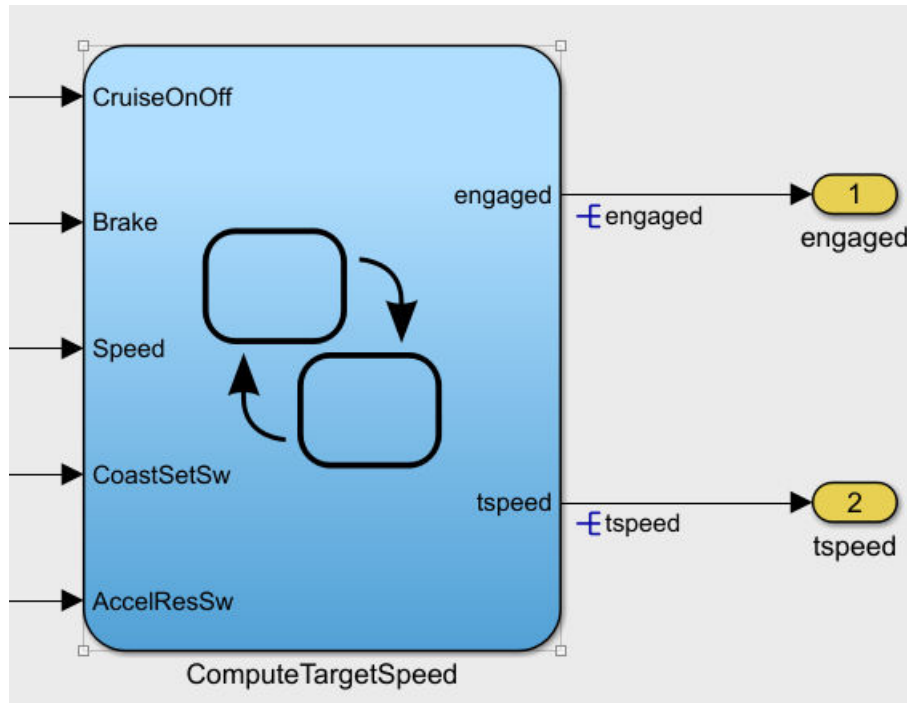


- 4 The highlighted subsystem block indicates a compliance issue. Place your cursor over the highlighted block and click the warning icon. A dialog box provides a description of the warning. For detailed documentation on the check that detected the issue, click the question mark. In this case, the warning indicates that the subsystem block name contains incorrect characters.



- 5 The subsystem block name, `Compute target speed`, contains incorrect spaces. To fix this violation, select `Compute target speed` and replace the name with `ComputeTargetSpeed`.

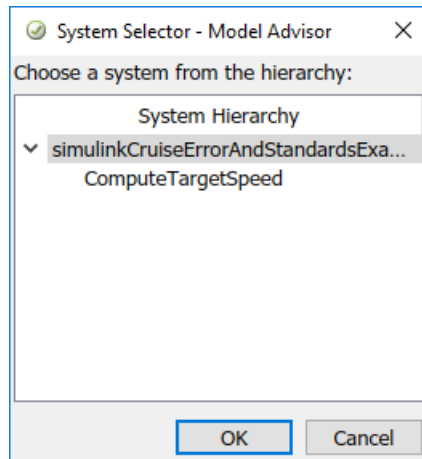
After renaming the block, you do not see a warning icon when you place your cursor over the subsystem block.



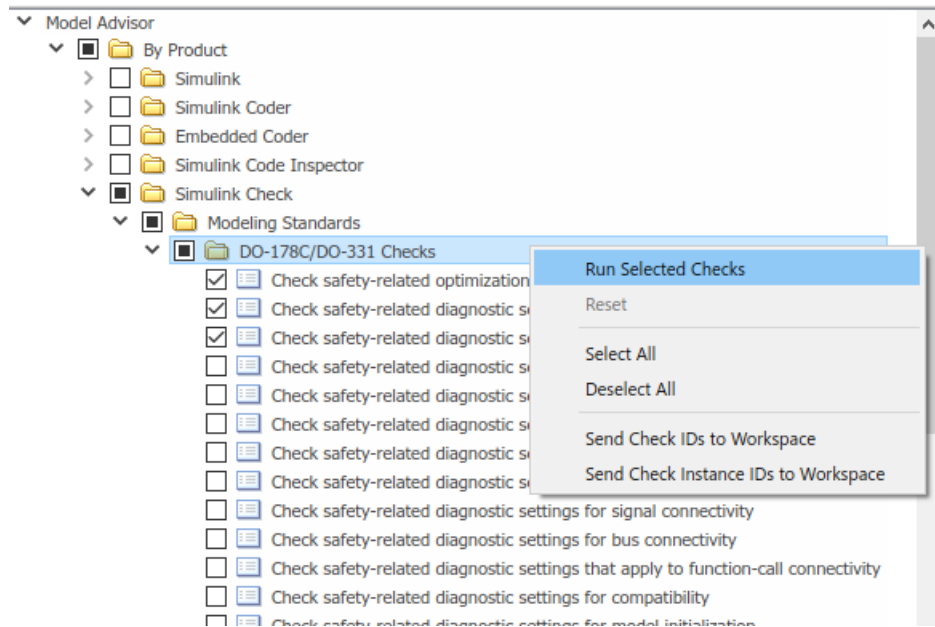
Detect Model Advisor Check Violations Interactively

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

- 1 In the model window, select **Analysis > Model Advisor > Model Advisor**.
- 2 To choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy, click **OK**.

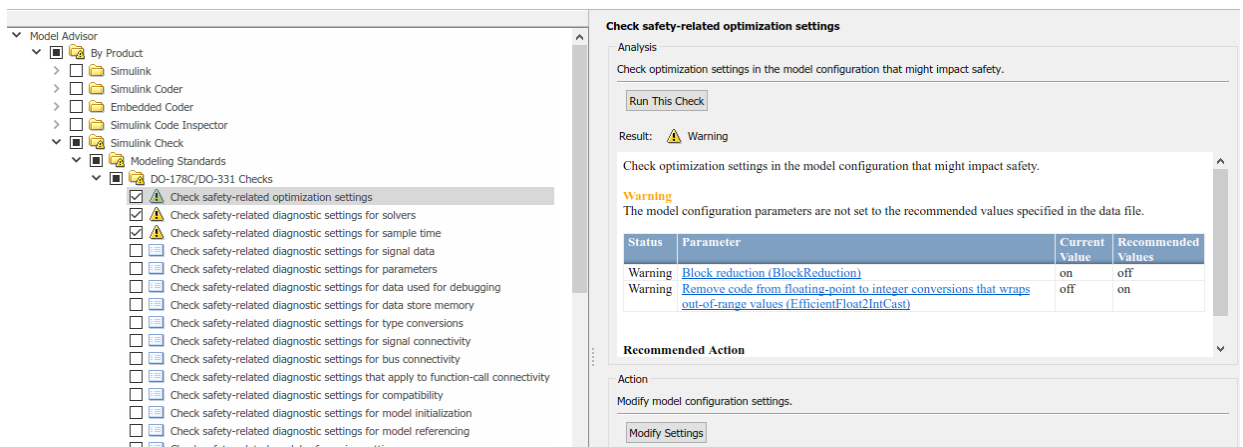


- 3 In the left pane, in the **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks** folder, select:
 - **Check safety-related optimization settings**
 - **Check safety-related diagnostic settings for solvers**
 - **Check safety-related diagnostic settings for sample time**
- 4 Right-click the **DO-178C/DO-331 Checks** node, and then select **Run Selected Checks**.

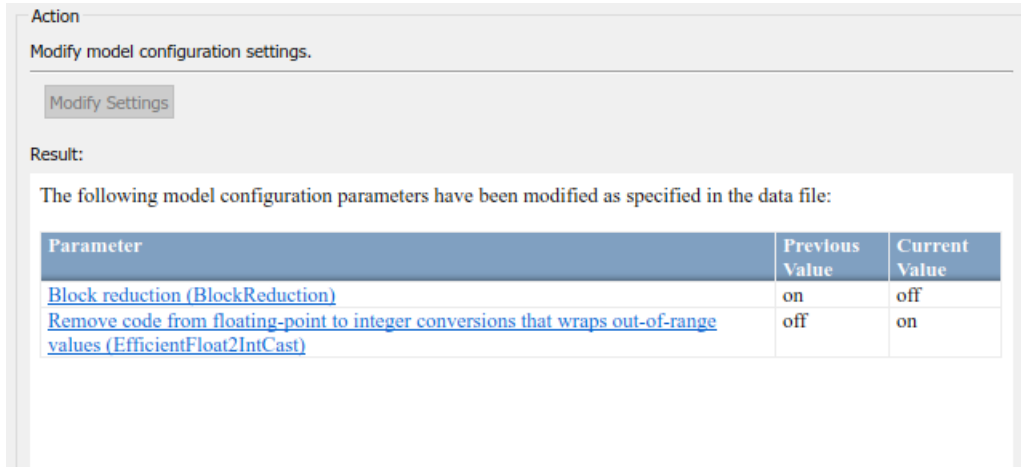


Update Model to Reach Compliance

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



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



- 3 Repeat steps 1 and 2 for the other two checks: **Check safety-related diagnostic settings for solvers** and **Check safety-related diagnostic settings for sample time**.
- 4 To verify that your model now passes, rerun the selected checks.

Display an HTML Report of Check Results

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

See Also

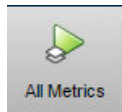
More About

- “Check for Compliance Using the Model Advisor and Edit-Time Checking” on page 3-2
- “Create Model Advisor Checks Workflow” on page 7-2

Collect Model Metric Data by Using the Metrics Dashboard

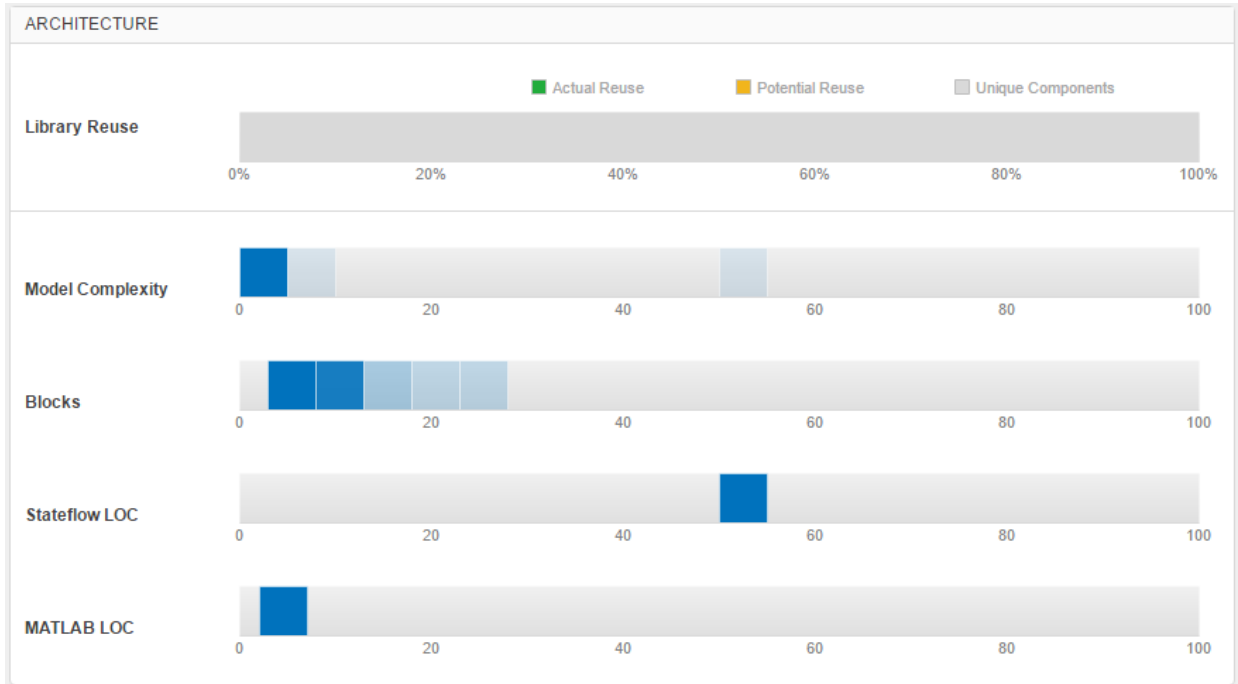
To collect model metric data and assess the design status and quality of your model, use the Metrics Dashboard. The Metrics Dashboard provides a view into the size, architecture, and guideline compliance for your model.

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



The Metrics Dashboard provides a view into the size, architecture, and guideline compliance for your model. In the ARCHITECTURE section of the dashboard, locate the **Model Complexity** widget. This widget is a visual representation of the distribution of complexity across the components in the model hierarchy. For each complexity range, a colored bar indicates the number of components that fall within that range. Darker 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.

1 Getting Started



- 4 To drill into model complexity details at the model, subsystem, and chart level, click anywhere in the **Model Complexity** widget.

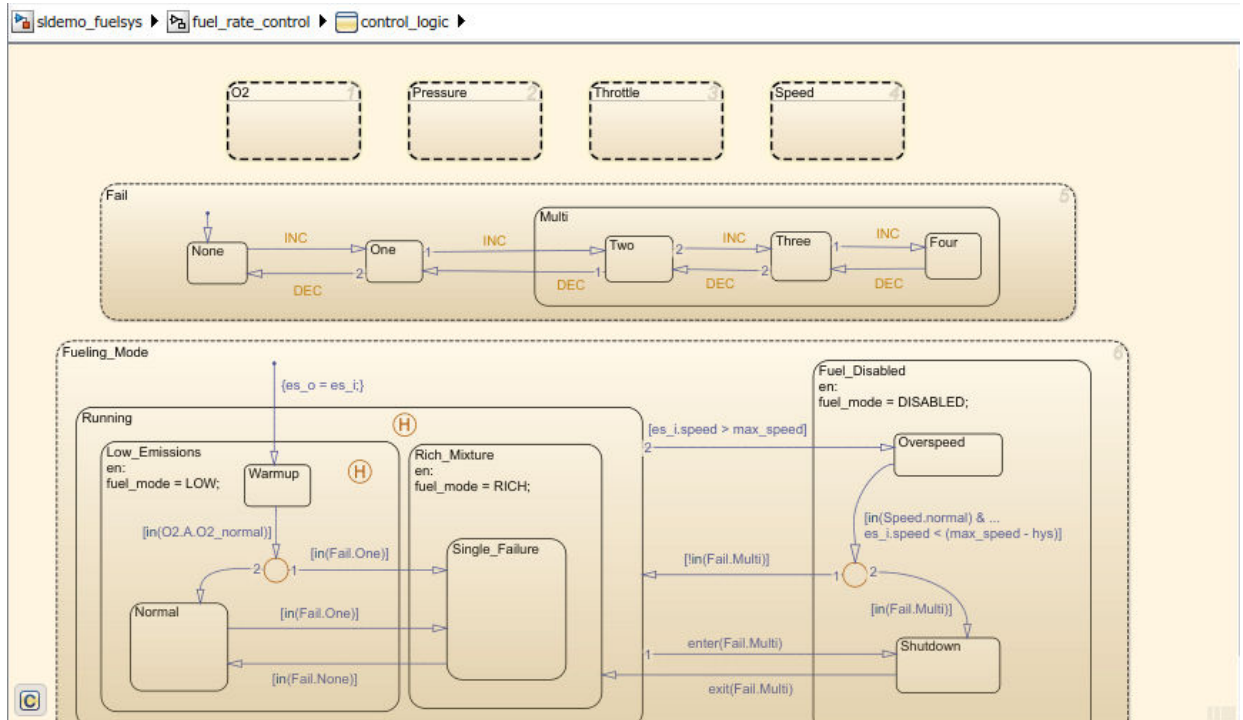
Cyclomatic complexity
Metric that calculates the cyclomatic complexity for model, subsystems and charts.

Type	Component	Path	Qty	Model Complexity	Model Complexity (incl. Descendants)
SubSystem	rich_mode	sldemo_fuelsys/...lc/switchable_compensation/rich_mode	1	0	0
SubSystem	fuel_calc	sldemo_fuelsys/fuel_rate_control/fuel_calc	1	0	4
SubSystem	To Plant	sldemo_fuelsys/To Plant	1	0	0
SubSystem	Engine Gas Dynamics	sldemo_fuelsys/Engine Gas Dynamics	1	0	13
SubSystem	To Controller	sldemo_fuelsys/To Controller	1	0	0
SubSystem	disabled_mode	sldemo_fuelsys/...witchable_compensation/disabled_mode	1	0	0
SubSystem	low_mode	sldemo_fuelsys/...alc/switchable_compensation/low_mode	1	0	0
SubSystem	Dashboard	sldemo_fuelsys/Dashboard	1	0	0
SubSystem	Intake Manifold	sldemo_fuelsys/...Throttle & Manifold/Intake Manifold	1	0	2
SubSystem	Pressure.map_estimate	sldemo_fuelsys/...control_logic/Pressure.map_estimate	1	1	1
SubSystem	airflow_calc	sldemo_fuelsys/fuel_rate_control/airflow_calc	1	1	1
SubSystem	Mixing & Combustion	sldemo_fuelsys/...ine Gas Dynamics/Mixing & Combustion	1	1	3
SubSystem	fuel_rate_control	sldemo_fuelsys/fuel_rate_control	1	1	62

- 5 To sort the components by complexity, click the column header, **Model Complexity**. In this example, the `control_logic` chart is the most complex.

Cyclomatic complexity					
Metric that calculates the cyclomatic complexity for model, subsystems and charts.					
Type	Component	Path	Qty	Model Complexity	Model Complexity (incl. Descendants)
Chart	control_logic	sidemo_fuelsys/fuel_rate_control/control_logic	1	51	56
Model	sidemo_fuelsys		1	5	80
SubSystem	Speed.speed_estimate	sidemo_fuelsys/.../control_logic/Speed.speed_estimate	1	3	3
SubSystem	switchable_compensation	sidemo_fuelsys/.../fuel_calc/switchable_compensation	1	2	2
MATLABFunction	EGO Sensor	sidemo_fuelsys/.../amics/Mixing & Combustion/EGO Sensor	1	2	2
MATLABFunction	MATLAB Function	sidemo_fuelsys/.../fold/Intake Manifold/MATLAB Function	1	2	2
MATLABFunction	f(theta)	sidemo_fuelsys/.../hrottle & Manifold/Throttle/f(theta)	1	2	2
SubSystem	feedforward_fuel_rate	sidemo_fuelsys/.../trol/fuel_calc/feedforward_fuel_rate	1	2	2
SubSystem	Throttle & Manifold	sidemo_fuelsys/.../ine Gas Dynamics/Throttle & Manifold	1	2	10
SubSystem	Throttle	sidemo_fuelsys/.../ynamics/Throttle & Manifold/Throttle	1	2	6
MATLABFunction	g(pratio)	sidemo_fuelsys/.../rottle & Manifold/Throttle/g(pratio)	1	2	2
SubSystem	Mixing & Combustion	sidemo_fuelsys/.../ine Gas Dynamics/Mixing & Combustion	1	1	3
SubSystem	airflow_calc	sidemo_fuelsys/fuel_rate_control/airflow_calc	1	1	1

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



- 7 Once you have used the dashboard to determine the high complexity components in your model, you can reduce complexity by refactoring your model. In this case, refactoring the `control_logic` chart by moving logic into atomic subcharts reduces the complexity for that component.

See Also

More About

- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2
- “Model Metrics”
- “Collect Model Metrics Programmatically” on page 5-18

Refactor Model with Clone Detection and Model Transformer Tools

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

Identify and Replace Clones with Links to Library Blocks

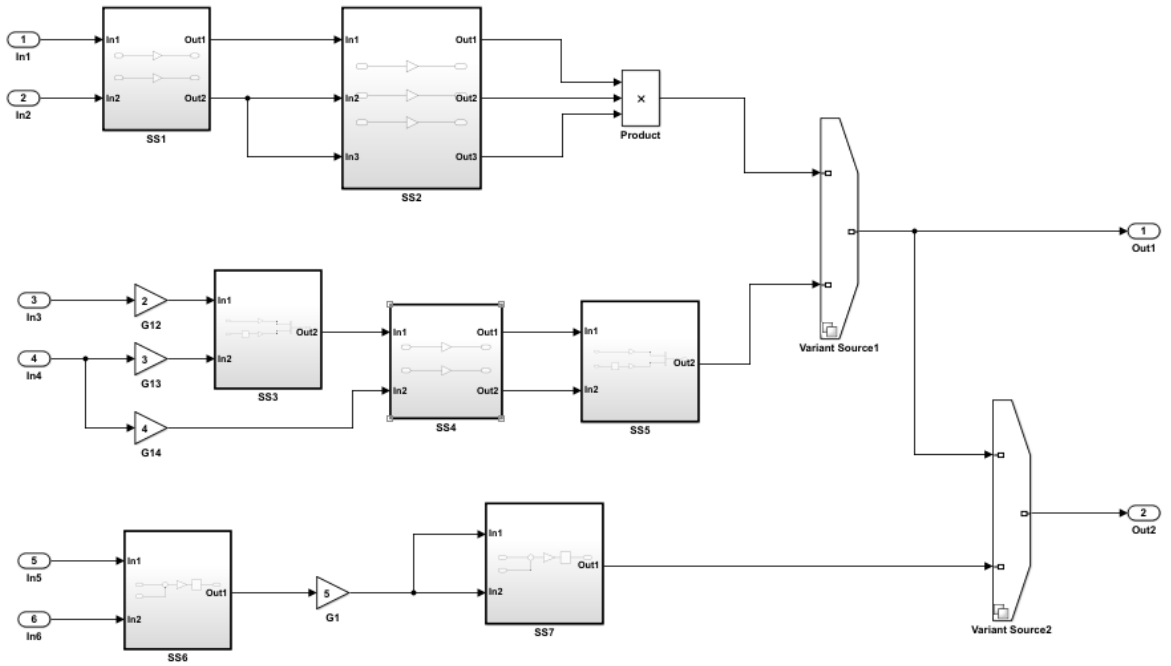
You can use the Identify Modeling Clones tool to enable component reuse by completing these tasks:

- Identify subsystem clones.
 - Create library blocks from clones.
 - Create a model that replaces clones with links to library blocks.
 - Identify similar clones.
- 1 Open the example model `ex_clone_detection` and the corresponding library `ex_clone_library`. At the MATLAB® command line, enter:

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

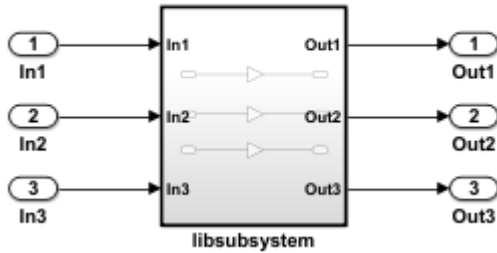
1 Getting Started

ex_clone_detection



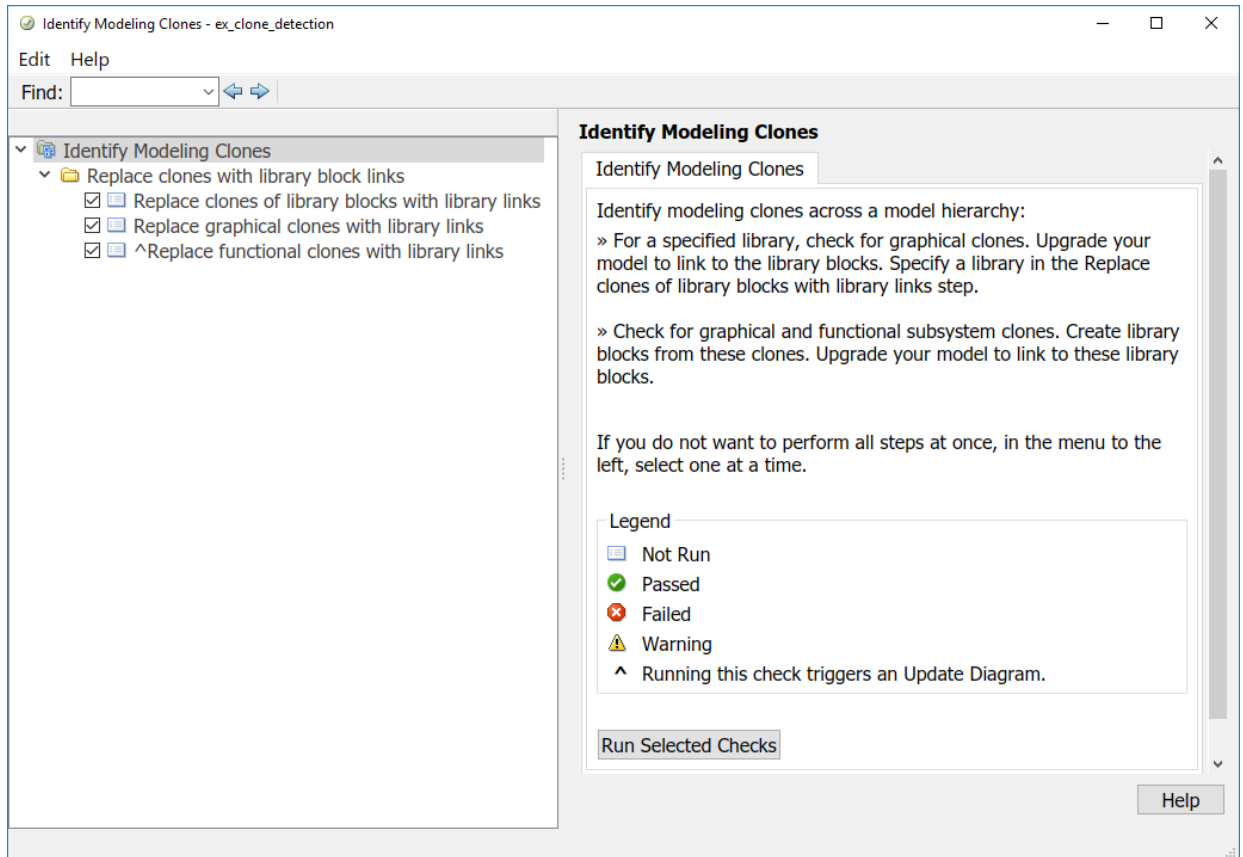
Copyright 2017 The MathWorks Inc.

ex_clone_library



- 2 Save the model and library to the current folder on the MATLAB path.

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



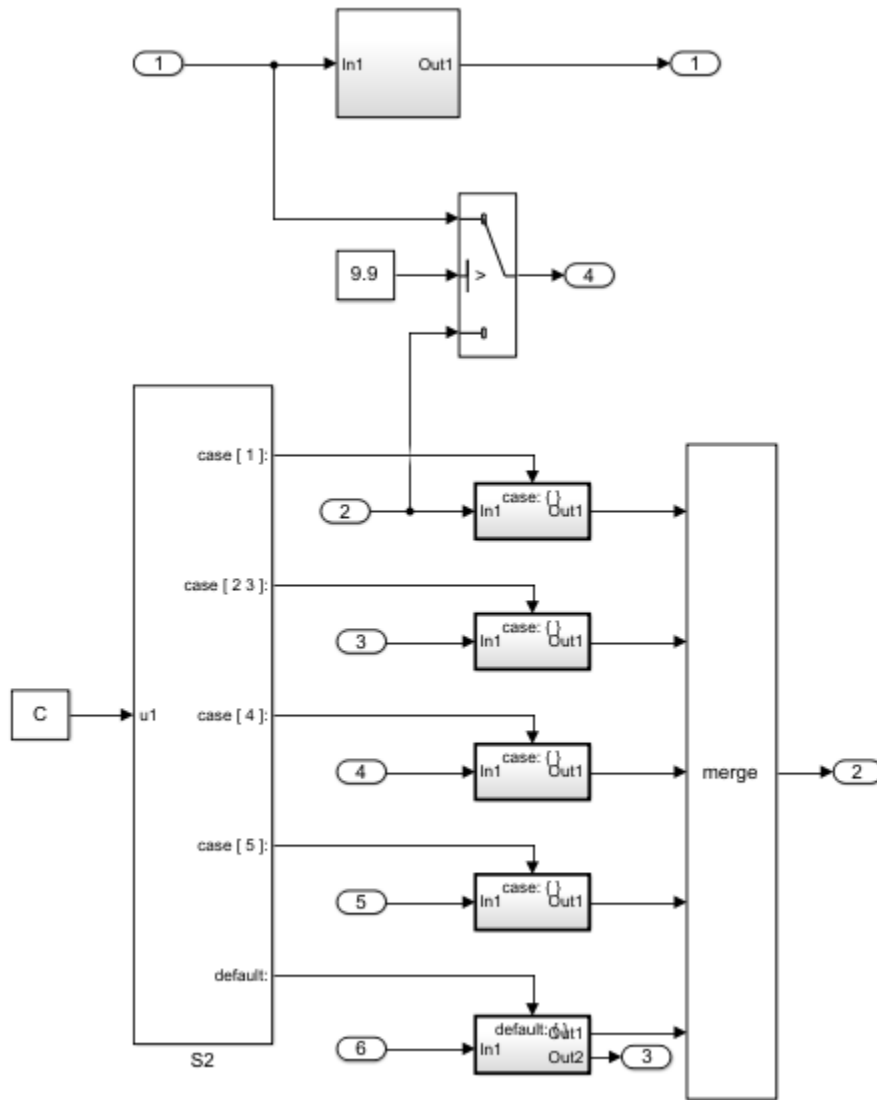
- 4 Open the **Identify Modeling Clones** folder.
- 5 Select **Replace clones of library blocks with library links**. In the **Library file name** field, insert the library name, `ex_clone_library`.
- 6 Select the **Identify Modeling Clones** folder. Then, click **Run Selected Checks**. Because every check is selected by default, the tool identifies all possible clones in the model.
- 7 Select each check. The checks contain hyperlinks to the clones in the model.

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

Replace Qualifying Modeling Patterns with Variant Blocks

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

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

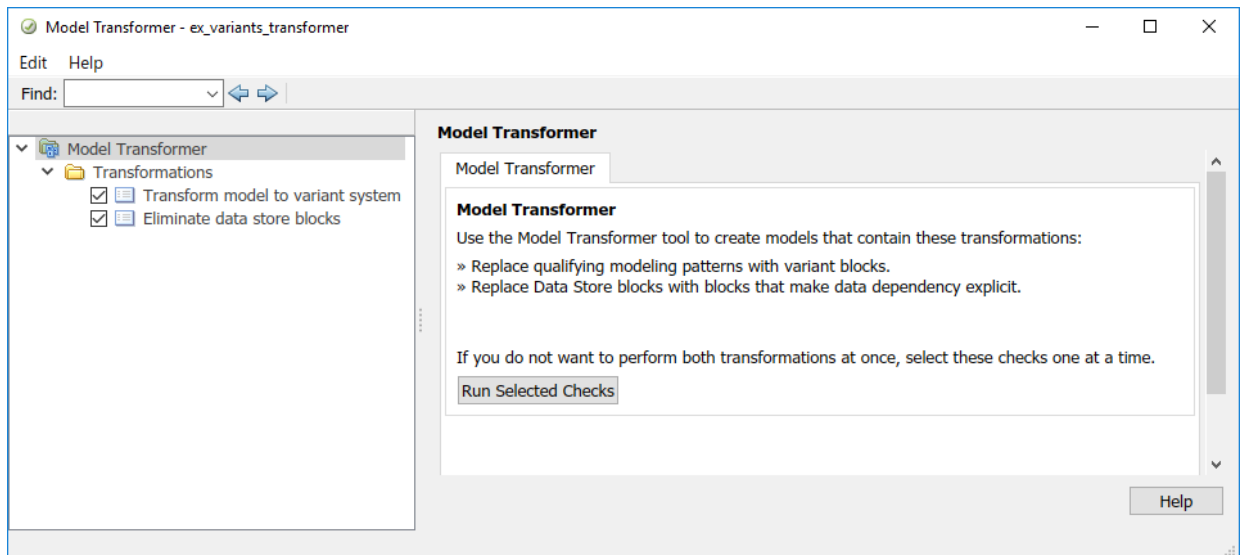


- 1 Open the example model `ex_variants_transformer`.

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

- 2 Save the model to your working folder.
- 3 From the Simulink Editor, open the Model Transformer tool by selecting **Refactor Model > Model Transformer**. Or, in the Command Window, type:

```
mdltransformer('ex_variants_transformer')
```



- 4 Select the **Transform model to variant system** check.
- 5 Click **Run This Check**. The top **Result** table contains a list of system constants that qualify to be part of condition expressions in Variant Source or Variant Subsystem blocks.
- 6 Click **Refactor Model**.
- 7 Your working folder contains a folder called `m2m_ex_variants_transformer`. This folder contains the transformed model `gen0_ex_variants_transformer`.
- 8 The bottom **Result** table contains hyperlinks to the original and transformed models.
- 9 Select the **Eliminate data store blocks** check. You can use this check to replace data stores with blocks that improve model readability by making data dependency explicit. For an example, see “Improve Model Readability by Eliminating Local Data Store Blocks” on page 3-21.

See Also

More About

- “Refactor Models”
- “Enable Component Reuse by Using Clone Detection” on page 3-14
- “Transform Model to Variant System” on page 3-8
- “Improve Model Readability by Eliminating Local Data Store Blocks” on page 3-21

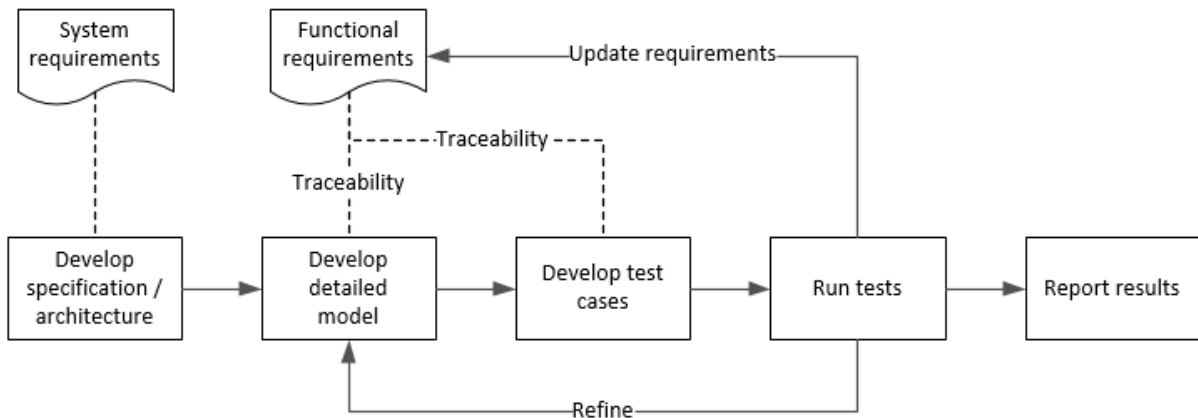
Verification and Validation

- “Test Model Against Requirements and Report Results” on page 2-2
- “Analyze a Model for Standards Compliance and Design Errors” on page 2-6
- “Perform Functional Testing and Analyze Test Coverage” on page 2-9
- “Analyze Code and Test Software-in-the-Loop” on page 2-13
- “Module Verification and Testing Processor-in-the-Loop” on page 2-22
- “Test a Model in Real Time” on page 2-23

Test Model Against Requirements and Report Results

Requirements Overview

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



Test a Cruise Control Safety Requirement

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

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


```
slVerificationCruiseStart
```

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

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

- 3 Open the Test Sequence block.

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

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

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

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

- 4 Open the requirements document. In the Simulink Project window, expand the **documents** folder and open **simulinkCruiseChartReqs.docx**.


- 5 Add links between the test steps and the requirements document.

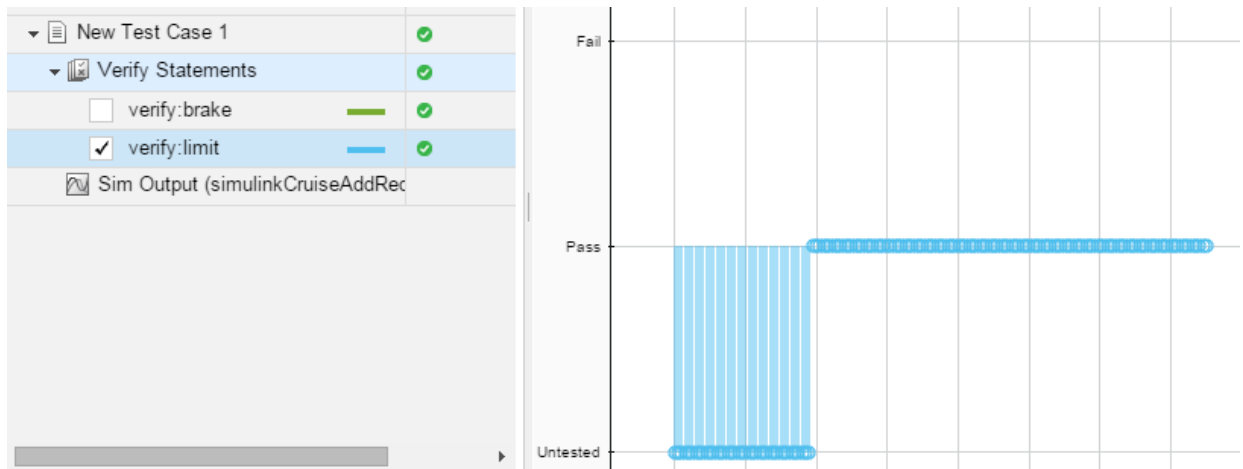
- a In the requirements document, highlight item 3.1, “Vehicle braking will transition system to disengaged (inactive) when engaged (active)”
- b With item 3.1 highlighted, in the test sequence, right-click the BrakeTest step. Select **Requirements traceability > Link to Selection in Word**.
- c In the requirements document, highlight item 3.4, “Transition to disengaged (inactive) when vehicle speed is outside the limits of 20 mph to 90 mph”
- d With item 3.4 highlighted, in the test sequence, right-click the LimitTest step. Select **Requirements traceability > Link to Selection in Word**.
- e Save the requirements document and the model.

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

- a Open the Test Manager. In the Simulink menu, select **Analysis > Test Manager**.
- b In the Test Manager toolstrip, click **New > Test File**. Select the tests folder in the project, and enter a name for the test file. Click **Save**.

A new baseline test is created.

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



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

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

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

See Also

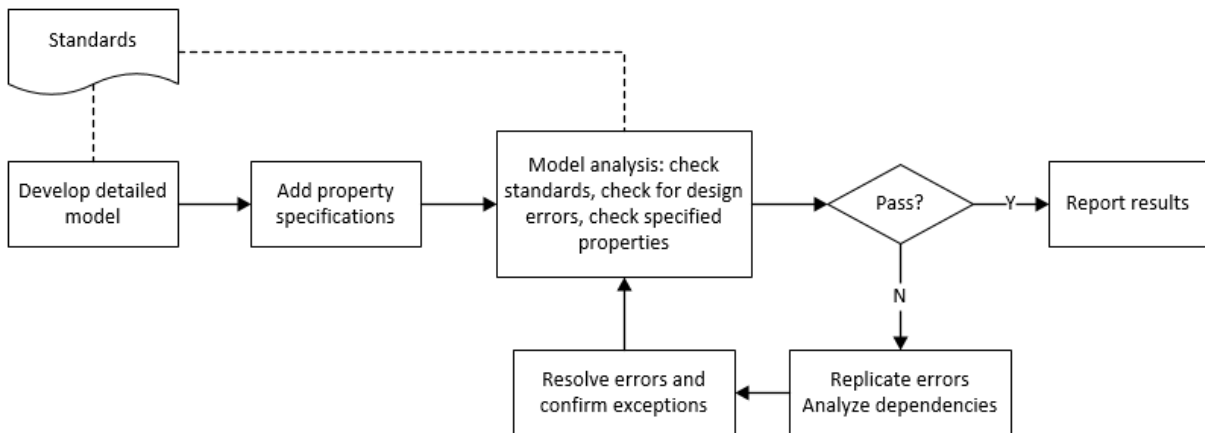
Related Examples

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

Analyze a Model for Standards Compliance and Design Errors

Standards and Analysis Overview

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



Check Model for Style Guideline Violations and Design Errors

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

Check Model for MAAB Style Guideline Violations

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

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

Check Model for Design Errors

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

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

See Also

Related Examples

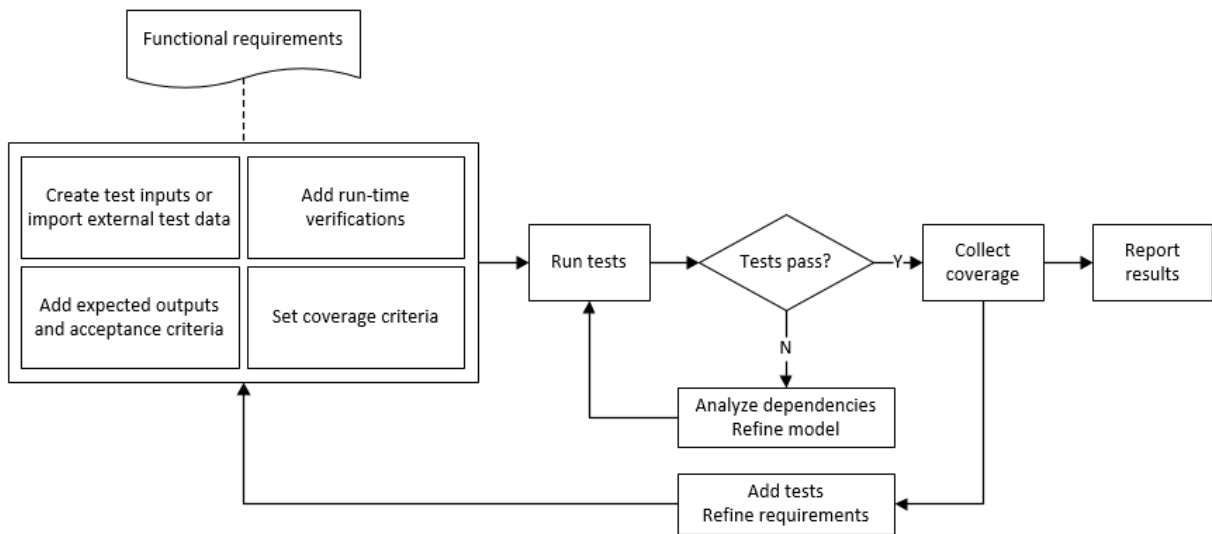
- “Check for Compliance Using the Model Advisor and Edit-Time Checking” on page 3-2
- “Collect Model Metrics Using the Model Advisor” on page 5-8
- “Run a Design Error Detection Analysis” (Simulink Design Verifier)
- “Prove Properties in a Model” (Simulink Design Verifier)

Perform Functional Testing and Analyze Test Coverage

Functional Testing and Coverage Analysis Overview

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

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



Incrementally Increase Test Coverage Using Test Case Generation

This example shows a functional testing-based testing workflow for a cruise control model. You start with a model that has tests linked to an external requirements document, analyze the model for coverage in Simulink 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:

```
slVerificationCruiseStart
```

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

```
open_system simulinkCruiseAddReqExample
```

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

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

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

```
sltest.testmanager.view
```

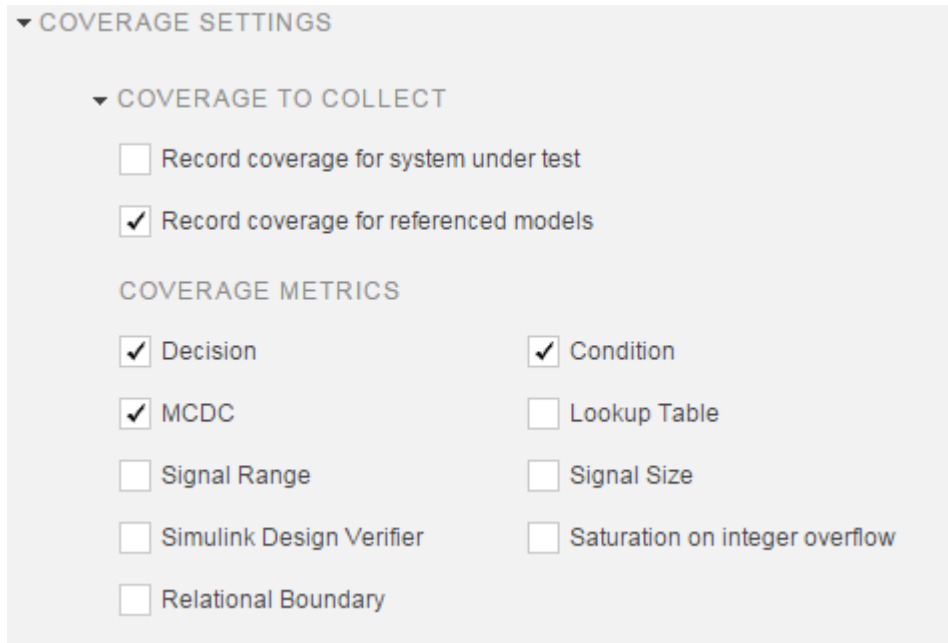
- 4 Open the test sequence block. The sequence tests:

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

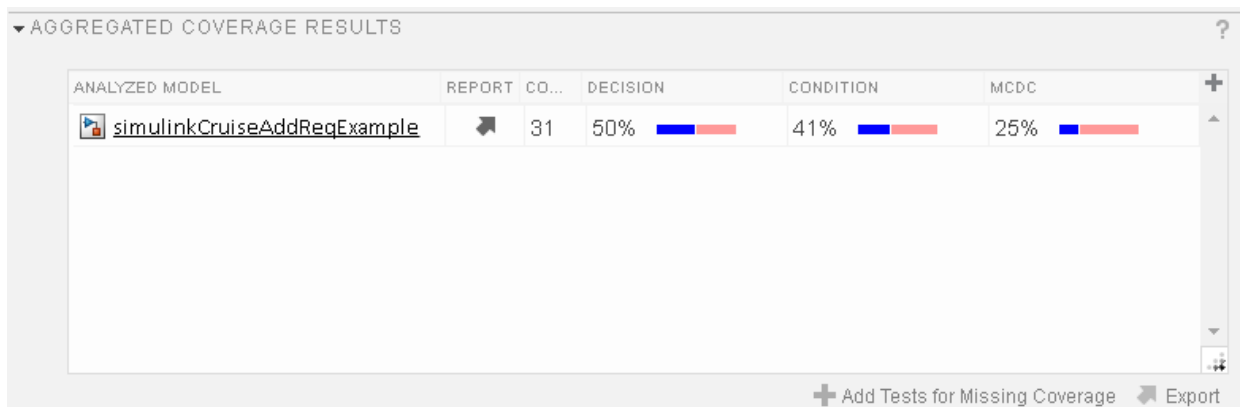
Some test sequence steps are linked to a requirements document `simulinkCruiseChartReqs.docx`.

Measure Model Coverage

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



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



Generate Tests to Increase Model Coverage

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

The **Add Tests for Missing Coverage** options open.

- 3 Under **Harness**, choose Create a new harness.
- 4 Click **OK** to add tests to the test suite using Simulink Design Verifier.
- 5 Run the updated test suite. On the test manager toolstrip, click **Run**. The test results include coverage for the combined test case inputs, achieving increased model coverage.

See Also

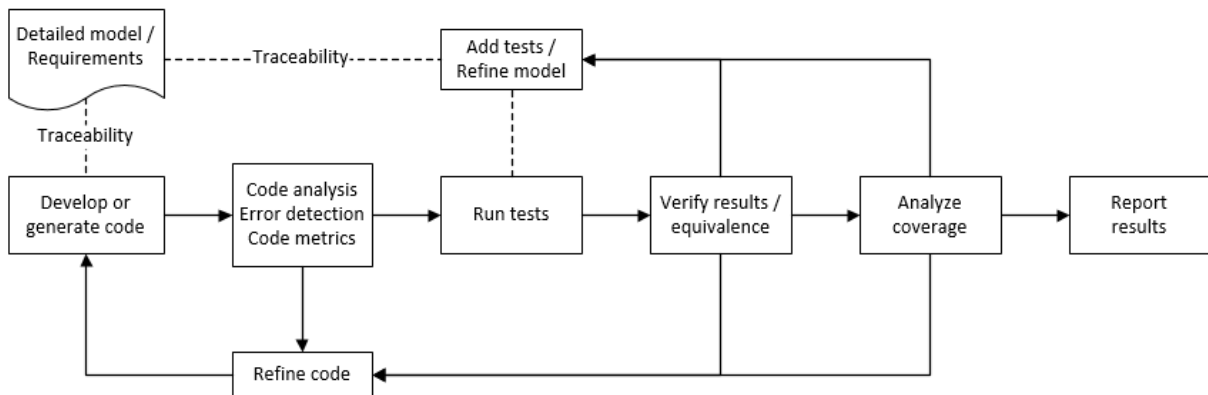
Related Examples

- “Link Tests to Requirements” (Simulink Test)
- “Run-Time Assessments” (Simulink Test)
- “Test Model Output Against a Baseline” (Simulink Test)
- “Highlight Functional Dependencies” (Simulink Design Verifier)
- “Generate Test Cases for Model Decision Coverage” (Simulink Design Verifier)
- “Extend Model Coverage of a Test Case” (Simulink Test)

Analyze Code and Test Software-in-the-Loop

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

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



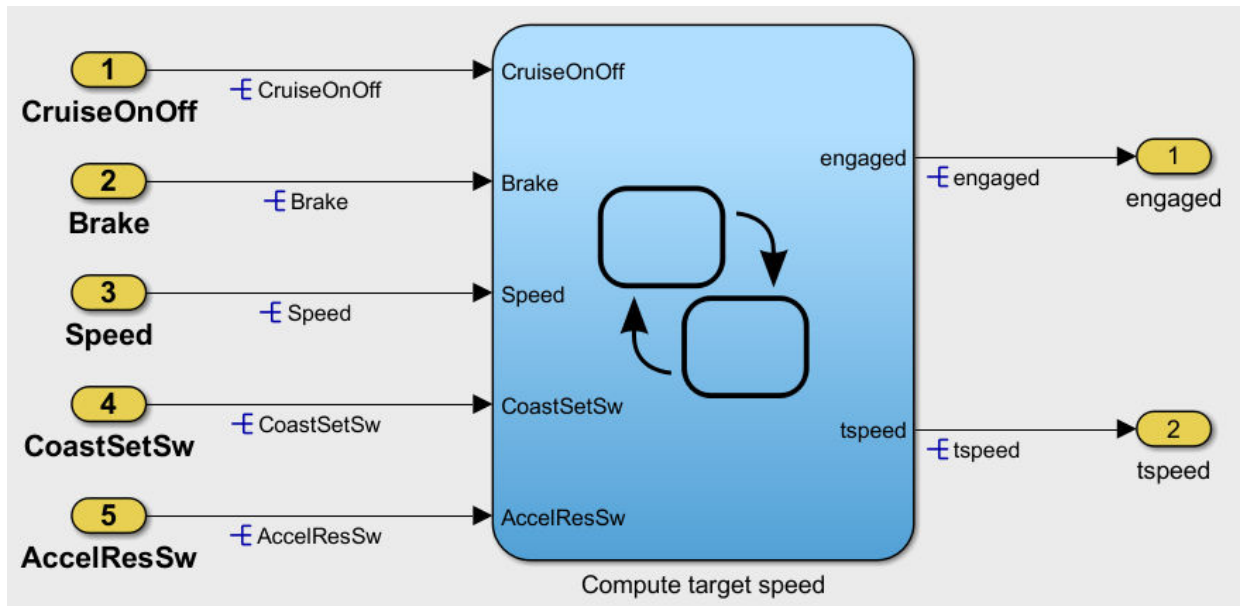
Analyze Code for Defects, Metrics, and MISRA C:2012

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

- 1 Open the Simulink project:

```
slVerificationCruiseStart
```

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

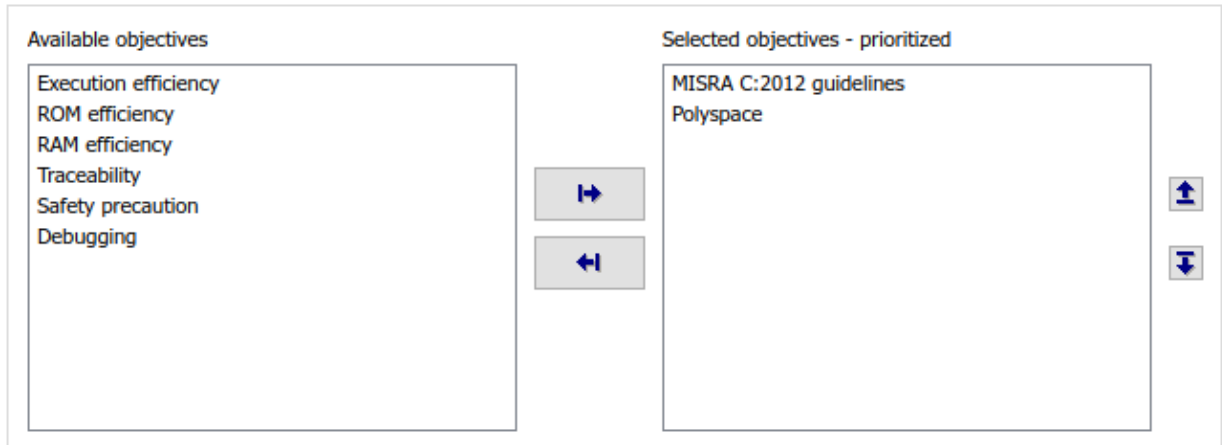


Run Code Generator Checks

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

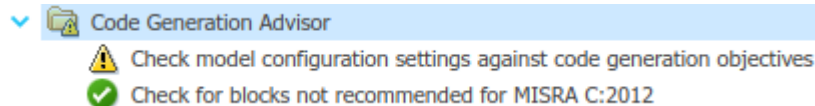
- Right-click `Compute target speed` and select **C/C++ > Code Generation Advisor**.
- Select the Code Generation Advisor folder. Add the Polyspace objective. The MISRA C:2012 guidelines objective is already selected.

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



3 Click **Run Selected Checks**.

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



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

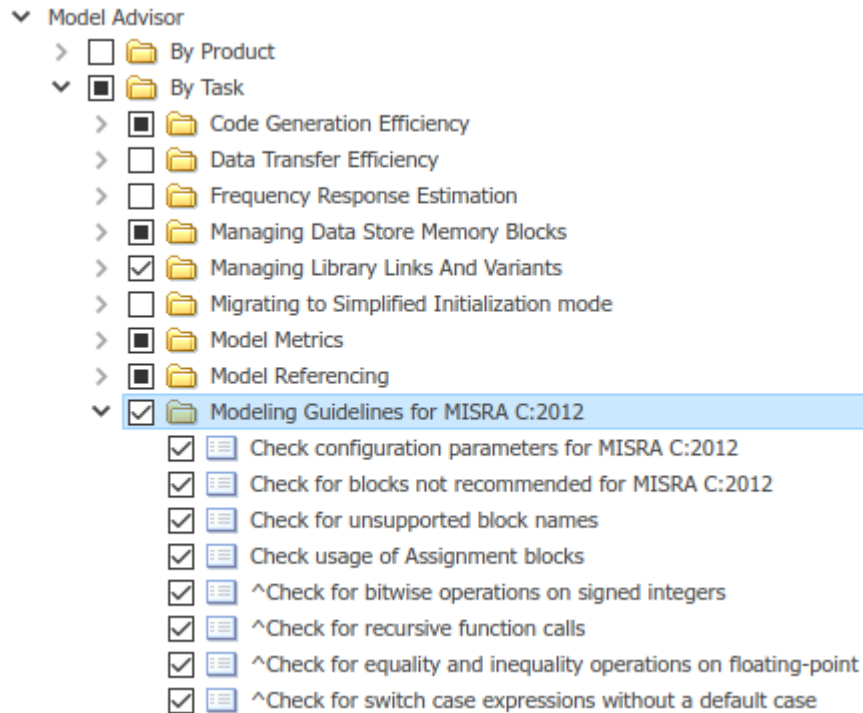
5 Rerun the check by selecting **Run This Check**.

Run Model Advisor Checks

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

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

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



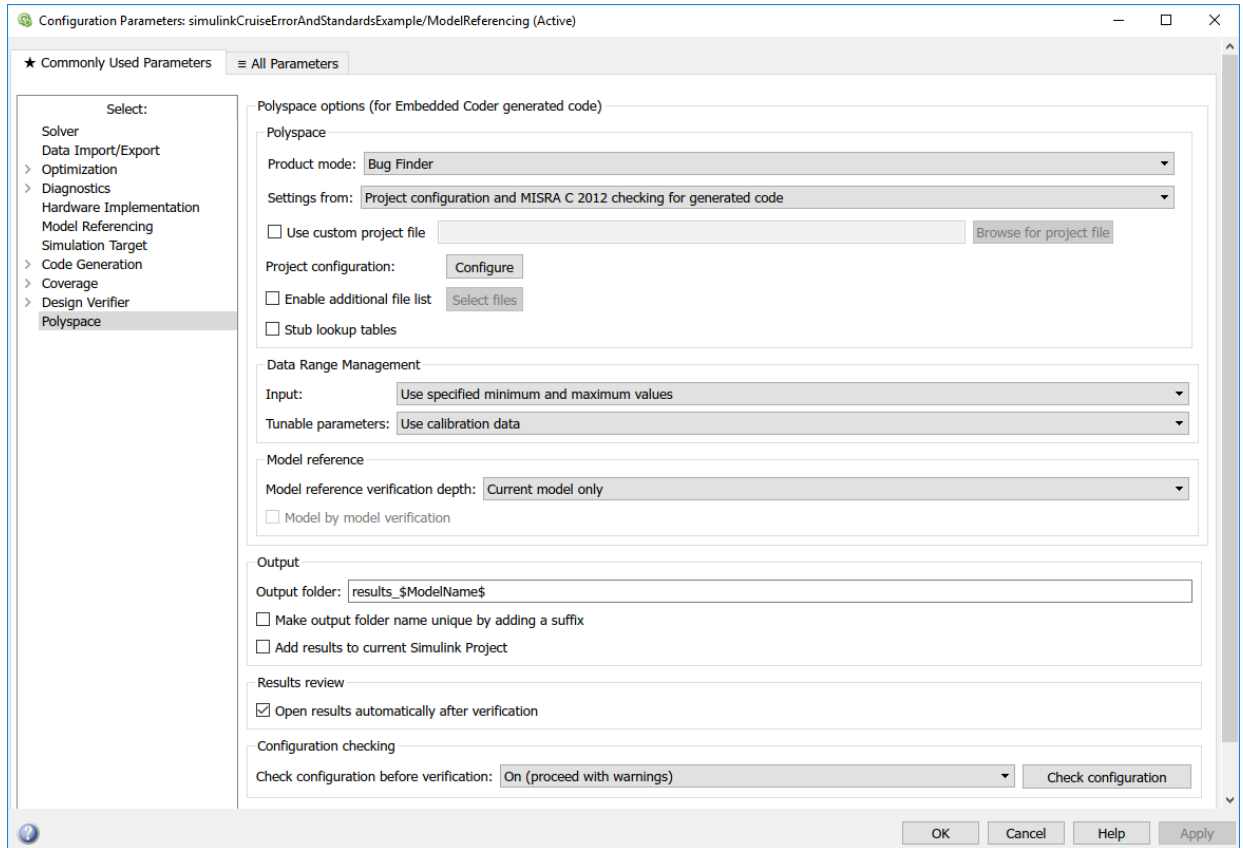
- 3 Click **Run Selected Checks** and review the results.
- 4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

Generate and Analyze Code

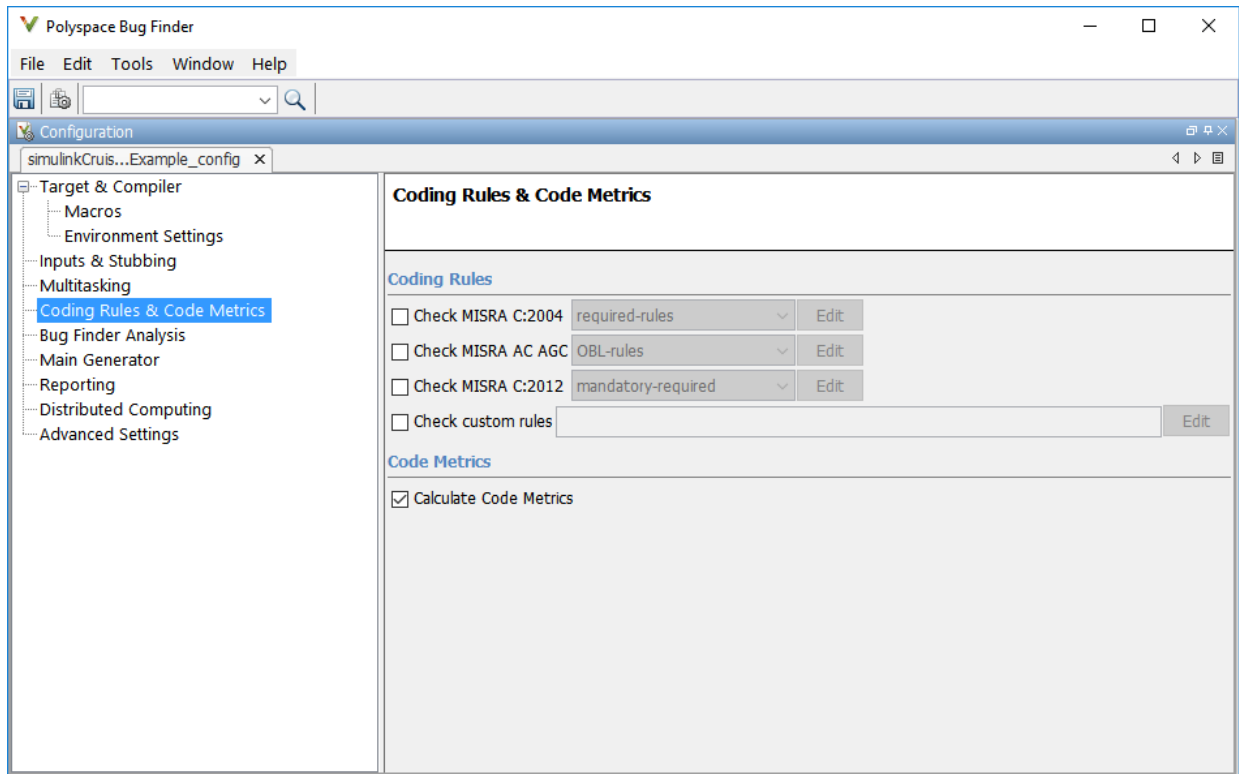
After you have done the model compliance checking, you can now generate code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

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

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



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



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

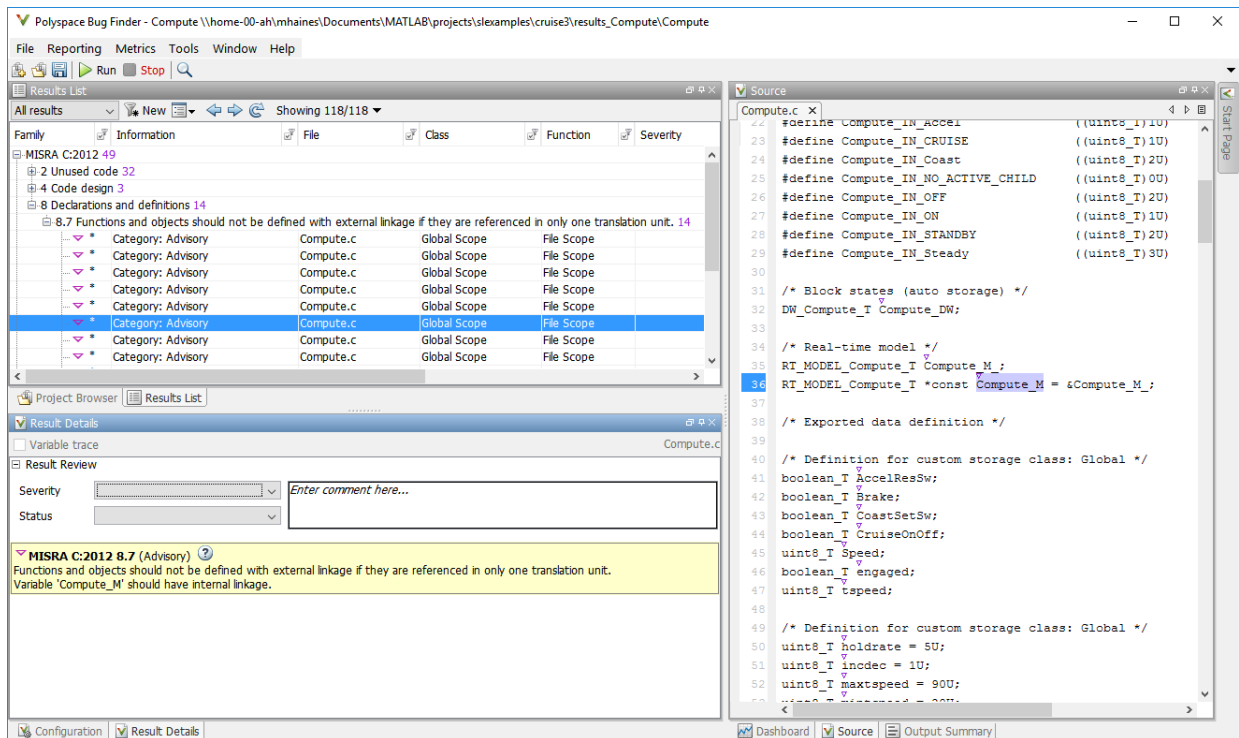
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

Review Results

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

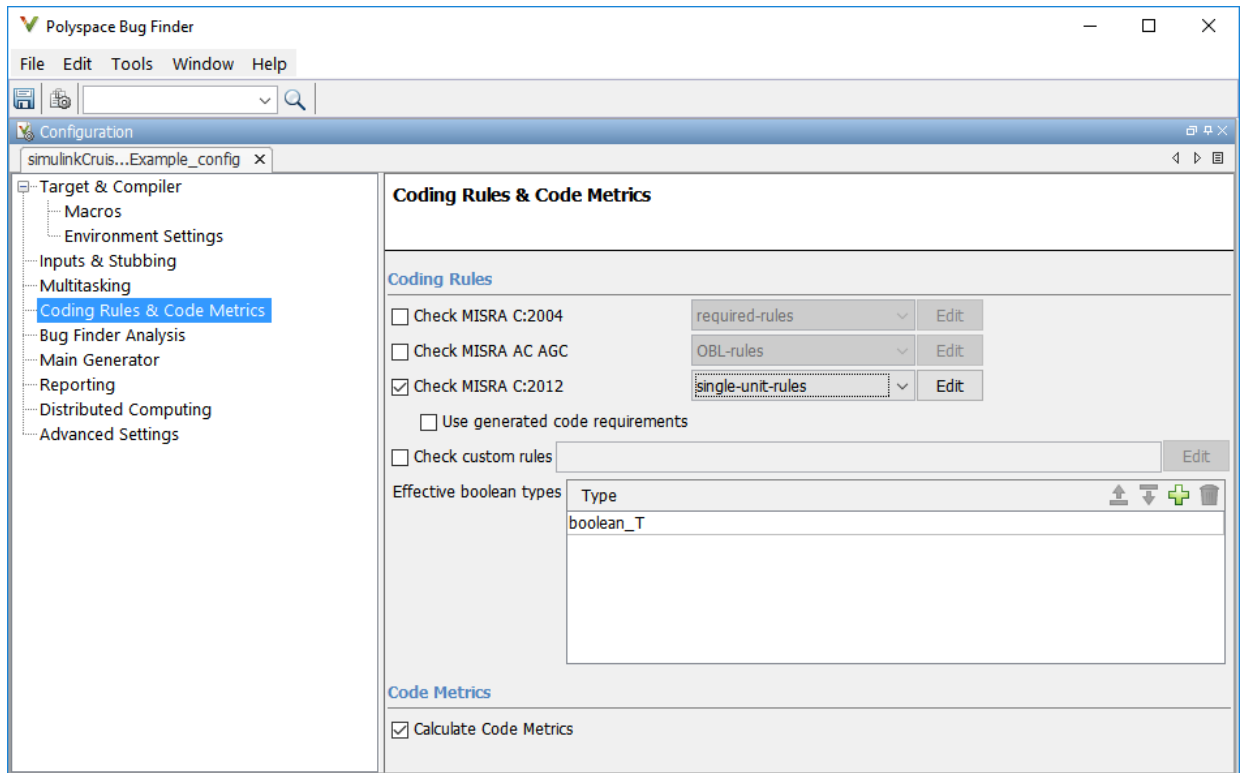
- 1 Expand the tree for rule 8.7 and click through the different results.

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



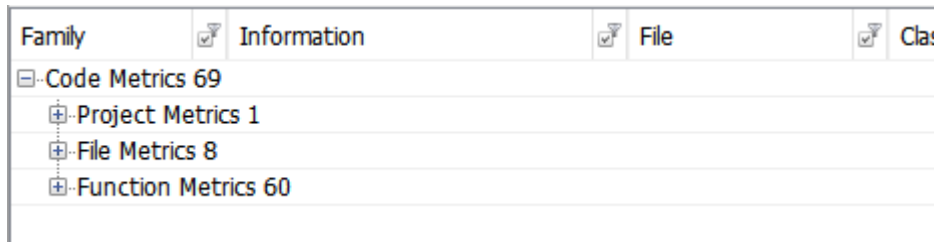
- 2 In your model, right-click Compute target speed and select **Polyspace > Options**.
- 3 Set the **Settings from** (Polyspace Bug Finder) option to **Project configuration**. This option allows you to choose a subset of MISRA rules in the Polyspace configuration.
- 4 Click the **Configure** button.
- 5 In the Polyspace Configuration window, on the **Coding Rules & Code Metrics** pane, select the check box **Check MISRA C:2012** and from the drop-down list, select

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



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

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



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

Generate Report

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

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

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

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

See Also

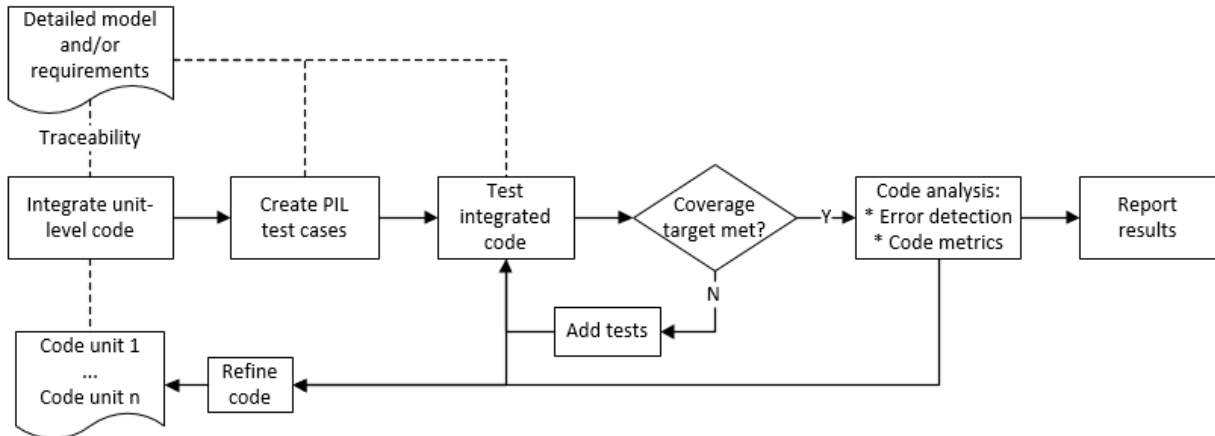
Related Examples

- [“Analyze Generated Code Using Polyspace Bug Finder”](#) (Polyspace Bug Finder)
- [“Test Two Simulations for Equivalence”](#) (Simulink Test)
- [“Export Test Results and Generate Reports”](#) (Simulink Test)

Module Verification and Testing Processor-in-the-Loop

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

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



See Also

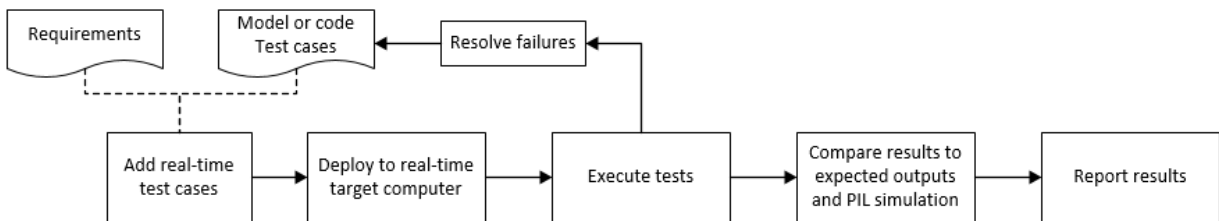
Related Examples

- “Test Two Simulations for Equivalence” (Simulink Test)
- “Analyze Generated Code Using Polyspace Bug Finder” (Polyspace Bug Finder)

Test a Model in Real Time

Real-Time Testing and Testing Production Models Overview

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



See Also

Related Examples

- “Create and Run Real-Time Application from Simulink Model” (Simulink Real-Time)
- “Test Models in Real Time” (Simulink Test)
- “Run-Time Assessments” (Simulink Test)

Checking Systems Interactively

Check for Compliance Using the Model Advisor and Edit-Time Checking

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

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

- Run checks interactively after you complete your model design.
- Configure the Model Advisor to check for violations while you edit.

The Model Advisor reviews your model for conditions and configuration settings that cause inaccurate or inefficient simulation and code generation of the system that the model represents.

Check Your Model Interactively

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

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

For more information	See
Checking model compliance with the DO-178C safety standard	“Model Checks for DO-178C/DO-331 Standard Compliance” on page 3-39
Checking model compliance with the IEC 61508, IEC 62304, ISO 26262, or EN 50182 safety standards	“Model Checks for IEC 61508, IEC 62304, ISO 26262, and EN 50128 Standard Compliance” on page 3-47

For more information	See
Checking model compliance with MathWorks Automotive Advisory Board (MAAB) guidelines	"Model Checks for MathWorks Automotive Advisory Board (MAAB) Guideline Compliance" on page 3-55
Checking model compliance with the MISRA C:2012 standard	"Model Checks for MISRA C:2012 Compliance" on page 3-70
Checking model compliance with CERT C, CWE, and ISO/IEC TS 17961 secure coding standards	"Model Checks for Secure Coding (CERT C, CWE, and ISO/IEC TS 17961 Standards)" on page 3-71
Checking requirements links	"Model Checks for Requirements Links" on page 3-72
Checking model metrics	"Collect Model Metrics Using the Model Advisor" on page 5-8

Check Your Model While You Edit

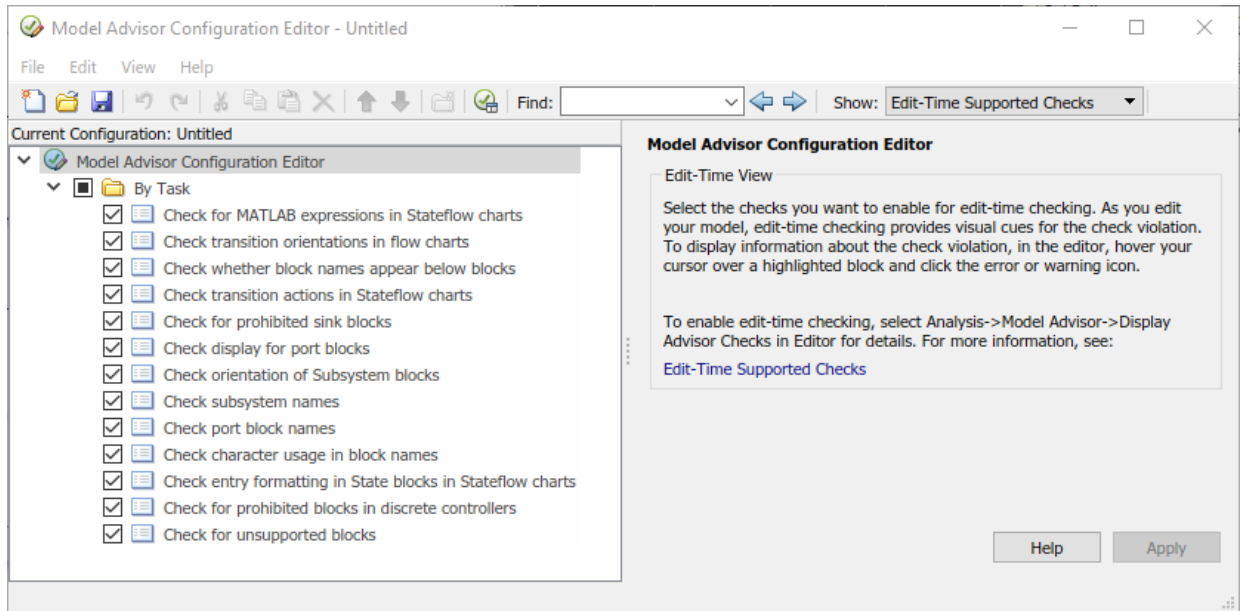
You can identify standards compliance issues earlier in the model design process by using edit-time checking. Edit-time checking provides visual cues for some Model Advisor check violations. In the model editor window, highlighted blocks alert you to issues as you design your model. Hover your cursor over a highlighted block for information about the violation.

Configure Your Model for Edit-Time Checking

To enable edit-time checking for Model Advisor checks, in the model window, select **Analysis > Model Advisor > Display Advisor Checks in Editor**.

Once you select this option, the Model Advisor provides visual cues for several standards compliance issues. To configure the selection and behavior of these checks:

- 1 To open a filtered view of the edit-time checks in the Model Advisor Configuration Editor, in the model window, select **Analysis > Model Advisor > Configure Advisor Edit-Time Checks**.



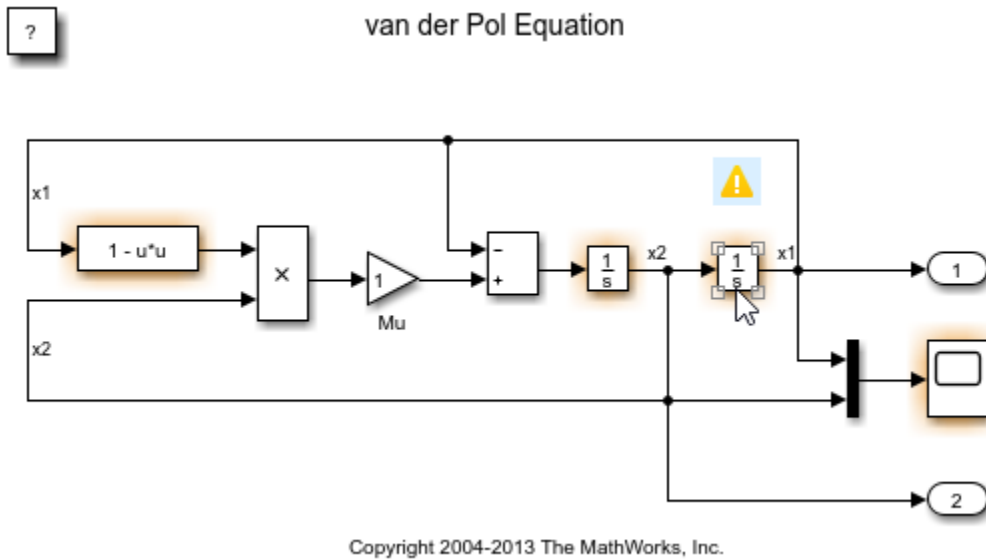
- 2 Use the configuration editor to enable, disable, or customize checks.
- 3 If you have made updates to check selection or behavior, save the current configuration. Then select **File > Set Current Configuration as Default**.

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

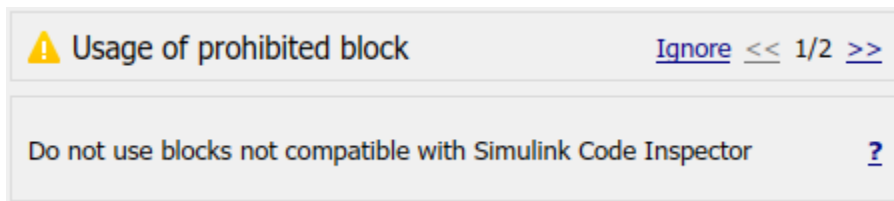
To customize the behavior of edit-time checks, configure updates in the filtered view of edit-time checks in the Model Advisor Configuration Editor. If a check appears in multiple folders of your Model Advisor tree, for edit-time checking, Model Advisor assigns priority to the check in your custom folder.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

Locate Highlighted Compliance Issues in the Model Editor

Once you have configured edit-time checking, as you edit your model, highlighted blocks alert you to compliance issues. Hover your cursor over a highlighted block and click the error or warning icon.



A dialog box provides a description of the warning. For detailed documentation on the check that detected the issue, click the question mark. To ignore the warning for a block and to add the block to the exclusion list for that check, click the **Ignore** button. For a block violating multiple checks, cycle through the edit-time warnings with the << and >> buttons. See “Exclude Checks During Edit-Time” on page 3-32.



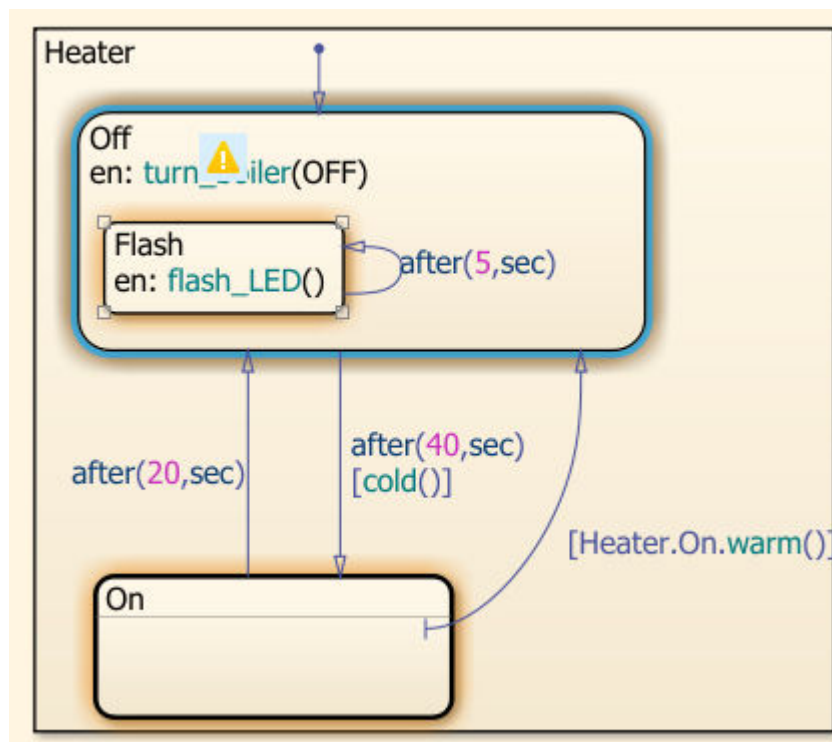
Software is inherently complex and may not be completely free of errors. Model Advisor checks might contain bugs. MathWorks reports known bugs brought to its attention on its Bug Report system at <https://www.mathworks.com/support/bugreports/>. The bug reports are an integral part of the documentation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

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

Check MAAB guideline compliance during Edit-Time for Stateflow

To check your Stateflow chart for compliance with the MAAB guidelines while you edit:

- 1 Open your model that contains Stateflow charts. For example, at the command prompt, open `sf_boiler`.
- 2 To enable the edit-time MAAB checks, go to **Analysis > Model Advisor > Display Advisor checks in editor**.
- 3 Open the **Bang-Bang Controller** chart by double-clicking it. The Model Advisor highlights multiple states. Each highlighted state contains a MAAB warning. Hover your cursor over the warning of the **Off** state to discover the issue.



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

Current guidelines checked for Stateflow charts during edit-time are listed in this table:

Guideline	Check
jc_0501: Format of entries in a State block	Check entry formatting in State blocks in Stateflow charts mathworks.maab.jc_0501
db_0151: State machine patterns for transition actions	Check transition actions in Stateflow charts mathworks.maab.db_0151
db_0132: Transitions in flow charts	Check transition orientations in flow charts mathworks.maab.db_0132
db_0127: MATLAB commands in Stateflow®	Check for MATLAB expressions in Stateflow charts mathworks.maab.db_0127

See Also

Related Examples

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

More About

- “Modeling Rules That Stateflow Detects During Edit Time” (Stateflow)
- “Select and Run Model Advisor Checks” (Simulink)
- “Model Advisor Limitations” (Simulink)

Transform Model to Variant System

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

The Model Transformer can perform these transformations:

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

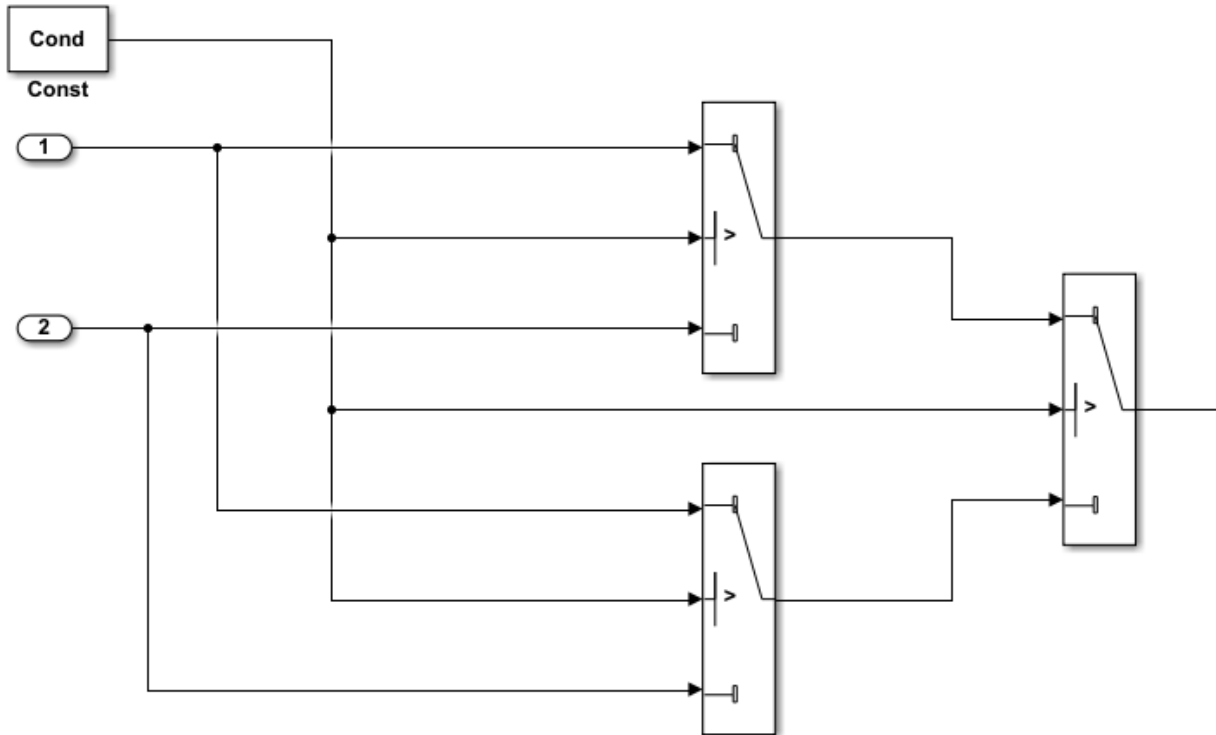
For the Model Transformer tool to perform the transformation, the control input to Multiport Switch or Switch blocks and the inputs to If or Switch Case blocks must be either of the following:

- A Constant block in which the **Constant value** parameter is a `Simulink.Parameter` object of scalar type.
- Constant blocks in which the **Constant value** parameters are `Simulink.Parameter` objects of scalar type and some other combination of blocks that form a supported MATLAB expression. The MATLAB expressions in “Operators and Operands in Variant Condition Expressions” (Simulink) are supported except for bitwise operations.

Example Model

This example shows how to use the Model Transformer to transform a model into a variant system. The example uses the model `rtwdemo_controlflow_opt`. This model

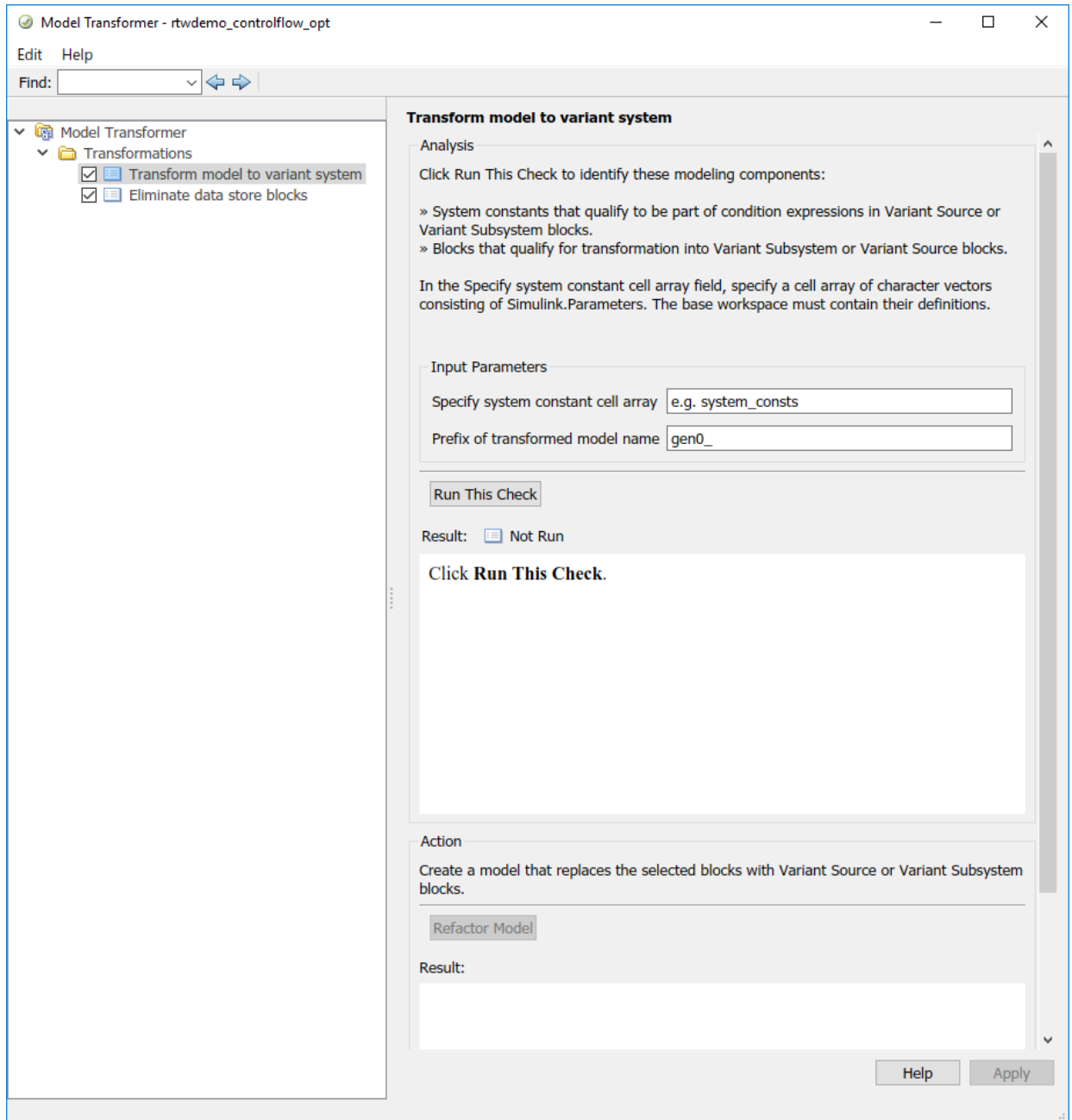
has three Switch blocks. The control input to these Switch blocks is the Simulink.Parameter cond. The Model Transformer dialog box and this example refer to cond as a system constant.



- 1 Open the model. In the Command Window, type `rtwdemo_controlflow_opt`.
- 2 Open the Switch1 Block Parameters dialog box. Change the **Threshold** parameter to 0. The **Threshold** parameter must be an integer because after the variant transformation it is part of the condition expression in the Variant Source block.
- 3 Repeat step 2 for the Switch blocks Switch1, Switch2, and Switch3.
- 4 Save the model to your working folder.

Perform Variant Transform on Example Model

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



- 3 In the **Specify system constant cell array** field, you can specify a cell array of character vectors consisting of `Simulink.Parameters`. The base workspace must contain their definitions.
- 4 In the **Prefix of transformed model name** field, specify a prefix for the model name. If you do not specify a prefix, the default is `gen0`.
- 5 Select **Run This Check**. The Model Transformer lists system constants and blocks that qualify to be part of condition expressions in Variant Source or Variant Subsystem blocks. For the Model Transformer to list a system constant, it must be a `Simulink.Parameter` object of scalar type. For this example, `Cond` qualifies to part of a condition expression.
- 6 If you do not want one of the transformations to occur, you can clear the check box next to it.
- 7 Select **Refactor Model**. The Model Transformer provides a hyperlink to the transformed model and hyperlinks to the corresponding blocks in the original model and the transformed model.

The transformed model or models are in the folder that has the prefix `m2m` plus the original model name. For this example, the folder name is `m2m_rtwdemo_controlflow_opt`.

- 8 In the original model `rtwdemo_controlflow_opt`, right-click one of the Switch blocks. In the menu, select **Model Transformer > Traceability to Transformed Block**. In the transformed model `gen0_rtwdemo_controlflow_opt`, the corresponding Variant Source block is highlighted.
- 9 In the transformed model `gen0_rtwdemo_controlflow_opt`, right-click one of the Switch blocks. In the menu, select **Model Transformer > Traceability to Original Block**. In the original model `rtwdemo_controlflow_opt`, the corresponding Switch block is highlighted.

Model Transformation Limitations

The Model Transformer tool has these limitations:

- In order to run the Model Transformer on a model, you must be able to simulate the model.
- If an If Action Subsystem block drives a Merge block, and the Merge block has another inport that is either unconnected or driven by another conditional subsystem, the Model Transformer does not add a Variant Source block. This modeling pattern produces a warning and an excluded candidate message.

- The Model Transformer cannot perform a variant transformation for every modeling pattern. This list contains some exceptions:
 - The model contains a protected model reference block.
 - A model contains a Variant Source block with the **Analyze all choices during update diagram and generate preprocessor conditionals** parameter set to off.
- After you run one or more tasks, you cannot rerun the tasks because the **Run this Task** and **Run All** buttons are deactivated. If you want to rerun a task, reset the Model Transformer by right-clicking **Model Transformer** and selecting **Reset**.
- Do not change a model in the middle of a transformation. If you want to change the model, close the **Model Transformer**, modify the model, and then reopen the **Model Transformer**.
- For the hyperlinks in the Model Transformer to work, you must have the model to which the links point to open.

See Also

Related Examples

- “Variant Systems” (Simulink)

Enable Component Reuse by Using Clone Detection

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

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

Exact Clones Versus Similar Clones

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

Exact clones and similar clones can have these differences:

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

To detect only exact clones, for each check in the Identify Modeling Clones tool, set the **Maximum number of different parameters** to 0 (default value). Increasing this parameter value increases the number of similar clones that the tool can potentially detect.

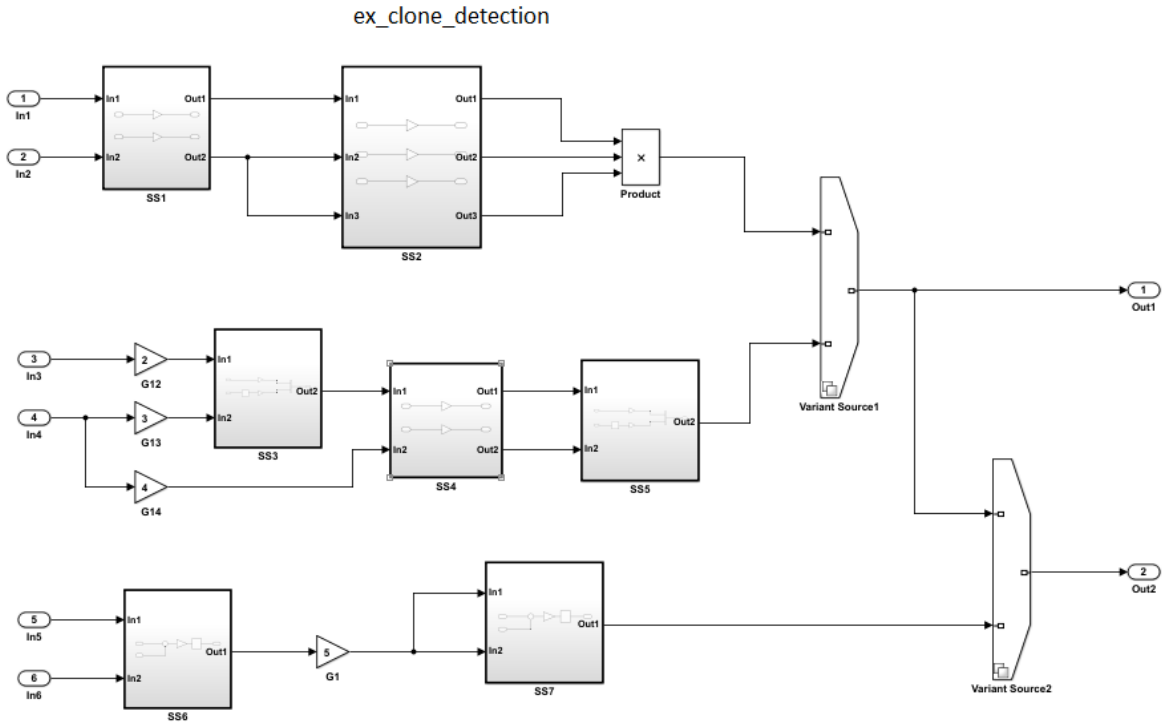
After you identify clones, you can replace them with links to library blocks. Similar clones link to masked library subsystems.

Identify Exact and Similar Clones

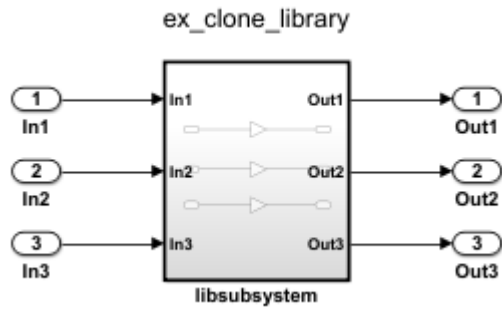
This example shows how to use the Identify Modeling Clones tool to identify exact clones and similar clones, and then replace them with links to library blocks.

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

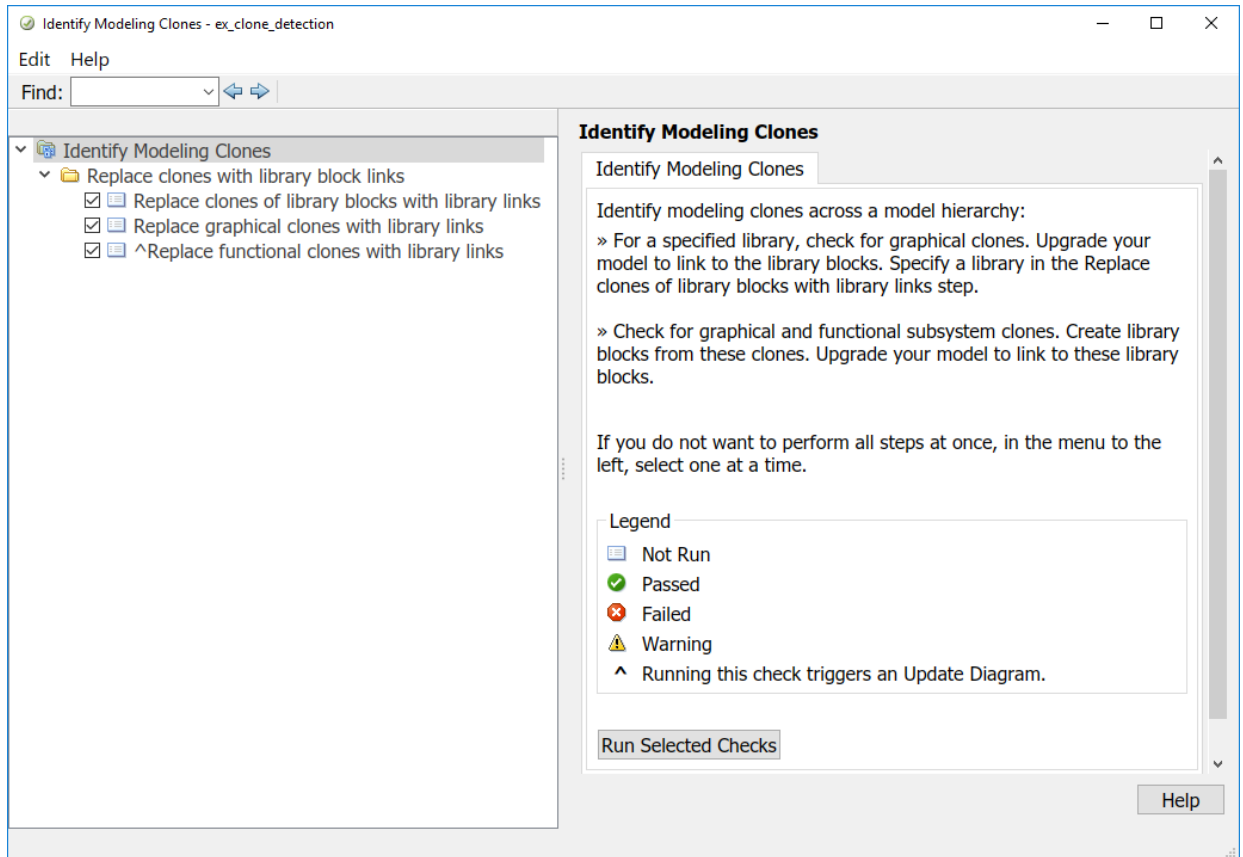
```
addpath(fullfile(docroot,'toolbox','simulink','examples'))
ex_clone_detection
ex_clone_library
```



Copyright 2017 The MathWorks Inc.



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



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

Replace Clones of Library Blocks with Library Links

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

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

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

Replace Graphical Clones with Library Links

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

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

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

Replace Functional Clones with Library Links

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

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

Save and View Clone Detection Reports

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

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

Additional Information

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

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

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

See Also

Related Examples

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

Improve Model Readability by Eliminating Local Data Store Blocks

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

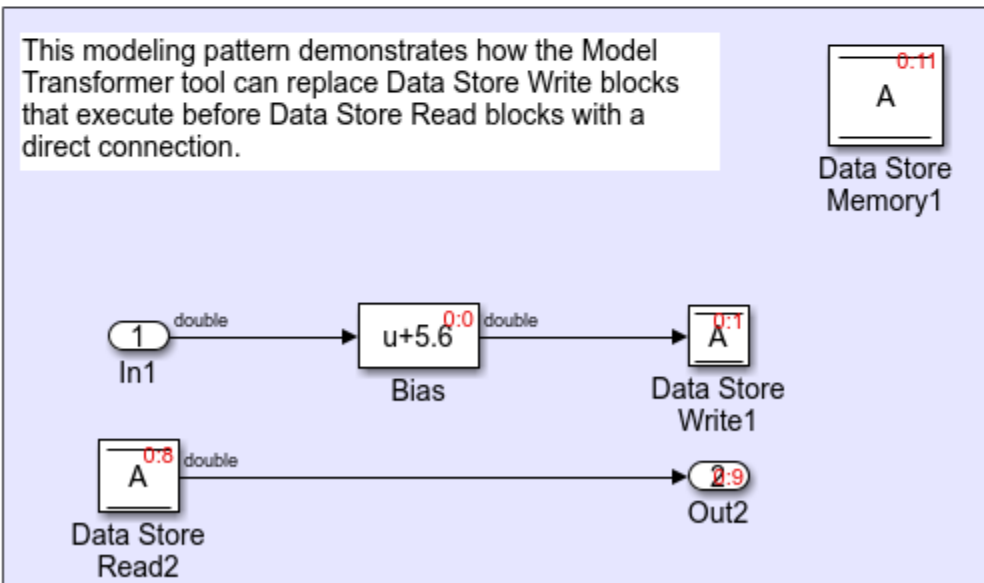
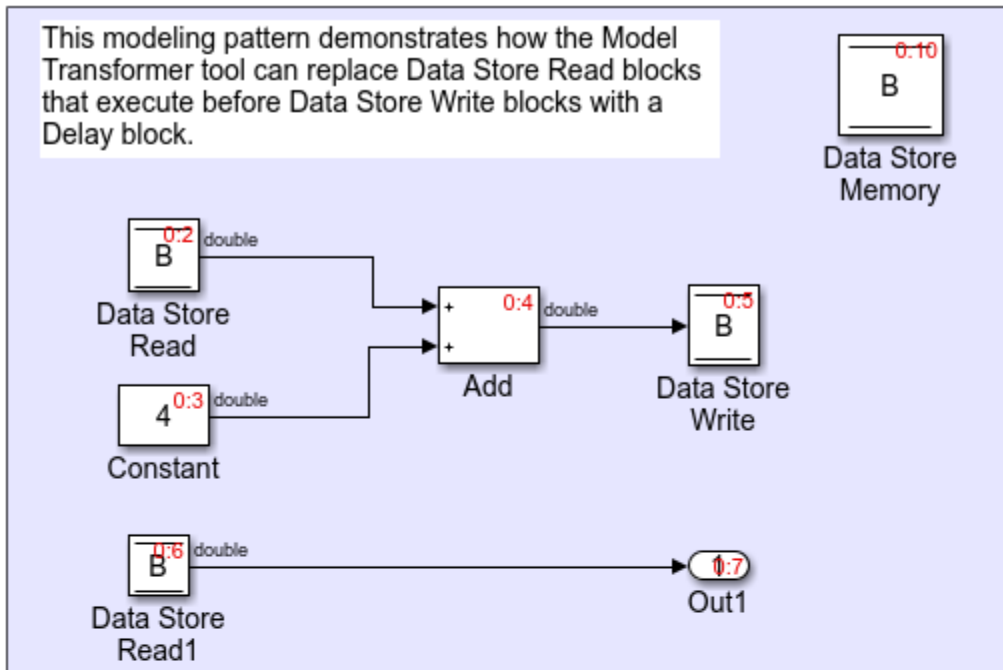
The Model Transformer can replace these data stores:

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

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

Example Model

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



Replace Data Store Blocks

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

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

```
addpath(fullfile(docroot,'toolbox','simulink','examples'))  
ex_data_store_elimination
```
- 2 Save the model to your working folder.
- 3 In the Simulink Editor, from the **Analysis** menu, select **Refactor Model > Model Transformer**. To open the Model Transformer programmatically, at the MATLAB command prompt, type this command:

```
mdltransformer('ex_data_store_replacement').
```
- 4 In the **Transformations** folder, select the **Eliminate data store blocks** check.
- 5 In the **Prefix of refactored model** field, specify a prefix for the refactored model.
- 6 Click the **Run This Check** button. The top **Result** table contains hyperlinks to the Data Store Memory blocks and the corresponding Data Store Read and Data Store Write blocks that qualify for elimination.
- 7 Click the **Refactor Model** button. The bottom **Result** table contains a hyperlink to the new model. The tool creates an `m2m_ex_data_store_replacement` folder. This folder contains the `gen_ex_data_store_replacement.slx` model.

The screenshot shows the 'Model Transformer - ex_data_store_elimination' window. The left sidebar shows a tree view with 'Model Transformer' expanded to 'Transformations', where 'Eliminate data store blocks' is selected. The main area is titled 'Eliminate data store blocks' and contains the following sections:

- Analysis:** A text box explaining the purpose: 'Identify data stores that are eligible for elimination. Deselect blocks that you do not want to eliminate from your model.'
- Input Parameters:** A text box for 'Prefix of refactored model' containing the value 'gen_'.
- Run This Check:** A button that has been clicked.
- Result:** A green checkmark and the text 'Passed'.
- Table:** A table with three columns: 'Data store memory block', 'Data store access blocks', and 'Sorted execution order'. It lists two memory blocks and their associated access blocks with execution order values.
- Action:** A text box explaining the next step: 'Create a model that replaces the selected blocks with blocks that improve model readability by making data dependency explicit. The new model name contains the prefix that you specify in the Prefix of model name field plus the original model name.'
- Refactor Model:** A button that has been clicked.
- Result:** A section titled 'Transformed model' with a hyperlink to 'gen_ex_data_store_elimination'.

At the bottom right of the window are 'Help' and 'Apply' buttons.

	Data store memory block	Data store access blocks	Sorted execution order
<input checked="" type="checkbox"/>	.../Data Store Memory	.../Data Store Read	0:2
		.../Data Store Write	0:5
		.../Data Store Read1	0:6
<input checked="" type="checkbox"/>	.../Data Store Memory1	.../Data Store Write1	0:1
		.../Data Store Read2	0:8

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

Limitations

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

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

See Also

Related Examples

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

Limit Model Checks

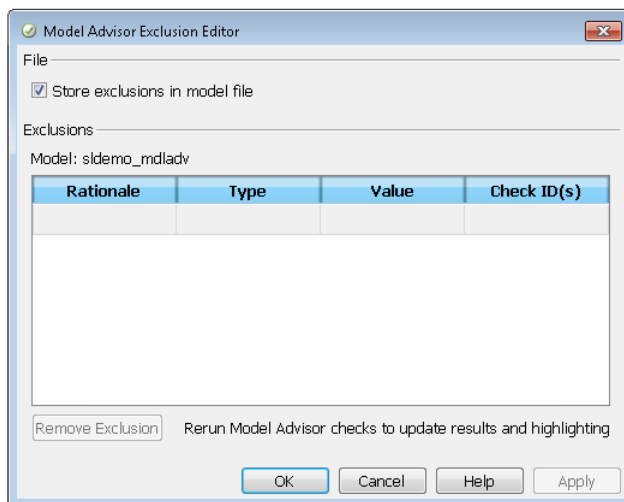
What Is a Model Advisor Exclusion?

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

- Simulink blocks
- Stateflow® charts

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

To view exclusion information for the model, right-click in the model window or right-click a block and select **Model Advisor > Open Model Advisor Exclusion Editor**. The Model Advisor Exclusion Editor dialog box includes the following information for each exclusion.



Field	Description
Rationale	A description of why this object is excluded from Model Advisor checks. The rationale field is the only field that you can edit.
Type	Whether a specific block is excluded or all blocks of a given type are excluded.
Value	Name of excluded block or blocks.
Check ID (s)	Names of checks for which the block exclusion applies.

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

Save Model Advisor Exclusions in a Model File

To save Model Advisor exclusions to the model SLX file, in the Model Advisor Exclusion Editor dialog box, select **Store exclusions in model file**. When you open the model SLX file, the model contains the exclusions.

Save Model Advisor Exclusions in Exclusion File

A Model Advisor exclusion file specifies the collection of blocks to exclude from specified checks in an exclusion file. You can create exclusions and save them in an exclusion file. To use an exclusion file, in the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**. The **Exclusion File** field is enabled.

The **Exclusion File** contains the exclusion file name and location associated with the model. You can use an exclusion file with several models. However, a model can have only one exclusion file.

Unless you specify a different folder, the Model Advisor saves exclusion files in the current folder. The default name for an exclusion file is `<model_name>_exclusions.xml`.

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

Create Model Advisor Exclusions

- 1 In the model window, right-click a block and select **Model Advisor**. Select the menu option for the type of exclusion that you want to do.

To	Select Model Advisor >
Exclude the block from all checks.	Exclude block only > All Checks
Exclude all blocks of this type from all checks.	Exclude all blocks with type <block_type> > All Checks
Exclude the block from selected checks.	<ul style="list-style-type: none"> • Exclude block only > Select Checks. • In the Check Selector dialog box, select the checks. Click OK.
Exclude all blocks of this type from selected checks.	<ul style="list-style-type: none"> • Exclude all blocks with type <block_type> > Select Checks. • In the Check Selector dialog box, select the checks. Click OK.
Exclude the block from all failed checks. After a Model Advisor analysis, this option is available.	Exclude block only > Only failed checks
Exclude all blocks of this type from all failed checks. After a Model Advisor analysis, this option is available.	Exclude all blocks with type <block_type> > Only failed checks
Exclude the block from a failed check. After a Model Advisor analysis, this option is available.	Exclude block only > <name of failed check>
Exclude all blocks of this type from a failed check. After a Model Advisor analysis, this option is available.	Exclude all blocks with type <block_type> > <name of failed check>

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

- Store exclusions in model file, select **Store exclusions in model file**. Click **OK** or **Apply** to create the exclusion.
 - Save the information to an exclusion file, clear **Store exclusions in model file**. Click **OK** or **Apply**. If this exclusion is the first one, a Save Exclusion File as dialog box opens. In this dialog box, click **Save** to create a exclusion file with the default name `<model_name>_exclusions.xml` in the current folder. Optionally, you can select a different file name or location.
- 3** Optionally, if you want to change the exclusion file name or location:
- a** In the Model Advisor Exclusion Editor dialog box, clear **Store exclusions in model file**.
 - b** In the Model Advisor Exclusion Editor dialog box, select **Change**.
 - c** In the Change Exclusion File dialog box, select **Save as**.
 - d** In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.
 - e** In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information to an exclusion file.

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


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

Review Model Advisor Exclusions

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

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

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

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

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

- HTML report. Before the analysis, in the Model Advisor window, make sure that you select the **Show report after run** check box.
- Model Advisor window. In the left pane of the Model Advisor window, select **By Product > Simulink > < name of check >**. If the **By Product** folder is not displayed, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

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

Manage Exclusions

Save Exclusions in a File

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

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

- 2 If you want to change the exclusion file name or location:
 - a In the Model Advisor Exclusion Editor dialog box, select **Change**.
 - b In the Change Exclusion File dialog box, select **Save as**.
 - c In the Save Exclusion File dialog box, navigate to the location that you want and enter a file name. Click **Save**.
 - d In the Model Advisor Exclusion Editor dialog box, select **OK** or **Apply** to create the exclusion and save the information in an exclusion file.

Load an Exclusion File

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

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

Detach an Exclusion File

To detach an exclusion file associated with your model:

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

Remove an Exclusion

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

Add a Rationale to an Exclusion

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

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

Programmatically Specify an Exclusion File

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

- 1 Use `get_param` to obtain the model object. For example, for `sldemo_mdadv`:

```
mo = get_param('sldemo_mdadv','Object')
```
- 2 Use `MAModelExclusionFile` to specify the name of an exclusion file. For example, to specify exclusion file `my_exclusion.xml` in `S:\work`:

```
mo.MAModelExclusionFile = ['S:\work\', 'my_exclusion.xml']
```
- 3 Open the Model Advisor Exclusion Editor dialog box. The **Exclusion File** field contains the specified exclusion file and path.

Edit-Time Exclusions

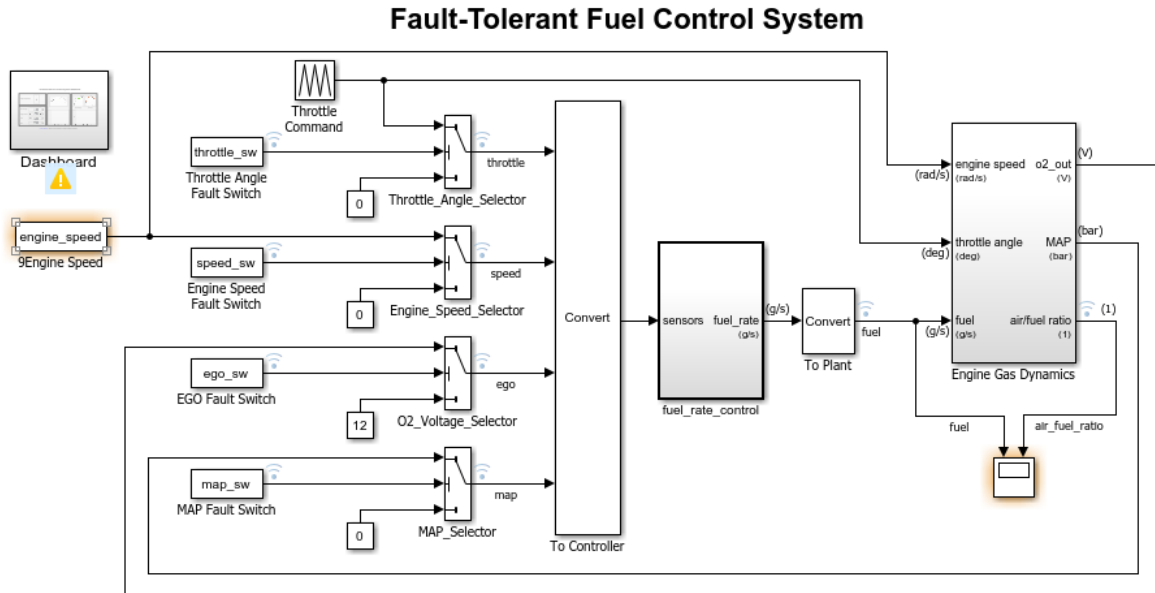
Exclude Checks During Edit-Time

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

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

- 1 From the command prompt, open `sldemo_fuelsys`.
- 2 Introduce a warning that is visible in edit-time checking. Add the number 9 to the beginning of the Engine Speed block name. This number causes a violation in “Check character usage in block names”.

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



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

- 4 To exclude the Engine Speed block from Model Advisor analysis, either:
- Right-click the block, select **Model Advisor > Exclude block only > Select checks**, and select the check.
 - Click the warning icon and click the **Ignore** button. For this block, clicking **Ignore** adds an exclusion to Model Advisor analysis.

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

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

See Also

Related Examples


- “Limit Model Checks By Excluding Gain and Outport Blocks” on page 3-35
- “Exclude Blocks From Custom Checks” on page 7-62

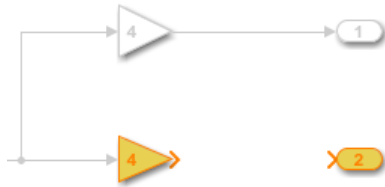
More About

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

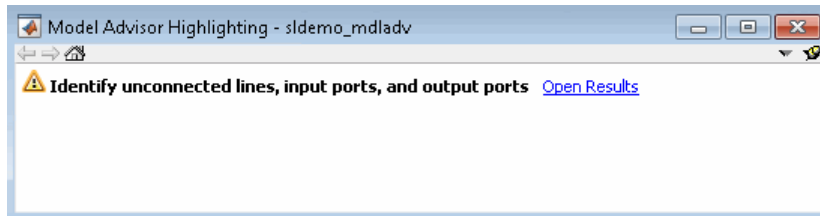
Limit Model Checks By Excluding Gain and Output Blocks

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

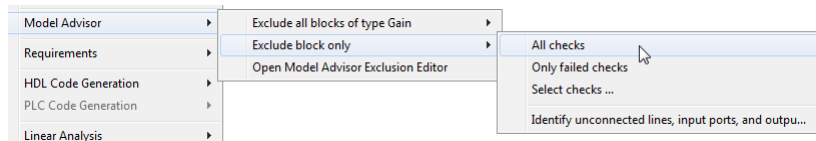
- 1 At the MATLAB command line, type `sldemo_mdadv`.
- 2 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 3 A System Selector — Model Advisor dialog box opens. Click **OK**.
- 4 If the **By Product** folder is not displayed in the Model Advisor window,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



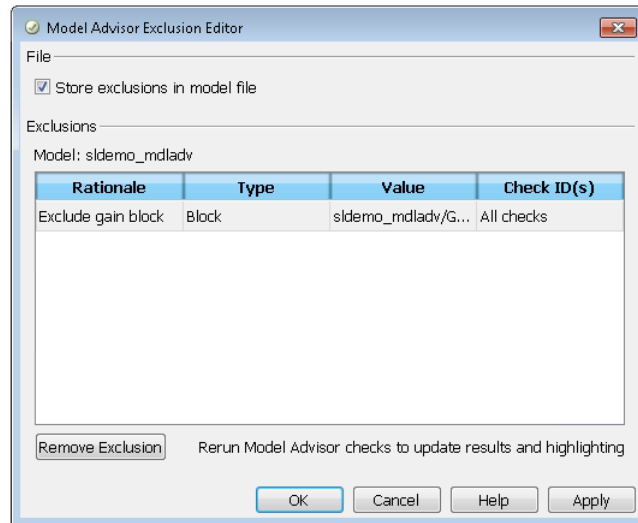
- The Model Advisor Highlighting window opens with a link to the Model Advisor window. In the Model Advisor window, you can find more information about the check results and how to fix the warning condition.



- 9 After reviewing the check results, exclude the Gain2 block from all Model Advisor checks:
 - a In the model window, right-click the Gain2 block and select **Model Advisor > Exclude block only > All checks** .



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



- c Click **OK** to store the exclusion in the model file.
- 10 After reviewing the check results, exclude all Output blocks from the Identify unconnected lines, input ports, and output ports check:
 - a Right-click the Out4 block and select **Model Advisor > Exclude all blocks of type Output > Identify unconnected lines, input ports, and output ports**.
 - b In the Model Advisor Exclusion Editor dialog box, click **OK** to store the exclusion in the model file.
- 11 In the left pane of the Model Advisor window, select **By Product > Simulink** and then:
 - Select the **Show report after run** check box.
 - Click **Run Selected Checks** to run a Model Advisor analysis.
- 12 After the Model Advisor completes the analysis, you can view exclusion information for the Identify unconnected lines, input ports, and output ports check in the:
 - HTML report:

✔ Identify unconnected lines, input ports, and output ports

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

Passed

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

Check Exclusions Rules

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

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

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

Passed

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

Check Exclusions Rules

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

13 Close `sldemo_mdadv`.

See Also

Related Examples

- “Exclude Blocks From Custom Checks” on page 7-62
- “Select and Run Model Advisor Checks” (Simulink)

More About

- “Limit Model Checks” on page 3-26
- “Select and Run Model Advisor Checks” (Simulink)

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

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

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

For information on the DO-178C Software Considerations in Airborne Systems and Equipment Certification and related standards, see Radio Technical Commission for Aeronautics (RTCA) .

The table lists the DO-178C/DO-331 checks. Applicable guidelines are provided for checks used in “High-Integrity System Modeling” (Simulink).

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
Display model version information	Not applicable
Check usage of lookup table blocks	hisl_0033: Usage of Lookup Table blocks
Check for inconsistent vector indexing methods	hisl_0021: Consistent vector indexing method
Check for blocks not recommended for C/C++ production code deployment	hisl_0020: Blocks not recommended for MISRA C:2012 compliance
Check for variant blocks with 'Generate preprocessor conditionals' active	hisl_0023: Verification of model and subsystem variants
Check for root Inports with missing properties	hisl_0024: Inport interface definition
Check usage of Math Operations blocks	<ul style="list-style-type: none"> • hisl_0001: Usage of Abs block • hisl_0002: Usage of Math Function blocks (rem and reciprocal) • hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm) • hisl_0029: Usage of Assignment blocks • hisl_0066: Usage of Gain blocks

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
Check usage of Signal Routing blocks	hisl_0034: Usage of Signal Routing blocks
Check usage of Logic and Bit Operations blocks	<ul style="list-style-type: none"> • hisl_0016: Usage of blocks that compute relational operators • hisl_0017: Usage of blocks that compute relational operators (2) • hisl_0018: Usage of Logical Operator block
Check usage of Ports and Subsystems blocks	<ul style="list-style-type: none"> • hisl_0006: Usage of While Iterator blocks • hisl_0007: Usage of While Iterator subsystems • hisl_0008: Usage of For Iterator Blocks • hisl_0009: Usage of For Iterator Subsystem blocks • hisl_0011: Usage of Switch Case blocks and Action Subsystem 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: Mealy and Moore semantics
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)

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
Check usage of shift operations for Stateflow data	hisf_0064: Shift operations for Stateflow data to improve code compliance
Check assignment operations in Stateflow charts	hisf_0065: Type cast operations in Stateflow to improve code compliance
Check Stateflow charts for unary operators	hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance
Check usage of Stateflow constructs	<ul style="list-style-type: none"> • hisf_0002: User-specified state/transition execution order • hisf_0009: Strong data typing (Simulink and Stateflow boundary) • hisf_0011: Stateflow debugging settings • hisf_0016: Stateflow port names • hisf_0017: Stateflow data object scoping • hisl_0061: Unique identifiers for clarity
Check for MATLAB Function interfaces with inherited properties	himl_0002: Strong data typing at MATLAB function boundaries
Check MATLAB Function metrics	himl_0003: Limitation of MATLAB function complexity
Check MATLAB Code Analyzer messages	himl_0004: MATLAB Code Analyzer recommendations for code generation
Check MATLAB code for global variables	himl_0005: Usage of global variables in MATLAB functions

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related optimization settings	<ul style="list-style-type: none"> • hisl_0018: Usage of Logical Operator block • hisl_0045: Configuration Parameters > Optimization > Implement logic signals as Boolean data (vs. double) • hisl_0046: Configuration Parameters > Optimization > Block reduction • hisl_0048: Configuration Parameters > Optimization > Application lifespan (days) • hisl_0052: Configuration Parameters > Optimization > Data initialization • hisl_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values • hisl_0054: Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions • hisl_0056: Configuration Parameters > Optimization > Optimize using the specified minimum and maximum values
Check safety-related model referencing settings	hisl_0037: Configuration Parameters > Model Referencing
Check safety-related code generation settings	<ul style="list-style-type: none"> • hisl_0038: Configuration Parameters > Code Generation > Comments • hisl_0039: Configuration Parameters > Code Generation > Interface • hisl_0047: Configuration Parameters > Code Generation > Code Style • hisl_0049: Configuration Parameters > Code Generation > Symbols

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
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 signal data	hisl_0005: Usage of Product blocks hisl_0314: Configuration Parameters > Diagnostics > Data Validity > Signals
Check safety-related diagnostic settings for parameters	hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters
Check safety-related diagnostic settings for data used for debugging	hisl_0305: Configuration Parameters > Diagnostics > Debugging
Check safety-related diagnostic settings for data store memory	hisl_0013: Usage of data store blocks
Check safety-related diagnostic settings for type conversions	hisl_0309: Configuration Parameters > Diagnostics > Type Conversion
Check safety-related diagnostic settings for signal connectivity	hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals
Check safety-related diagnostic settings for bus connectivity	hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses
Check safety-related diagnostic settings that apply to function-call connectivity	hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls
Check safety-related diagnostic settings for compatibility	hisl_0301: Configuration Parameters > Diagnostics > Compatibility
Check safety-related diagnostic settings for model initialization	hisl_0304: Configuration Parameters > Diagnostics > Model initialization

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for model referencing	hisl_0310: Configuration Parameters > Diagnostics > Model Referencing
Check safety-related diagnostic settings for saving	hisl_0036: Configuration Parameters > Diagnostics > Saving
Check safety-related diagnostic settings for Merge blocks	hisl_0303: Configuration Parameters > Diagnostics > Merge block
Check safety-related diagnostic settings for Stateflow	hisl_0311: Configuration Parameters > Diagnostics > Stateflow
Check safety-related optimization settings for Loop unrolling threshold	hisl_0051: Configuration Parameters > Optimization > Loop unrolling threshold
Check model object names	hisl_0032: Model object names
Check for model elements that do not link to requirements	hisl_0070: Placement of requirement links in a model
Check for blocks not recommended for MISRA C:2012	hisl_0020: Blocks not recommended for MISRA C:2012 compliance
Check configuration parameters for MISRA C:2012	hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
Check for Discrete-Time Integrator blocks with initial condition uncertainty	Not applicable
Check root model Inport block specifications	Not applicable
Identify unconnected lines, input ports, and output ports	Not applicable
Check usage of tunable parameters in blocks	Not applicable
Check for Strong Data Typing with Simulink I/O	Not applicable
Check for blocks that have constraints on tunable parameters	Not applicable
Identify questionable subsystem settings	Not applicable
Check bus signals treated as vectors	Not applicable

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
Check for potentially delayed function-call subsystem return values	Not applicable
Check usage of Merge blocks	Not applicable
Check Stateflow data objects with local scope	Not applicable
Check usage of exclusive and default states in state machines	Not applicable
Identify disabled library links	Not applicable
Identify parameterized library links	Not applicable
Identify unresolved library links	Not applicable
Check for model reference configuration mismatch	Not applicable
Check for parameter tunability information ignored for referenced models	Not applicable
Identify requirement links with missing documents	Not applicable
Identify requirement links that specify invalid locations within documents	Not applicable
Identify selection-based links having descriptions that do not match their requirements document text	Not applicable
Identify requirement links with path type inconsistent with preferences	Not applicable
Check sample times and tasking mode	Not applicable
Check solver for code generation	Not applicable
Check the hardware implementation	Not applicable
Display bug reports for DO Qualification Kit	Not applicable
Display bug reports for Embedded Coder	Not applicable
Display bug reports for Polyspace Code Prover	Not applicable

DO-178C/DO-331 Check	Applicable High-Integrity System Modeling Guidelines
Display bug reports for Polyspace Bug Finder	Not applicable
Display bug reports for Simulink Code Inspector	Not applicable
Display bug reports for Simulink Report Generator	Not applicable
Display bug reports for Simulink Check	Not applicable
Display bug reports for Simulink Coverage	Not applicable
Display bug reports for Simulink Test	Not applicable
Display bug reports for Simulink Design Verifier	Not applicable

See Also

Related Examples

- Select and Run Model Advisor Checks (Simulink)

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

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

- IEC 61508-3 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements
- IEC 62304 Medical device software - Software life cycle processes
- ISO 26262-6 Road vehicles - Functional safety - Part 6: Product development: Software level
- EN 50128 Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems

To check compliance with these standards, open the Model Advisor and run the checks in these folders.

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

The table lists the IEC 61508, IEC 62304, ISO 26262, and EN 50128 checks. Applicable guidelines are provided for checks used in “High-Integrity System Modeling” (Simulink).

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
Display configuration management data	Not applicable
Display model metrics and complexity report	Not applicable
Check for unconnected objects	Not applicable
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

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
Check for blocks not recommended for C/C++ production code deployment	hisl_0020: Blocks not recommended for MISRA C:2012 compliance
Check for variant blocks with 'Generate preprocessor conditionals' active	hisl_0023: Verification of model and subsystem variants
Check for root Inports with missing properties	hisl_0024: Inport interface definition
Check usage of Math Operations blocks	<ul style="list-style-type: none"> • hisl_0001: Usage of Abs block • hisl_0002: Usage of Math Function blocks (rem and reciprocal) • hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm) • hisl_0029: Usage of Assignment blocks • hisl_0066: Usage of Gain blocks
Check usage of Signal Routing blocks	hisl_0034: Usage of Signal Routing blocks
Check usage of Logic and Bit Operations blocks	<ul style="list-style-type: none"> • hisl_0016: Usage of blocks that compute relational operators • hisl_0017: Usage of blocks that compute relational operators (2) • hisl_0018: Usage of Logical Operator block

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
Check usage of Ports and Subsystems blocks	<ul style="list-style-type: none"> • hisl_0006: Usage of While Iterator blocks • hisl_0007: Usage of While Iterator subsystems • hisl_0008: Usage of For Iterator Blocks • hisl_0009: Usage of For Iterator Subsystem blocks • hisl_0010: Usage of If blocks and If Action Subsystem blocks • hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks
Check for 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: Mealy and Moore semantics
Check Stateflow charts for transition paths that cross parallel state boundaries	hisf_0013: Usage of transition paths (crossing parallel state boundaries)
Check Stateflow charts for ordering of states and transitions	hisf_0002: User-specified state/transition execution order
Check Stateflow debugging options	hisf_0011: Stateflow debugging settings
Check Stateflow charts for uniquely defined data objects	hisl_0061: Unique identifiers for clarity
Check Stateflow charts for strong data typing	hisf_0015: Strong data typing (casting variables and parameters in expressions)
Check usage of shift operations for Stateflow data	hisf_0064: Shift operations for Stateflow data to improve code compliance
Check assignment operations in Stateflow charts	hisf_0065: Type cast operations in Stateflow to improve code compliance

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
Check Stateflow charts for unary operators	hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance
Check usage of Stateflow constructs	<ul style="list-style-type: none"> • hisf_0002: User-specified state/transition execution order • hisf_0009: Strong data typing (Simulink and Stateflow boundary) • hisf_0011: Stateflow debugging settings • hisf_0016: Stateflow port names • hisf_0017: Stateflow data object scoping • hisl_0061: Unique identifiers for clarity
Check for MATLAB Function interfaces with inherited properties	himl_0002: Strong data typing at MATLAB function boundaries
Check MATLAB Function metrics	himl_0003: Limitation of MATLAB function complexity
Check MATLAB Code Analyzer messages	himl_0004: MATLAB Code Analyzer recommendations for code generation
Check MATLAB code for global variables	himl_0005: Usage of global variables in MATLAB functions

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
Check safety-related optimization settings	<ul style="list-style-type: none"> • hisl_0018: Usage of Logical Operator block • hisl_0045: Configuration Parameters > Optimization > Implement logic signals as Boolean data (vs. double) • hisl_0046: Configuration Parameters > Optimization > Block reduction • hisl_0048: Configuration Parameters > Optimization > Application lifespan (days) • hisl_0052: Configuration Parameters > Optimization > Data initialization • hisl_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values • hisl_0054: Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions • hisl_0056: Configuration Parameters > Optimization > Optimize using the specified minimum and maximum values
Check safety-related model referencing settings	hisl_0037: Configuration Parameters > Model Referencing
Check safety-related code generation settings	<ul style="list-style-type: none"> • hisl_0038: Configuration Parameters > Code Generation > Comments • hisl_0039: Configuration Parameters > Code Generation > Interface • hisl_0047: Configuration Parameters > Code Generation > Code Style • hisl_0049: Configuration Parameters > Code Generation > Symbols

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
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 signal data	hisl_0005: Usage of Product blocks hisl_0314: Configuration Parameters > Diagnostics > Data Validity > Signals
Check safety-related diagnostic settings for parameters	hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters
Check safety-related diagnostic settings for data used for debugging	hisl_0305: Configuration Parameters > Diagnostics > Debugging
Check safety-related diagnostic settings for data store memory	hisl_0013: Usage of data store blocks
Check safety-related diagnostic settings for type conversions	hisl_0309: Configuration Parameters > Diagnostics > Type Conversion
Check safety-related diagnostic settings for signal connectivity	hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals
Check safety-related diagnostic settings for bus connectivity	hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses
Check safety-related diagnostic settings that apply to function-call connectivity	hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls
Check safety-related diagnostic settings for compatibility	hisl_0301: Configuration Parameters > Diagnostics > Compatibility
Check safety-related diagnostic settings for model initialization	hisl_0304: Configuration Parameters > Diagnostics > Model initialization

IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for model referencing	hisl_0310: Configuration Parameters > Diagnostics > Model Referencing
Check safety-related diagnostic settings for saving	hisl_0036: Configuration Parameters > Diagnostics > Saving
Check safety-related diagnostic settings for Merge blocks	hisl_0303: Configuration Parameters > Diagnostics > Merge block
Check safety-related diagnostic settings for Stateflow	hisl_0311: Configuration Parameters > Diagnostics > Stateflow
Check safety-related optimization settings for Loop unrolling threshold	hisl_0051: Configuration Parameters > Optimization > Loop unrolling threshold
Check model object names	hisl_0032: Model object names
Check for model elements that do not link to requirements	hisl_0070: Placement of requirement links in a model
Check for blocks not recommended for MISRA C:2012	hisl_0020: Blocks not recommended for MISRA C:2012 compliance
Check configuration parameters for MISRA C:2012	hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
Display bug reports for Embedded Coder	Not applicable
Display bug reports for IEC Certification Kit	Not applicable
Display bug reports for Polyspace Code Prover	Not applicable
Display bug reports for Polyspace Bug Finder	Not applicable
Display bug reports for Simulink Design Verifier	Not applicable
Display bug reports for Simulink Check	Not applicable
Display bug reports for Simulink Coverage	Not applicable
Display bug reports for Simulink Test	Not applicable

See Also

Related Examples

- [Select and Run Model Advisor Checks \(Simulink\)](#)

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

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

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

For MAAB checks, you can control whether the Model Advisor looks under masks or follows links. See “Set MAAB and JMAAB Checks to Check Under Masks or Follow Links” on page 3-69.

The table lists the MAAB checks with the applicable MAAB Control Algorithm Modeling guideline. For JMAAB checks, see “Model Checks for Japan MATLAB Automotive Advisory Board (JMAAB) Guideline Compliance” on page 3-61.

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

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

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check for nondefault block attributes	db_0140: Display of basic block parameters
	Check for matching port and signal names	jm_0010: Port block names in Simulink models
	Check Trigger and Enable block names	jc_0281: Naming of Trigger Port block and Enable Port block
	Check signal line labels	na_0008: Display of labels on signals
	Check for propagated signal labels	na_0009: Entry versus propagation of signal labels
	Check for unconnected ports and signal lines	db_0081: Unconnected signals, block inputs and block outputs
	Check for prohibited blocks in discrete controllers	jm_0001: Prohibited Simulink standard blocks inside controllers
	Check for blocks not recommended for C/C++ production code deployment	
	Check for prohibited sink blocks	hd_0001: Prohibited Simulink sinks
	Check scope of From and Goto blocks	na_0011: Scope of Goto and From blocks
	Check usage of Switch blocks	jc_0141: Use of the Switch block
	Check usage of Relational Operator blocks	jc_0131: Use of Relational Operator block
	Check for indexing in blocks	db_0112: Indexing
	Check usage of buses and Mux blocks	na_0010: Grouping data flows into signals
	Check usage of tunable parameters in blocks	db_0110: Tunable parameters in basic blocks

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check orientation of Subsystem blocks	jc_0111: Direction of Subsystem
	Check fundamental logical and numerical operations	na_0002: Appropriate implementation of fundamental logical and numerical operations
	Check logical expressions in 'If' blocks	na_0003: Simple logical expressions in If Condition block
	Check usage of merge blocks	na_0032: Use of merge blocks
	Check usage of restricted variable names	na_0019: Restricted Variable Names
	Check usage of character vector inside MATLAB Function block	na_0021: Strings
	Check usage of recommended patterns for Switch/Case statements	na_0022: Recommended patterns for Switch/Case statements
Stateflow	Check usage of exclusive and default states in state machines	db_0137: States in state machines
	Check transition orientations in flow charts	db_0132: Transitions in flow charts
	Check entry formatting in State blocks in Stateflow charts	jc_0501: Format of entries in a State block
	Check return value assignments of graphical functions in Stateflow charts	jc_0511: Setting the return value from a graphical function
	Check default transition placement in Stateflow charts	jc_0531: Placement of the default transition
	Check for Strong Data Typing with Simulink I/O	db_0122: Stateflow and Simulink interface signals and parameters

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check Stateflow data objects with local scope	db_0125: Scope of internal signals and local auxiliary variables
	Check usage of return values from a graphical function in Stateflow charts	jc_0521: Use of the return value from graphical functions
	Check for MATLAB expressions in Stateflow charts	db_0127: MATLAB commands in Stateflow
	Check for pointers in Stateflow charts	jm_0011: Pointers in Stateflow
	Check for event broadcasts in Stateflow charts	jm_0012: Event broadcasts
	Check transition actions in Stateflow charts	db_0151: State machine patterns for transition actions
	Check for bitwise operations in Stateflow charts	na_0001: Bitwise Stateflow operators
	Check for unary minus operations on unsigned integers in Stateflow charts	jc_0451: Use of unary minus on unsigned integers in Stateflow
	Check for comparison operations in Stateflow charts	na_0013: Comparison operation in Stateflow
	Check for equality operations between floating-point expressions in Stateflow charts	jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow
	Check for mismatches between names of Stateflow ports and associated signals	db_0123: Stateflow port names
	Check nested states in Stateflow charts	na_0038: Levels in Stateflow charts

By Task > Modeling Standards for MAAB subfolder	Model Advisor Check	Guideline from the MAAB Control Algorithm Modeling Guidelines, Version 3.0
	Check use of Simulink in Stateflow charts	na_0039: Use of Simulink in Stateflow charts
	Check number of Stateflow states per container	na_0040: Number of states per container
MATLAB Functions and Code	Check input and output settings of MATLAB Functions	na_0034: MATLAB Function block input/output settings
	Check MATLAB Function metrics	<ul style="list-style-type: none"> • na_0016: Source lines of MATLAB Functions • na_0018: Number of nested if/else and case statement
	Check MATLAB code for global variables	na_0024: Global Variables

See Also

Related Examples

- Select and Run Model Advisor Checks (Simulink)

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

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

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

For JMAAB checks, you can control whether the Model Advisor looks under masks or follows links. See “Set MAAB and JMAAB Checks to Check Under Masks or Follow Links” on page 3-69.

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

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
Naming Conventions	Check file names	ar_0001: File names
	Check folder names	ar_0002: Directory names
	Check subsystem names	jc_0201: Subsystem block names include incorrect characters

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
	Check port block names	jc_0211: Inport or Outport block names include incorrect characters
	Check character usage in signal labels	jc_0222: Signal line names include incorrect characters
	Check character usage in block names	jc_0231: Block names include incorrect characters
Model Architecture	Check for mixing basic blocks and subsystems	db_0143: Similar block types on the model levels
Model Configuration Options	Check Implement logic signals as Boolean data (vs. double)	jc_0011: Optimization parameters for Boolean data types
Simulink	Check for Simulink diagrams using nonstandard display attributes	na_0004: Simulink model appearance
	Check font formatting	db_0043: Simulink font and font size

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
	Check positioning and configuration of ports	db_0042: Ports in Simulink models
	Check whether block names appear below blocks	db_0142: Position of block names
	Check the display attributes of block names	jc_0061: Display of block names
	Check position of Trigger and Enable blocks	db_0146: Triggered or enabled subsystems
	Check for nondefault block attributes	db_0140: Display of basic block attributes
	Check Trigger and Enable block names	jc_0281: Naming of Trigger block and Enable block
	Check signal line labels	na_0008: Labeling of signal lines
	Check for propagated signal labels	na_0009: Labeling for propagated signal lines

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
	Check for unconnected ports and signal lines	db_0081: Unconnected signals, block input ports, and block output ports
	Check for prohibited blocks in discrete controllers	jm_0001: Prohibited Simulink standard blocks inside controllers
	Check for prohibited sink blocks	hd_0001: Prohibited Simulink sink blocks
	Check usage of Switch blocks	jc_0141: Use of the Switch block
	Check usage of Relational Operator blocks	jc_0131: Use of relational operators with constant input values
	Check for indexing in blocks	db_0112: Indexing
	Check usage of tunable parameters in blocks	db_0110: Tunable parameters in basic blocks

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
	Check orientation of Subsystem blocks	jc_0111: Direction of Subsystem
	Check usage of Discrete-Time Integrator block	jc_0627: Guideline for using the Discrete-time Integrator block
	Check for blocks with a fixed-point data type whose bias is not zero	jc_0643: Fixed-point setting
	Check input and output datatype for Switch blocks	jc_0650: Block input/output data type with switching function
	Check input signal data types in product blocks that perform division	jc_0611: Input signal sign during product block division
Stateflow	Check transition orientations in flow charts	db_0132: Transitions in flow charts
	Check return value assignments of graphical functions in Stateflow charts	jc_0511: Setting the return value from a graphical function

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
	Check default transition placement in Stateflow charts	jc_0531: Placement of default transition
	Check for Strong Data Typing with Simulink I/O	db_0122: Check whether labeled input and output signals are strongly typed
	Check Stateflow data objects with local scope	db_0125: Scope of internal signals and local auxiliary variables
	Check usage of return values from a graphical function in Stateflow charts	jc_0521: Use of the return value from graphical functions
	Check for MATLAB expressions in Stateflow charts	db_0127: MATLAB commands in Stateflow
	Check for pointers in Stateflow charts	jm_0011: Pointers in Stateflow
	Check for event broadcasts in Stateflow charts	jm_0012: Event broadcasts

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
	Check transition actions in Stateflow charts	db_0151: State machine patterns for transition actions
	Check for bitwise operations in Stateflow charts	na_0001: Bitwise Stateflow operators
	Check for unary minus operations on unsigned integers in Stateflow charts	jc_0451: Use of unary minus on unsigned integers in Stateflow
	Check for comparison operations in Stateflow charts	na_0013: Comparison operation in Stateflow
	Check for mismatches between names of Stateflow ports and associated signals	db_0123: Stateflow port names
MATLAB Function	Check input and output settings of MATLAB Functions	na_0034: MATLAB Function block input/output settings.

By Task > Modeling Standards for JMAAB subfolder	Model Advisor Check	Guideline from the JMAAB Control Algorithm Modeling Guidelines, Version 4.01
	Check MATLAB Function metrics	na_0016: Source lines of MATLAB Functions na_0018: Number of nested if/else and case statement
	Check MATLAB code for global variables	na_0024: Global Variables.

See Also

Related Examples

- Select and Run Model Advisor Checks (Simulink)

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

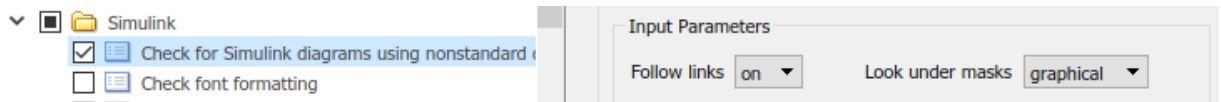
Control Whether Checks Look Under Masks or Follow Links

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

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

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

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



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

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

See Also

`find_system`

More About

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

Model Checks for MISRA C:2012 Compliance

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

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

See Also

Related Examples

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

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

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

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

See Also

Related Examples

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

Model Checks for Requirements Links

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

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

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

When modeling for high-integrity systems, to check that model elements link to requirement documents, run these high-integrity checks:

- For DO-178C/DO-331, see Check for blocks that do not link to requirements
- For IEC 61508, IEC 62304, EN 50128, and ISO 26262, see Check for model objects that do not link to requirements

See Also

Related Examples

- “Validate Requirements Links in a Model” (Simulink Requirements)
- “Select and Run Model Advisor Checks” (Simulink)
- “High-Integrity System Modeling” (Simulink)

Generate Model Advisor Reports in Adobe PDF and Microsoft Word Formats

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

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

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

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

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

Modify Default Template

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

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

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

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

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

My Custom Header

Model Advisor Report – Model name

Simulink version: Simulink version

Model version: Model version

System: System name

Current run: Timestamp

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

Run Summary

Pass	Fail	Warning	Not Run	Total
 Passed check	 Failed check	 Warning check	 Not run check	Total number

Results of all checks

- 4 In the Model Advisor window Report pane, click **Generate Report**.
- 5 In the Generate Model Advisor Report dialog box:
 - Enter the path to the folder where you want to generate the report and provide a file name.
 - Set **File format** to PDF.
 - Select **View report after generation**.
 - Set **Report template** to C:\work\ma_format\mytemplate.dotx.
- 6 Click **OK**. The Model Advisor generates the report in PDF format with a custom header. Because the template mytemplate.dotx contains holes that Model Advisor uses to populate the generated report, the report contains information about the Model Advisor analysis. For example, the report contains the model name, model version, and number of checks that pass.

My Custom Header

Model Advisor Report – `sldemo_mdldv`

Simulink version: 8.5

Model version: 1.78

System: `sldemo_mdldv`

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

Treat as Referenced Model: off

Run Summary

Pass	Fail	Warning	Not Run	Total
 1	 0	 2	 30	33

See Also

Related Examples

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

Check Systems Programmatically

Checking Systems Programmatically

The Simulink Check product includes a programmable interface for scripting and for command-line interaction with the Model Advisor. Using this interface, you can:

- Create scripts and functions for distribution that check one or more systems using the Model Advisor.
- Run the Model Advisor on multiple systems in parallel on multicore machines (requires a Parallel Computing Toolbox™ license).
- Check one or more systems using the Model Advisor from the command line.
- Archive results for reviewing at a later time.

To define the workflow for running multiple checks on systems:

- 1 Specify a list of checks to run. Do one of the following:
 - Create a Model Advisor configuration file that includes only the checks that you want to run.
 - Create a list of check IDs.
- 2 Specify a list of systems to check.
- 3 Run the Model Advisor checks on the list of systems using the `ModelAdvisor.run` function.
- 4 Archive and review the results of the run.

See Also

`ModelAdvisor.run`

Related Examples

- “Archive and View Results” on page 4-10
- “Find Check IDs” on page 4-3

More About

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

Find Check IDs

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

To Find	Do This
Check Title, ID, or location of the MATLAB source code	<ol style="list-style-type: none"> 1 On the model window toolbar, select Settings > Preferences. 2 In the Model Advisor Preferences dialog box, select Show Source Tab. 3 In the right pane of the Model Advisor window, click the Source tab. The Model Advisor window displays the check Title, TitleId, and location of the MATLAB source code for the check.
Check ID	<ol style="list-style-type: none"> 1 In the left pane of the Model Advisor, select the check. 2 Right-click the check name and select Send Check ID to Workspace. The ID is displayed in the Command Window and sent to the base workspace.
Check IDs for selected checks in a folder	<ol style="list-style-type: none"> 1 In the left pane of the Model Advisor, select the checks for which you want IDs. Clear the other checks in the folder. 2 Right-click the folder and select Send Check ID to Workspace. An array of the selected check IDs are sent to the base workspace.

If you know a check ID from a previous release, you can find the current check ID using the `ModelAdvisor.lookupCheckID` function. For example, the check ID for **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check safety-related optimization settings** prior to Release 2010b was `D0178B:OptionSet`. Using the `ModelAdvisor.lookupCheckID` function returns:

```
>> NewID = ModelAdvisor.lookupCheckID('D0178B:OptionSet')

NewID =

mathworks.do178.OptionSet
```

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

See Also

`ModelAdvisor.lookupCheckID`

Create a Function for Checking Multiple Systems

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

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

```
fileName = 'slvndemo_mdadv_config.mat';
```

- 7 Call the `ModelAdvisor.run` function:

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

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

```
fail=0;
warn=0;

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

The function should now look like this:

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

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

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

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

end
```

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

```
[failures, warnings] = run_configuration(...  
    'sldemo_auto_climatecontrol/Heater Control');
```
- 11 Review the results. Click the Summary Report link to open the Model Advisor Command-Line Summary report.

See Also

`ModelAdvisor.run`

Related Examples

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

Check Multiple Systems in Parallel

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

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

Each parallel process runs checks on one model at a time. In parallel mode, load the model data from the model workspace or data dictionary. The Model Advisor in parallel mode does not support model data in the base workspace.

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

See Also

`ModelAdvisor.run`

Related Examples

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

Create a Function for Checking Multiple Systems in Parallel

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

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

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

The function should now look like this:

```
function [fail, warn] = run_fast_configuration(SysList)  
%RUN_FAST_CONFIGURATION Check systems in parallel with Model Advisor  
% Return number of warnings and failures.  
fileName = 'slvndemo_mdladv_config.mat';  
fail=0;  
warn=0;  
  
SysResultObjArray = ModelAdvisor.run(SysList,'Configuration',fileName,...  
    'ParallelMode','On');
```

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

- 5 Save the function.
- 6 Test the function. In the MATLAB Command Window, create a list of systems:

```
SysList={'sldemo_auto_climatecontrol/Heater Control',...  
    'sldemo_auto_climatecontrol/AC Control','rtwdemo_iec61508'};
```

- 7 Run `run_fast_configuration` on the list of systems:

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

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

See Also

ModelAdvisor.run

Related Examples

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

Archive and View Results

Archive Results

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

View Results in Command Window

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

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

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

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

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

The results displayed in the Command Window are:

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

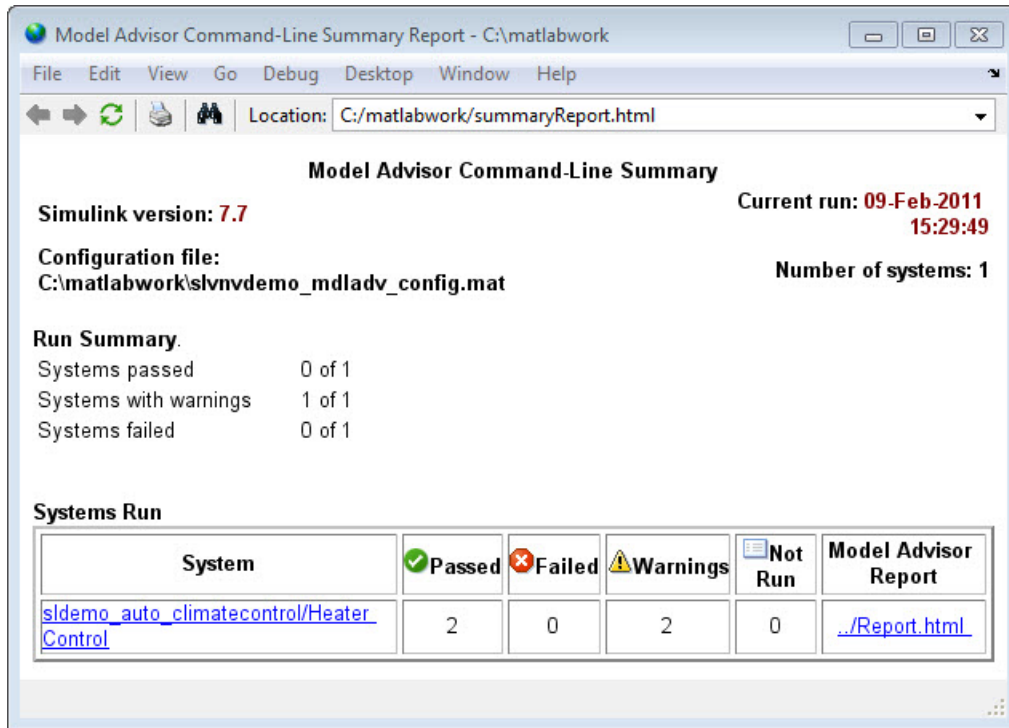
```
Summary:   Pass   Warning   Fail   Not Run
          2       2         0     0
=====

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

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

View Results in Model Advisor Command-Line Summary Report

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



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

View Results in Model Advisor GUI

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

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

View Model Advisor Report

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

To view the Model Advisor report for a system:

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

To view individual check results:

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

See Also

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

Related Examples

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

More About

- “Run Model Checks” (Simulink)
- “Save and Load Process for Objects” (MATLAB)

Archive and View Model Advisor Run Results

This example guides you through archiving the results of running checks so that you can review them at a later time. To simulate archiving and reviewing, the steps in the tutorial detail how to save the results, clear out the MATLAB workspace (simulates shutting down MATLAB), and then load and review the results.

- 1 Call the `ModelAdvisor.run` function:

```
SysResultObjArray = ModelAdvisor.run({'sldemo_auto_climatecontrol/Heater Control'},...  
    'Configuration','slvndemo_mdadv_config.mat');
```

- 2 Save the `SysResultObj` for use at a later time:

```
save my_model_advisor_run SysResultObjArray
```

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

```
clear
```

- 4 Load the results of the Model Advisor run:

```
load my_model_advisor_run SysResultObjArray
```

- 5 View the results in the Model Advisor:

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

See Also

`ModelAdvisor.run`

Related Examples

- “Archive and View Results” on page 4-10

Model Metrics

Collect and Explore Metric Data by Using the Metrics Dashboard

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

- From a model editor window, select **Analysis > Metrics Dashboard**.
- At the command line, enter `metricsdashboard(system)`. The *system* can be either a model name or a block path to a subsystem. The system cannot be a Configurable Subsystem block.

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

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

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

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

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

- 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 all variants.

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 (<code>mathworks.metrics.SimulinkBlockCount</code>)	Number of blocks by component
Models	Model file count (<code>mathworks.metrics.ModelFileCount</code>)	Number of model files by component
Files	File count (<code>mathworks.metrics.FileCount</code>)	Number of model and library files by component
MATLAB LOC	Effective lines of MATLAB code (<code>mathworks.metrics.MatlabLOCCount</code>)	Effective lines of code, in MATLAB Function block and MATLAB functions in Stateflow, by component
Stateflow LOC	Effective lines of code for Stateflow blocks (<code>mathworks.metrics.StateflowLOCCount</code>)	Effective lines of code for Stateflow blocks by component

Widget	Metric	Drill-In Data
System Interface	<ul style="list-style-type: none"> • Input and Output count (<code>mathworks.metrics.ExplicitIOCount</code>) • Parameter count (<code>mathworks.metrics.ParameterCount</code>) 	<ul style="list-style-type: none"> • 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 (<code>mathworks.metrics.ModelAdvisorCheckCompliance.hisl_dol78</code>)	<p>For each component:</p> <ul style="list-style-type: none"> • Percentage of checks passed • Status of each check <p>Integration with the Model Advisor for more detailed results. Click Table or Grid to specify the format in which you want to view results.</p>
MAAB Compliance	Model Advisor standards check compliance - MAAB (<code>mathworks.metrics.ModelAdvisorCheckCompliance.maab</code>)	<p>For each component:</p> <ul style="list-style-type: none"> • Percentage of checks passed • Status of each check <p>Integration with the Model Advisor for more detailed results. Click Table or Grid to specify the format in which you want to view results.</p>

Widget	Metric	Drill-In Data
High Integrity Check Issues	Model Advisor standards issues - High Integrity (<code>mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178</code>)	<ul style="list-style-type: none"> Number of compliance check issues by component (see the following Note below). Components without issues or aggregated issues are not listed.
MAAB Check Issues	Model Advisor standards issues - MAAB (<code>mathworks.metrics.ModelAdvisorCheckIssues.maab</code>)	<ul style="list-style-type: none"> 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 (<code>mathworks.metrics.MatlabCodeAnalyzerWarnings</code>)	Number of Code Analyzer warnings by component.
Diagnostic Warnings	Simulink diagnostic warning count (<code>mathworks.metrics.DiagnosticWarningsCount</code>)	<ul style="list-style-type: none"> Numer 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.

Architecture

These widgets provide a view of your system architecture:

- The **Library Reuse** widget is a percentage scale that shows the percentage of subsystems that are candidates for reuse. Orange indicates potential reuse. Green 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
Library Reuse	<p>Directly:</p> <ul style="list-style-type: none"> • Clone detection (<code>mathworks.metrics.CloneDetection</code>) • Library link (<code>mathworks.metrics.LibraryCount</code>) <p>Indirectly (for percentage values):</p> <ul style="list-style-type: none"> • MATLAB Function count (<code>mathworks.metrics.MatlabFunctionCount</code>) • Chart count (<code>mathworks.metrics.StateflowChartCount</code>) • Subsystem count (<code>mathworks.metrics.SubsystemCount</code>) 	<p>Number of clones per component, broken down into clone patterns</p> <p>Number of components involved in a library, excluding clones</p> <p>Integrate with the Identify Modeling Clones tool by clicking the Open Conversion Tool button.</p>
Model Complexity	Cyclomatic complexity (<code>mathworks.metrics.CyclomaticComplexity</code>)	Model complexity by component
Blocks	Simulink block count (<code>mathworks.metrics.SimulinkBlockCount</code>)	Number of blocks by component
Stateflow LOC	Effective lines of code for Stateflow blocks (<code>mathworks.metrics.StateflowLOCCount</code>)	Effective lines of code for Stateflow blocks by component
MATLAB LOC	Effective lines of MATLAB code (<code>mathworks.metrics.MatlabLOCCount</code>)	Effective lines of code, in MATLAB Function block and MATLAB functions in Stateflow, by component

Dashboard Limitations

When using the Metrics Dashboard, note these considerations:

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

See Also

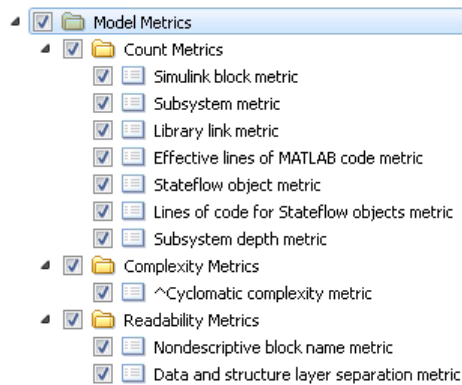
More About

- “Collect Model Metrics Programmatically” on page 5-18
- “Model Metrics”

Collect Model Metrics Using the Model Advisor

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

- 1 Open the `sldemo_fuelsys` model.
- 2 From the Simulink Editor, select **Analysis > Model Advisor > Model Advisor**. A System Selector — Model Advisor dialog box opens. Click **OK**.
- 3 In the left pane of the Model Advisor, navigate to **By Task > Model Metrics**. Select the model metrics to run on your model.



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

Display number of blocks in the model or subsystem.

Passed

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

▼ More (17 rows)

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

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

See Also

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-22
- “Collect Model Metrics Programmatically” on page 5-18
- “Create a Custom Model Metric” on page 5-10

Create a Custom Model Metric

To create your own custom model metric:

- 1 Use the `slmetric.metric.createNewMetricClass` function to create a new metric class derived from the base class `slmetric.metric.Metric`.
- 2 Set the following properties of the class:
 - **ID:** Unique metric identifier that retrieves the new metric data.
 - **Name:** Name of the metric algorithm.
 - **ComponentScope:** Model components for which the metric is calculated.
 - **CompileContext:** Compile mode for metric calculation. If your model requires model metric requires model compilation, specify `PostCompile`. Collecting metric data for compiled models slows performance.
 - **ResultCheckSumCoverage:** Specify whether you want the metric data regenerated if source file and `Version` have not changed.
 - **AggregationMode:** How the metric algorithm aggregates metric data.
 - **AggregateComponentDetails:** Returns all detailed results or aggregates detailed results of the component.

Optionally, set these additional properties:

- **Description:** Description of the metric.
 - **Version:** Metric version.
- 3 Write the metric algorithm into the `slmetric.metric.Metric` method, `algorithm`. The algorithm calculates the metric data specified by the `Advisor.component.Component` class. The `Advisor.component.Types` class specifies the types of model objects for which you can calculate metric data.

Create Model Metric for Nonvirtual Block Count

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

Create Metric Class

Using the `createNewMetricClass` function, create a new metric class named `nonvirtualblockcount`. The function creates a file, `nonvirtualblockcount.m`, in

the current working folder. The file contains a constructor and empty metric algorithm method. For this example, make sure you are in a writable folder.

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

Create Nonvirtual Block Count Metric

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

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

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

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

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

```
res.MetricID = this.ID;

% Practice

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

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

isNonVirtual = true(size(blocks));

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

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

```

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

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

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

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

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

blocks = blocks(isNonVirtual);

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

```

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

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

Collect Metric Data

To collect metric data on models, use instances of `slmetric.Engine`. Using the `getMetrics` method, specify the metrics you want to collect. For this example, specify the nonvirtual block count metric for the `sldemo_mdhref_bus` model.

Load the `sldemo_mdhref_bus` model.

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

Create a metric engine object and set the analysis root.

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

Collect metric data for the nonvirtual block count metric.

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

Display and Export Results

To access the metrics for your model, use instances of `slmetric.metric.Result`. In this example, display the nonvirtual block count metrics for the `sldemo_mdhref_busmodel`. For each result, display the `MetricID`, `ComponentPath`, and `Value`.

```
for n=1:length(rc)  
    if rc(n).Status == 0  
        results = rc(n).Results;  
  
        for m=1:length(results)  
            disp(['MetricID: ', results(m).MetricID]);  
            disp([' ComponentPath: ', results(m).ComponentPath]);  
            disp([' Value: ', num2str(results(m).Value)]);  
            disp(' ');  
        end  
    else  
        disp(['No results for:', rc(n).MetricID]);  
    end  
    disp(' ');  
end
```

Here are the results.

MetricID: nonvirtualblockcount
ComponentPath: sldemo_mdref_bus
Value: 15

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

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

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

MetricID: nonvirtualblockcount
ComponentPath: sldemo_mdref_bus/More Info2
Value: 0

MetricID: nonvirtualblockcount
ComponentPath: sldemo_mdref_counter_bus
Value: 2

MetricID: nonvirtualblockcount
ComponentPath: sldemo_mdref_counter_bus/COUNTER
Value: 6

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

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

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

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

```
MetricID: nonvirtualblockcount  
ComponentPath: sldemo_mdhref_counter_bus/COUNTER/Counter/SaturationCheck  
Value: 5
```

```
MetricID: nonvirtualblockcount  
ComponentPath: sldemo_mdhref_counter_bus/COUNTER/LimitsProcess  
Value: 1
```

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

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

To export the metric results to an XML file, use the `exportMetrics` method. For each metric result, the XML file includes the `ComponentID`, `ComponentPath`, `MetricID`, `Value`, `AggregatedValue`, and `Measure`.

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

For this example, unregister the nonvirtual block count metric.

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

Close the model.

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

Limitations

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

- Linked Stateflow charts
- MATLAB Function blocks

Custom metric algorithms do not follow library links.

See Also

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

More About

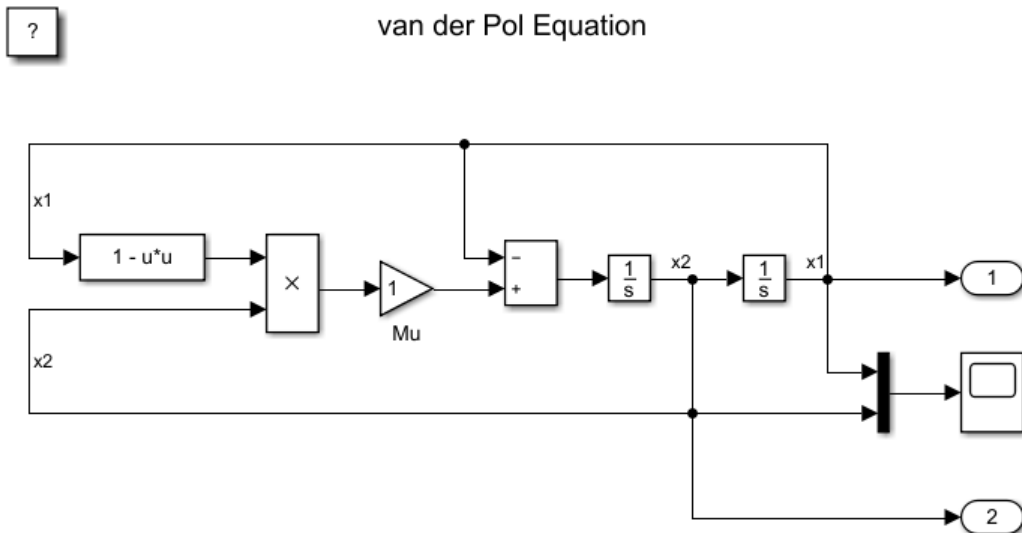
- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-22
- “Collect Model Metrics Programmatically” on page 5-18

Collect Model Metrics Programmatically

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

Example Model

Open model vdp.



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

Collect Metrics

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

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

Access Results

Using the `getMetrics` method, specify the metrics you want to collect. For this example, specify the block count and subsystem count metrics for the vdp model. `getMetrics` returns an array of `slmetric.metric.ResultCollection` objects.

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

Display and Store Results

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

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

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

Here are the results.

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

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

Export Results to a Spreadsheet

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

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

Export Results to an XML File

To export the metric results to an XML file, use the `exportMetrics` method. For each metric result, the XML file includes the `ComponentID`, `ComponentPath`, `MetricID`, `Value`, `AggregatedValue`, and `Measure`.

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

Close the model `vdp`.

```
bdclose(model);
```

Limitations

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

See Also

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

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-22
- “Collect Model Metrics Using the Model Advisor” on page 5-8
- “Create a Custom Model Metric” on page 5-10

Model Metric Data Aggregation

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

How Model Metric Aggregation Works

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

- `AggregationMode`, which has these options:
 - `Sum`: Returns the sum of the `Value` property and the `Value` properties of its children components across the component hierarchy. Returns the sum of the `Measures` property and the `Measures` properties of its children components across the component hierarchy.
 - `Max`: Returns the maximum of the `Value` property and the `Value` properties of its children components across the component hierarchy. Returns the maximum of the `Measures` property and the `Measures` properties of its children components across the component hierarchy.
 - `None`: No aggregation of metric values.
- `AggregateComponentDetails` is a Boolean value, which has these options:
 - `true`: For metrics that return fine-granular results (that is, more than one result per component), the software aggregates these results to the component level by taking the sum of the values and measures properties. Returns a result that spans the complete component.
 - `false`: Returns the component results. The software does not aggregate the fine-granular results.

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

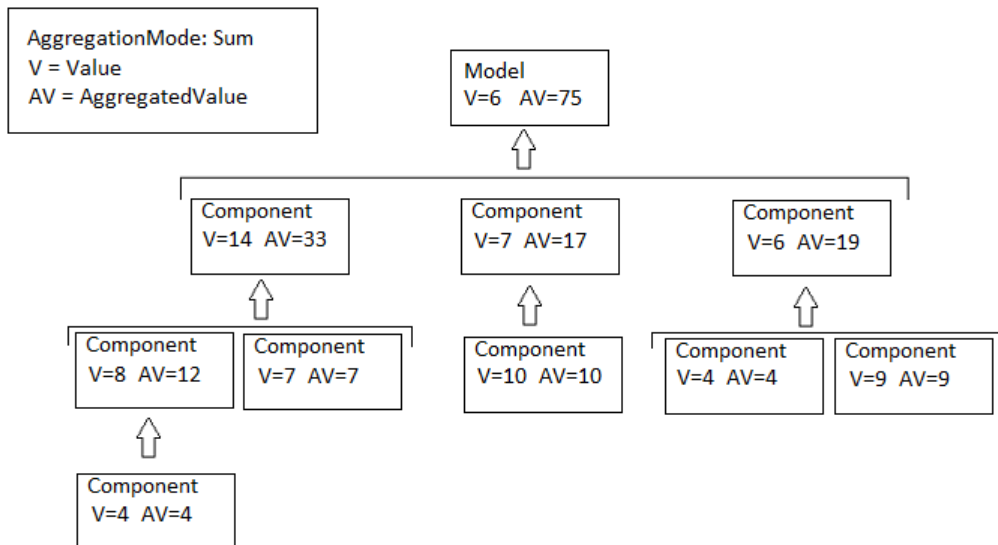
- “Cyclomatic complexity metric”, which creates a result for each state in a Chart.

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

You can find descriptions of MathWorks model metrics and their aggregation property settings in “Model Metrics”. For custom metrics, as part of the `algorithm` method, you can define how the metric aggregates data. For more information, see “Create a Custom Model Metric” on page 5-10.

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

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



Access Aggregated Metric Data

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

- 1 Load the `sldemo_applyVarStruct` model.

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

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

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

- 3 Collect data for the Input output model metric.

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

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

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

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

- 5 Display the results.

```
metricData = {'MetricID', 'ComponentPath', 'Value', ...  
             'AggregatedValue', 'Measures', 'AggregatedMeasures'};  
cnt = 1;  
for n=1:length(res_col)  
    if res_col(n).Status == 0  
        results = res_col(n).Results;  
  
        for m=1:length(results)  
            disp(['MetricID: ', results(m).MetricID]);  
            disp([' ComponentPath: ', results(m).ComponentPath]);  
            disp([' Value: ', num2str(results(m).Value)]);  
            disp([' Aggregated Value: ', num2str(results(m).AggregatedValue)]);  
            disp([' Measures: ', num2str(results(m).Measures)]);  
            disp([' Aggregated Measures: ', ...  
                num2str(results(m).AggregatedMeasures)]);  
        end  
    end  
end
```



```

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

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

See Also

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

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-22
- “Collect Model Metrics Using the Model Advisor” on page 5-8
- “Create a Custom Model Metric” on page 5-10

Overview of Customizing the Model Advisor

Model Advisor Customization

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

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

To	See
Create Model Advisor checks.	“Create Model Advisor Checks”
Format check results.	“Format Check Results” on page 7-65
Create custom Model Advisor configurations.	“Create Custom Configurations” on page 8-2
Specify the order in which you make changes to your model.	“Organize and Deploy Model Advisor Checks”
Deploy custom configurations to your users.	“Organize and Deploy Model Advisor Checks”
Verify that models comply with modeling guidelines.	“Check Model Compliance”

Requirements for Customizing the Model Advisor

Before customizing the Model Advisor:

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

Create Model Advisor Checks

Create Model Advisor Checks Workflow

- 1 On your MATLAB path, create a recustomization file named `sl_customization.m`. In this file, create a `sl_customization()` function to register the custom checks that you create with the Model Advisor. For detailed information, see “Register Checks” on page 7-35.
- 2 Define custom checks and where they appear in the Model Advisor. For detailed information, see “Define Custom Checks” on page 7-40.
- 3 Specify what actions you want the Model Advisor to take for the custom checks by creating a check callback function for each custom check. For detailed information, see “Create Callback Functions and Results” on page 7-48.
- 4 Optionally, specify what automatic fix operations the Model Advisor performs by creating an action callback function. For detailed information, see “Action Callback Function” on page 7-55.

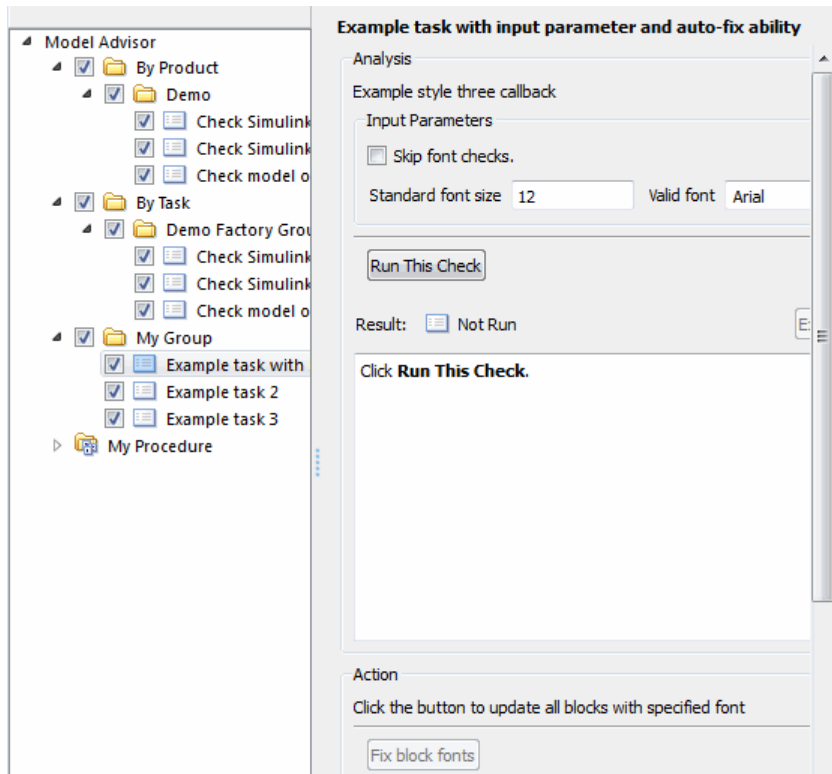
Customization File Overview

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

Function	Description	When Required
<code>sl_customization()</code>	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at startup. See “Register Checks” on page 7-35.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Define Custom Checks” on page 7-40.	Required for custom checks and to add custom checks to the By Product folder. If the By Product folder is not displayed in the Model Advisor window, select Show By Product Folder from the Settings > Preferences dialog box.
Check callback functions	Defines the actions of the custom checks. See “Create Callback Functions and Results” on page 7-48.	Required for custom checks. You must write one callback function for each custom check.
One or more calls to check input parameters	Specifies input parameters to custom checks. See “Define Check Input Parameters” on page 7-43.	Optional
One or more calls to check list views	Specifies calls to the Model Advisor Result Explorer for custom checks. See “Define Model Advisor Result Explorer Views” on page 7-45.	Optional

Function	Description	When Required
One or more calls to check actions	Specifies actions the software performs for custom checks. See “Define Check Actions” on page 7-46 and “Action Callback Function” on page 7-55.	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: <ul style="list-style-type: none"> • Blocks • Lines • Annotations When getting the object, you can: <ul style="list-style-type: none"> • Specify a search depth • Search under masks and libraries
get_param / set_param	Getting and setting system and block parameter values.
Property Inspector	Getting object properties. First you must get a handle to the object.
evalin	Working in the base workspace.
Simulink identifier (SID)	Identifying Simulink blocks, model annotations or Stateflow objects. The SID is a unique number within the model, assigned by Simulink. For details, see “Locate Diagram Components Using Simulink Identifiers” (Simulink).
Stateflow API (Stateflow)	Programmatic access to Stateflow objects.

Create and Add Custom Checks - Basic Examples

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

Add Custom Check to By Product Folder

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

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

```
function sl_customization(cm)

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

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

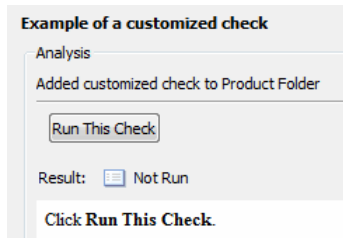
% --- creates SimpleCallback function
function result = SimpleCallback(system);
result={};
```

- 2 Close the Model Advisor and your model if either are open.
- 3 In the Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

- 4 From the MATLAB window, select **New > Simulink Model** to open a new Simulink model window.
- 5 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 6 A **System Selector – Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.
- 7 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 8 In the left pane, expand the **By Product** folder to display the subfolders. The customized check **Example of a customized check** appears in the **By Product > Demo** subfolder.

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



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

Create Customized Pass/Fail Check

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

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

```

function sl_customization(cm)

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

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

mdladvRoot.publish(rec, 'Demo');

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

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

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

```

- 2 Close the Model Advisor and your model if either are open.
- 3 In the Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

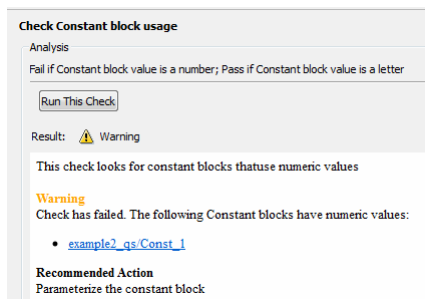
- 4 From the MATLAB window, select **New > Simulink Model** to open a new Simulink model window.
- 5 In the Simulink model window, create two Constant blocks named Const_one and Const_1:

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



- 6 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 7 A **System Selector – Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.
- 8 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 9 In the left pane, select **By Product > Demo > Check Constant block usage**.
- 10 Select **Run This Check**. The Model Advisor check fails for the Const_1 block and displays a **Recommended Action** to parametrize the constant block.

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



- **Check Constant block usage**

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

- **Recommended Action**

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

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

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

Create Customized Pass/Fail Check with Fix Action

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

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

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

```
function sl_customization(cm)  
  
% --- register custom checks  
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);  
  
% --- defineModelAdvisorChecks function  
function defineModelAdvisorChecks  
mdladvRoot = ModelAdvisor.Root;
```

```

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

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

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

mdladvRoot.publish(rec, 'Demo');

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

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

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

```

```
% --- creates SimpleActionCallback function that fixes failed check
function result = simpleActionCallback(taskobj)
mdladvObj = taskobj.MAObj;
result = {};

system = getfullname(mdladvObj.System);

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

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

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

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

3 At the command prompt, enter:

```
Advisor.Manager.refresh_customizations
```

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

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

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

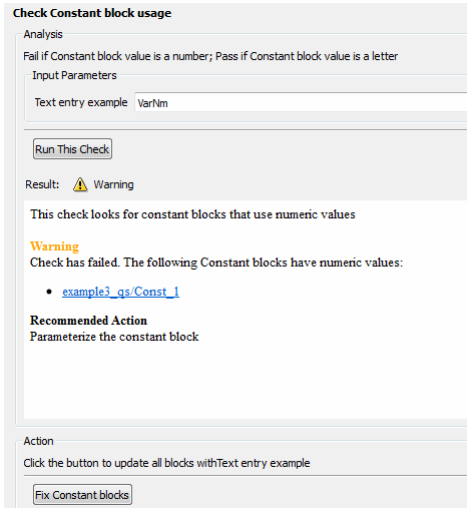
6 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.

7 A **System Selector – Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens. It might take a few minutes.

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

- 9 In the left pane, select **By Product > Demo > Check Constant block usage**.
- 10 Select **Run This Check**. The Model Advisor check fails for the Const_1 block.

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



- **Check Constant block usage**

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

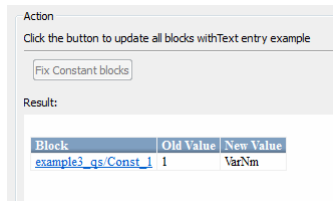
- **Action**

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

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

- 11 In the Model Advisor Dialog box, enter a nonnumeric value in the **Text entry example** parameter field in the **Analysis** section of the Model Advisor dialog box. In this example, the value is VarNm.
- 12 Click **Fix Constant blocks**. The Const_1 **Constant block value** changes from 1 to the nonnumeric value that you entered in step 10. The **Result** section of the dialog box lists the Old Value and New Value of the Const_1 block.

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



- **Action**

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

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

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

See Also

ModelAdvisor.Action | ModelAdvisor.FormatTemplate

More About

- “Register Checks” on page 7-35
- “Define Check Input Parameters” on page 7-43

Create Check for Model Configuration Parameters

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

Decide which configuration parameter settings to use for your model.

Guidelines	See
MathWorks Automotive Advisory Board (MAAB) Control Algorithm Modeling Guidelines	"Model Configuration Options" (Simulink)
High-Integrity System Modeling Guideline	"Configuration Parameter Considerations" (Simulink)
Code Generation Modeling Guidelines	"Code Generation" (Simulink)

- 1 Create an XML data file containing the configuration parameter settings you want to check. You can use `Advisor.authoring.generateConfigurationParameterDataFile` or manually create the file yourself.
- 2 Register the model configuration parameter check using an `sl_customization.m` file.
- 3 Run the check on your models.

Create Data File for Diagnostics Pane Configuration Parameter Check

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

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

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

At the command prompt, type `vdp`.

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

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

Use `Advisor.authoring.generateConfigurationParameterDataFile` to create a data file specifying configuration parameter constraints in the **Diagnostics** pane. Additionally, to create a check with a fix action, set `FixValue` to true. At the command prompt, type:

```
model='vdp';  
dataFileName = 'ex_DataFile.xml';  
Advisor.authoring.generateConfigurationParameterDataFile(dataFileName,...  
model, 'Pane', 'Diagnostics', 'FixValues', true);
```

In the Command Window, select `ex_DataFile.xml`. The data file opens in the MATLAB editor.

- The **Minimize algebraic loop** (command-line: `ArtificialAlgebraicLoopMsg`) configuration parameter tagging specifies a value of error with a `fixvalue` of error. When you run the configuration parameter check using `ex_DataFile.xml`, the check fails if the **Minimize algebraic loop** setting is not error. The check fix action modifies the setting to error.
- The **Block Priority Violation** (command-line: `BlockPriorityViolationMsg`) configuration parameter tagging specifies a value of error with a `fixvalue` of error. When you run the configuration parameter check using `ex_DataFile.xml`, the check fails if the **Block Priority Violation** setting is not error. The check fix action modifies the setting to error.

In `ex_DataFile.xml`, edit the **Algebraic loop** (command-line: `AlgebraicLoopMsg`) parameter tagging so that the check warns if the value is none. Because you are specifying a configuration parameter that you do not want, you need a `NegativeModelParameterConstraint`. Additionally, to create a subcheck that does not have a fix action, remove the line with `<fixvalue>` tagging. The tagging for the configuration parameter looks as follows:

```
<!-- Algebraic loop: (AlgebraicLoopMsg)-->  
<NegativeModelParameterConstraint>  
  <parameter>AlgebraicLoopMsg</parameter>  
  <value>none</value>  
</NegativeModelParameterConstraint>
```

In `ex_DataFile.xml`, delete the lines with tagging for configuration parameters that you do not want to check. The data file `ex_DataFile.xml` has tagging only for **Algebraic loop**, **Minimize algebraic loop** and **Block Priority Violation**. `ex_DataFile.xml` looks similar to:

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.w3schools.com
MySchema.xsd">
  <checkdata>
<!-- Algebraic loop: (AlgebraicLoopMsg)-->
  <NegativeModelParameterConstraint>
    <parameter>AlgebraicLoopMsg</parameter>
    <value>none</value>
  </NegativeModelParameterConstraint>
<!--Minimize algebraic loop: (ArtificialAlgebraicLoopMsg)-->
  <PositiveModelParameterConstraint>
    <parameter>ArtificialAlgebraicLoopMsg</parameter>
    <value>error</value>
    <fixvalue>error</fixvalue>
  </PositiveModelParameterConstraint>
<!--Block priority violation: (BlockPriorityViolationMsg)-->
  <PositiveModelParameterConstraint>
    <parameter>BlockPriorityViolationMsg</parameter>
    <value>error</value>
    <fixvalue>error</fixvalue>
  </PositiveModelParameterConstraint>
</checkdata>
</customcheck>
```

Verify the data syntax with `Advisor.authoring.DataFile.validate`. At the command prompt, type:

```
dataFile = 'myDataFile.xml';
msg = Advisor.authoring.DataFile.validate(dataFile);

if isempty(msg)
    disp('Data file passed the XSD schema validation.');
```

```
else
```

```
        disp(msg);  
    end
```

Create Check for Diagnostics Pane Model Configuration Parameters

This example shows how to create a check for **Diagnostics** pane model configuration parameters using a data file and an `sl_customization` file. First, you register the check using an `sl_customization` file. Using `ex_DataFile.xml`, the check warns when:

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

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

The check uses the `ex_DataFile.xml` data file created in “Create Data File for Diagnostics Pane Configuration Parameter Check” on page 7-15.

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

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

```
function sl_customization(cm)  
  
% register custom checks.  
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);  
  
% register items to factory group.  
cm.addModelAdvisorTaskFcn(@defineModelAdvisorGroups);  
  
%% defineModelAdvisorChecks  
function defineModelAdvisorChecks  
  
    rec = ModelAdvisor.Check('com.mathworks.Check1');  
    rec.Title = 'Example: Check model configuration parameters';  
    rec.setCallbackFcn(@(system) (Advisor.authoring.CustomCheck.checkCallback...  
        (system)), 'None', 'StyleOne');  
    rec.TitleTips = 'Example check for model configuration parameters';  
  
    % --- data file input parameters  
    rec.setInputParametersLayoutGrid([1 1]);  
    inputParam1 = ModelAdvisor.InputParameter;
```

```

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

% -- set fix operation
act = ModelAdvisor.Action;
act.setCallbackFcn(@(task)(Advisor.authoring.CustomCheck.actionCallback...
                    (task)));

act.Name = 'Modify Settings';
act.Description = 'Modify model configuration settings.';
rec.setAction(act);

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

%% defineModelAdvisorGroups
function defineModelAdvisorGroups
mdladvRoot = ModelAdvisor.Root;

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

```

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

```
Advisor.Manager.refresh_customizations
```

At the command prompt, type vdp.

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

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

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

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

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

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

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

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

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

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

Data File for Configuration Parameter Check

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


```

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

```

The `<messages>` tagging is optional.

`Advisor.authoring.generateConfigurationParameterDataFile` does not generate `<messages>` tagging. The `<messages>` tagging contains:

- `<Description>` - Description of the check. Displayed in Model Advisor window. Optional.
- `<PassMessage>` - Pass message displayed in Model Advisor window. Optional.
- `<FailMessage>` - Fail message displayed in Model Advisor window. Optional.
- `<RecommendedActions>` - Recommended actions displayed in Model Advisor window when check does not pass. Optional.

Within the `<checkdata>` tagging, the data file specifies two types of constraints:

- `<PositiveModelParameterConstraint>` - Specifies the configuration parameter setting that you want.
- `<NegativeModelParameterConstraint>` - Specifies the configuration parameter setting that you do not want.

Within the tagging for each of the two types of constraints, for each configuration parameter that you want to check, the data file has the following tags:

- `parameter` - Specifies the configuration parameter that you want to check. The tagging uses the command-line name for the configuration parameter. For example:
 - `<parameter>AlgebraicLoopMsg</parameter>`
 - `<parameter>BlockPriorityViolationMsg</parameter>`
- `value` - For `PositiveModelParameterConstraint` constraints, specifies the setting(s) that you want for the configuration parameter. For `NegativeModelParameterConstraint`, specifies the setting(s) you that do not want for the configuration parameter. You can specify more than one value tag. Values can have the following formats:
 - For a scalar value:
`<value>xyz</value>`
 - For a structure or object:
`<value><param1>xyz</param1><param2>yza</param2></value>`
 - For an array:
`<value><element>value</element><element>value</element></value>`

For example:

- `PositiveModelParameterConstraint` constraints that warn when the setting is not error:
`<value>error</value>`
- `NegativeModelParameterConstraint` constraints that warn when the setting is none or warning:
`<value>none</value>`
`<value>warning</value>`
- `PositiveModelParameterConstraint` constraints that warn when the setting is not a valid structure:
`<value>`
 `<double>a</double>`
 `<single>b</single>`
`</value>`

- `NegativeModelParameterConstraint` constraints that warn when the setting is an invalid array:

```
<value>
  <element>A</element>
  <element>B</element>
</value>
```

- `fixvalue` - Specifies the setting to use when applying the Model Advisor fix action. Optional. Fix values can have the following formats:

- For a scalar value:

```
<fixvalue>xyz</fixvalue>
```

- For a structure or object:

```
<fixvalue><param1>xyz</param1><param2>yza</param2></fixvalue>
```

- For an array:

```
<fixvalue><element>value</element><element>value</element></fixvalue>
```

For example:

- Fix action to modify configuration parameter setting to error:

```
<fixvalue>error</fixvalue>
```

- Fix action to modify configuration parameter setting to warning:

```
<fixvalue>warning</fixvalue>
```

- Fix action to modify configuration parameter setting to a structure:

```
<fixvalue>
  <double>a</double>
  <single>b</single>
</fixvalue>
```

- Fix action to modify configuration parameter setting to an array:

```
<fixvalue>
  <element>A</element>
  <element>B</element>
</fixvalue>
```

- `dependson` - Specifies a prerequisite subcheck. Optional. For example, to specify that configuration parameter subcheck `ID_B` must pass before configuration parameter subcheck `ID_A` is run, use this tagging:

```
<PositiveModelParameterConstraint id="ID_A">  
  <dependson>ID_B</value>  
</PositiveModelParameterConstraint>
```

Data file tagging specifying a configuration parameter

The following tagging specifies a subcheck for configuration parameter SolverType. If the configuration parameter is set to Fixed-Step, the subcheck passes.

```
<PositiveModelParameterConstraint id="ID_A">  
  <parameter>SolverType</parameter>  
  <value>Fixed-step</value>  
</PositiveModelParameterConstraint>
```

Data file tagging specifying configuration parameter with fix action

The following tagging specifies a subcheck for configuration parameter AlgebraicLoopMsg. If the configuration parameter is set to none or warning, the subcheck passes. If the subcheck does not pass, the check fix action modifies the configuration parameter to error.

```
<PositiveModelParameterConstraint id="ID_A">  
  <parameter>AlgebraicLoopMsg</parameter>  
  <value>none</value>  
  <value>warning</value>  
  <fixvalue>error</value>  
</PositiveModelParameterConstraint>
```

Data file tagging specifying an array type configuration parameter

```
<PositiveModelParameterConstraint id="A">  
  <parameter>ReservedNameArray</parameter>  
  <value>  
    <element>A</element>  
    <element>B</element>  
  </value>  
  <value>  
    <element>A</element>  
    <element>C</element>  
  </value>  
</PositiveModelParameterConstraint>
```

Data file tagging specifying a structure type configuration parameter with fix action

```
<PositiveModelParameterConstraint id="A">
  <parameter>ReplacementTypes</parameter>
  <value>
    <double>a</double>
    <single>b</single>
  </value>
  <value>
    <double>c</double>
    <single>b</single>
  </value>
  <fixvalue>
    <double>a</double>
    <single>b</single>
  </fixvalue>
</PositiveModelParameterConstraint>
```

Data file tagging specifying configuration parameter with fix action and prerequisite check

The following tagging specifies a subcheck for configuration parameter SolverType. The subcheck for SolverType runs only after the ID_B subcheck passes. If the ID_B subcheck does not pass, the subcheck for SolverType does not run. The Model Advisor reports that the prerequisite constraint is not met.

If the SolverType subcheck runs and SolverType is set to Fixed-Step, the SolverType subcheck passes. If the subcheck runs and does not pass, the check fix action modifies the configuration parameter to Fixed-Step.

```
<PositiveModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
  <fixvalue>Fixed-step</value>
  <dependson>ID_B</value>
</PositiveModelParameterConstraint>
```

Data file tagging specifying unwanted configuration parameter

The following tagging specifies a subcheck for configuration parameter SolverType. The subcheck does not pass if the configuration parameter is set to Fixed-Step.

```
<NegativeModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
```

```
<value>Fixed-step</value>
</NegativeModelParameterConstraint>
```

Data file tagging specifying unwanted configuration parameter with fix action

The following tagging specifies a subcheck for configuration parameter `SolverType`. If the configuration parameter is set to `Fixed-Step`, the subcheck does not pass. If the subcheck does not pass, the check fix action modifies the configuration parameter to `Variable-Step`.

```
<NegativeModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
  <fixvalue>Variable-step</value>
</NegativeModelParameterConstraint>
```

Data file tagging specifying unwanted configuration parameter with fix action and prerequisite check

The following tagging specifies a check for configuration parameter `SolverType`. The subcheck for `SolverType` runs only after the `ID_B` subcheck passes. If the `ID_B` subcheck does not pass, the subcheck for `SolverType` does not run. The Model Advisor reports that the prerequisite constraint is not met.

If the `SolverType` subcheck runs and `SolverType` is set to `Fixed-Step`, the subcheck does not pass. The check fix action modifies the configuration parameter to `Variable-Step`.

```
<NegativeModelParameterConstraint id="ID_A">
  <parameter>SolverType</parameter>
  <value>Fixed-step</value>
  <fixvalue>Variable-step</value>
  <dependson>ID_B</value>
</NegativeModelParameterConstraint>
```

See Also

`Advisor.authoring.CustomCheck.actionCallback` |
`Advisor.authoring.CustomCheck.checkCallback` |
`Advisor.authoring.DataFile.validate` |
`Advisor.authoring.generateConfigurationParameterDataFile`

More About

- “Organize and Deploy Model Advisor Checks”

Define Checks for Supported or Unsupported Blocks and Parameters

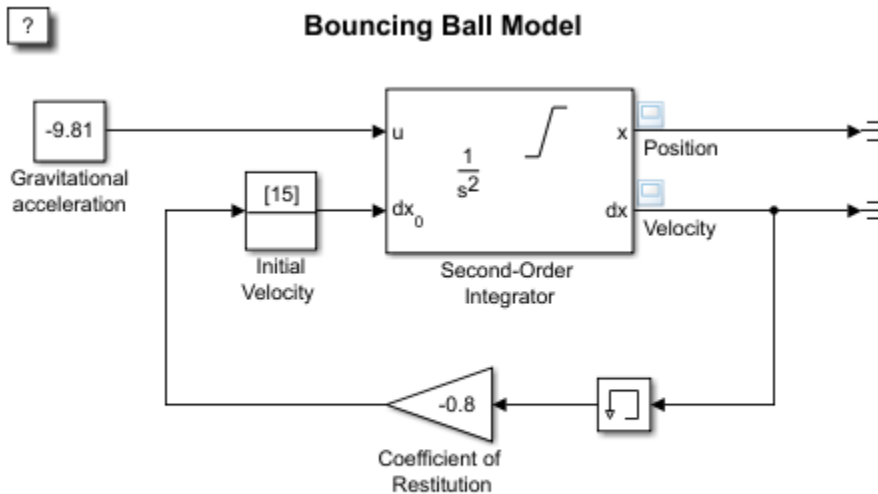
For modeling guidelines, such as MAAB or MISRA, that require you to use a subset of block or parameter values, you can create Model Advisor checks in which you specify these constraints:

- Supported or unsupported block parameter values
- Supported or unsupported model parameter values
- Supported or unsupported blocks
- Check for whether blocks or parameters meet a combination of constraints

You can also create constraints that check for prerequisite constraints before checking the actual constraint. You can check your model against these constraints as you edit or run the checks interactively after you complete your model design.

Example

The `sldemo_bounce` model simulates a ball bouncing on Earth. In this example, you create two Model Advisor checks consisting of constraints. Then, check your model against those constraints.



Copyright 2004-2013 The MathWorks, Inc.

Create Block Parameter Constraints

- 1 Create these block parameter constraints:

```
c1=Advisor.authoring.PositiveBlockParameterConstraint;
c1.ID='ID_1';
c1.BlockType='Gain';
c1.ParameterName='Gain';
c1.SupportedParameterValues={'-.7'};
c1.ValueOperator='eq';
```

```
c2=Advisor.authoring.NegativeBlockParameterConstraint;
c2.ID='ID_2';
c2.BlockType='InitialCondition';
c2.ParameterName='Value';
c2.UnsupportedParameterValues={'0'};
c2.ValueOperator='le';
```

Constraint c1 specifies that a Gain block must have a value equal to $-.7$. Constraint c2 specifies that the Initial Condition block must have a value less than or equal to zero.

- 2 Create this positive model parameter constraint.

```
c3=Advisor.authoring.PositiveModelParameterConstraint;
c3.ID='ID_3';
c3.ParameterName='SolverType';
c3.SupportedParameterValues={'Variable-step'};
```

Constraint c3 specifies that the **Solver** parameter must be equal to Variable-step.

- 3 Create this positive block type constraint:

```
c4=Advisor.authoring.PositiveBlockTypeConstraint;
c4.ID='ID_5';
s1=struct('BlockType','Constant','MaskType','');
s2=struct('BlockType','Subsystem','MaskType','');
s3=struct('BlockType','InitialCondition','MaskType','');
s4=struct('BlockType','Gain','MaskType','');
s5=struct('BlockType','Memory','MaskType','');
s6=struct('BlockType','SecondOrderIntegrator','MaskType','');
s7=struct('BlockType','Terminator','MaskType','');
c4.SupportedBlockTypes={s1;s2;s3;s4;s5;s6;s7};
c4.addPreRequisiteConstraintID('ID_3');
```

Constraint c4 specifies the supported blocks. Constraint c3 is a prerequisite to c4 meaning that the Model Advisor only checks c4 if c3 passes.

- 4 Create a data file that contains these constraints. This data file corresponds to one Model Advisor check.

```
Advisor.authoring.generateBlockConstraintsDataFile( ...
    'sldemo_constraints_1.xml','constraints',{c1,c2,c3,c4});
```

The data file contains tagging specifically for constraints.

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck>
  <checkdata>
    <PositiveBlockParameterConstraint BlockType="Gain" id="ID_1">
      <parameter type="string">Gain</parameter>
      <value>-.7</value>
      <operator>eq</operator>
    </PositiveBlockParameterConstraint>
    <NegativeBlockParameterConstraint BlockType="InitialCondition" id="ID_2">
      <parameter type="string">Value</parameter>
      <value>0</value>
      <operator>le</operator>
    </NegativeBlockParameterConstraint>
    <PositiveModelParameterConstraint id="ID_3">
      <parameter type="enum">SolverType</parameter>
      <value>Variable-step</value>
    </PositiveModelParameterConstraint>
    <PositiveBlockTypeConstraint id="ID_5">
```

```

    <BlockType MaskType="">Constant</BlockType>
    <BlockType MaskType="">Subsystem</BlockType>
    <BlockType MaskType="">InitialCondition</BlockType>
    <BlockType MaskType="">Gain</BlockType>
    <BlockType MaskType="">Memory</BlockType>
    <BlockType MaskType="">SecondOrderIntegrator</BlockType>
    <BlockType MaskType="">Terminator</BlockType>
    <dependson>ID_3</dependson>
  </PositiveBlockTypeConstraint>
  <CompositeConstraint>
    <ID>ID_1</ID>
    <ID>ID_2</ID>
    <ID>ID_5</ID>
    <operator>and</operator>
  </CompositeConstraint>
</checkdata>
</customcheck>

```

Note For model configuration parameter constraints, use the `Advisor.authoring.generateBlockConstraintsDataFile` method only when specifying model configuration parameter constraints as prerequisites to block constraints or as part of a composite constraint consisting of block and model configuration parameter constraints. For other cases, use the `Advisor.authoring.generateConfigurationParameterDatafile` method.

- 5 Create two block parameter constraints and a composite constraint.

```

cc1=Advisor.authoring.PositiveBlockParameterConstraint;
cc1.ID='ID_cc1';
cc1.BlockType='SecondOrderIntegrator';
cc1.ParameterName='UpperLimitX';
cc1.SupportedParameterValues={'inf'};
cc1.ValueOperator='eq';

```

```

cc2=Advisor.authoring.PositiveBlockParameterConstraint;
cc2.ID='ID_cc2';
cc2.BlockType='SecondOrderIntegrator';
cc2.ParameterName='LowerLimitX';
cc2.SupportedParameterValues={'0.0'};
cc2.ValueOperator='eq';

```

```

cc=Advisor.authoring.CompositeConstraint;
cc.addConstraintID('ID_cc1');
cc.addConstraintID('ID_cc2');
cc.CompositeOperator='and';

```

Constraint `cc1` specifies that for a Second-Order Integrator block, the **Upper limit x** parameter must have a value equal to `inf`. Constraint `cc2` specifies that for a

Second-Order Integrator block, the **Lower limit x** parameter must have a value equal to zero. Constraint cc specifies that for this check to pass, both cc1 and cc2 have to pass.

- 6 Create a data file that contains these constraints. This data file corresponds to a second Model Advisor check.

```
Advisor.authoring.generateBlockConstraintsDataFile( ...
    'sldemo_constraints_2.xml', 'constraints', {cc1, cc2, cc});
```

Create Model Advisor Checks from Constraints

- 1 To specify and register these checks, use this `sl_customization.m` file.

```
function sl_customization(cm)

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

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

% defineModelAdvisorChecks
function defineModelAdvisorChecks

% check1
rec = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0001');
rec.Title = 'Example1: Check block parameter constraints';
rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec.TitleTips = 'Example check block parameter constraints';

% --- data file input parameters
rec.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Data File';
inputParam1.Value = 'sldemo_constraints_1.xml';
inputParam1.Type = 'String';
inputParam1.Description = 'Name or full path of XML data file.';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec.setInputParameters({inputParam1});
rec.SupportExclusion = false;
rec.SupportLibrary = true;

% check2
rec1 = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0002');
rec1.Title = 'Example2: Check block parameter constraints';
rec1.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec1.TitleTips = 'Example check block parameter constraints';

% --- data file input parameters
rec1.setInputParametersLayoutGrid([1 1]);
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Data File';
```

```

inputParam1.Value = 'sldemo_constraints_2.xml';
inputParam1.Type = 'String';
inputParam1.Description = 'Name or full path of XML data file.';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
rec1.setInputParameters({inputParam1});
rec1.SupportExclusion = false;
rec1.SupportLibrary = true;
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(rec);
mdladvRoot.register(rec1);

%% defineModelAdvisorGroups
function defineModelAdvisorGroups
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group 1
rec = ModelAdvisor.FactoryGroup('com.mathworks.Test.factoryGroup');
rec.DisplayName='Example: My Group';
rec.addCheck('mathworks.check_0001');
rec.addCheck('mathworks.check_0002');

mdladvRoot.publish(rec);

```

You must use the `Advisor.authoring.createBlockConstraintCheck` function to create the `ModelAdvisor.Check` object and specify the constraint data file as an input parameter to this object.

- 2 At the command prompt, type **create the Example1: Check block parameter constraints** and **Example2: Check block parameter constraints** checks by typing this command:

```
Advisor.Manager.refresh_customizations
```

- 3 At the command prompt, type `sldemo_bounce`.
- 4 To open the Model Advisor, from the model window, select **Analysis > Model Advisor > Model Advisor**
- 5 In the left pane, select **By Task > Example: My Group**. For each check, in the right pane, the **Data File** parameters are set to the data files that you previously created.
- 6 Click **Run Selected Checks**.
- 7 The **Example1: Check block parameter constraints** check produces a warning because the Gain block has a value of -0.8 not -0.7 . The **Example2: Check block parameter constraints** check passes because the Second-Order Integrator block meets both constraints.

See Also

`Advisor.authoring.generateBlockConstraintsDataFile` |
`NegativeBlockParameterConstraint` | `NegativeBlockTypeConstraint` |

NegativeModelParameterConstraint | PositiveBlockParameterConstraint |
PositiveBlockTypeConstraint | PositiveModelParameterConstraint

Register Checks

Create `sl_customization` Function

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

Tip

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

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

The customization manager object includes methods for registering custom checks. Use these methods to register customizations specific to your application, as described in the following sections.

Register Checks

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

- `addModelAdvisorCheckFcn (@checkDefinitionFcn)`

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

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

This example shows how to register custom checks:

```
function sl_customization(cm)

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

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

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

See Also

`ModelAdvisor.Check`

Related Examples

- Registering Tasks and Folders on page 8-12

More About

- “Define Custom Checks” on page 7-40

Define Startup and Post-Execution Actions Using Process Callback Functions

The process callback function is an optional function that you use to configure the Model Advisor and process check results at run time. The process callback function specifies actions that the software performs at different stages of Model Advisor execution:

- **configure stage:** The Model Advisor executes `configure` actions at startup, after checks and tasks have been initialized. At this stage, you can customize how the Model Advisor constructs lists of checks and tasks by modifying `Visible`, `Enable`, and `Value` properties. For example, you can remove, rename, and selectively display checks and tasks in the **By Task** folder.
- **process_results stage:** The Model Advisor executes `process_results` actions after checks complete execution. You can specify actions to examine and report on the results returned by check callback functions.

Process Callback Function Arguments

The process callback function uses the following arguments.

Argument	I/O Type	Data Type	Description
<code>stage</code>	Input	Enumeration	Specifies the stages at which process callback actions are executed. Use this argument in a switch statement to specify actions for the stages <code>configure</code> and <code>process_results</code> .
<code>system</code>	Input	Path	Model or subsystem that the Model Advisor analyzes.
<code>checkCellArray</code>	Input/Output	Cell array	As input, the array of checks constructed in the check definition function. As output, the array of checks modified by actions in the <code>configure</code> stage.

Argument	I/O Type	Data Type	Description
taskCellArray	Input/Output	Cell array	As input, the array of tasks constructed in the task definition function. As output, the array of tasks modified by actions in the configure stage.

Process Callback Function

This example shows a process callback function that specifies actions in the `configure` stage that makes only custom checks visible. In the `process_results` stage, this function displays information at the command prompt for checks that do not pass.

```
% Process Callback Function
% Defines actions to execute at startup and post-execution
function [checkCellArray taskCellArray] = ...
    ModelAdvisorProcessFunction(stage, system, checkCellArray, taskCellArray)
switch stage
    % Specify the appearance of the Model Advisor window at startup
    case 'configure'
        for i=1:length(checkCellArray)
            % Hide all checks that do not belong to custom folder
            if isempty(strfind(checkCellArray{i}.ID, 'mathworks.example'))
                checkCellArray{i}.Visible = false;
                checkCellArray{i}.Value = false;
            end
        end
    end
    % Specify actions to perform after the Model Advisor completes execution
    case 'process_results'
        for i=1:length(checkCellArray)
            % Print message if check does not pass
            if checkCellArray{i}.Selected && (strcmp(checkCellArray{i}.Title, ...
                'Check Simulink window screen color'))
                mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
                % Verify whether the check was run and if it failed
                if mdladvObj.verifyCheckRan(checkCellArray{i}.ID)
                    if ~mdladvObj.getCheckResultStatus(checkCellArray{i}.ID)
                        % Display text in MATLAB Command Window
                        disp(['Example message from Model Advisor Process'...
                            ' callback.']);
                    end
                end
            end
        end
    end
end
end
end
end
```

Tips for Using the Process Callback Function in a `sl_customization` File

Observe the following tips when using process callback function in a `sl_customization` file:

- If you delete a check in the Model Advisor Configuration Editor, you can retrieve a copy of it from the Model Advisor Check Browser. However, if you use a process callback function in a `sl_customization` file to hide checks and folders, the Model Advisor Configuration Editor and Model Advisor Check Browser do not display the hidden checks and folders. For a complete list of checks and folders, remove process callback functions and update the Simulink environment.
- The Model Advisor registers only one process callback function. If you have more than one `sl_customization.m` file on your MATLAB path, the Model Advisor registers the process callback function from the `sl_customization.m` file that has the highest priority.
- If you add process callbacks within the `sl_customization.m` file, include methods for registering the process callbacks in the `sl_customization` function.

See Also

“Create Model Advisor Checks Workflow” on page 7-2 | “Register Checks” on page 7-35 | “Organize Customization File Checks and Folders” on page 8-11 | “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5

Define Custom Checks

About Custom Checks

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

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

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

Contents of Check Definitions

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

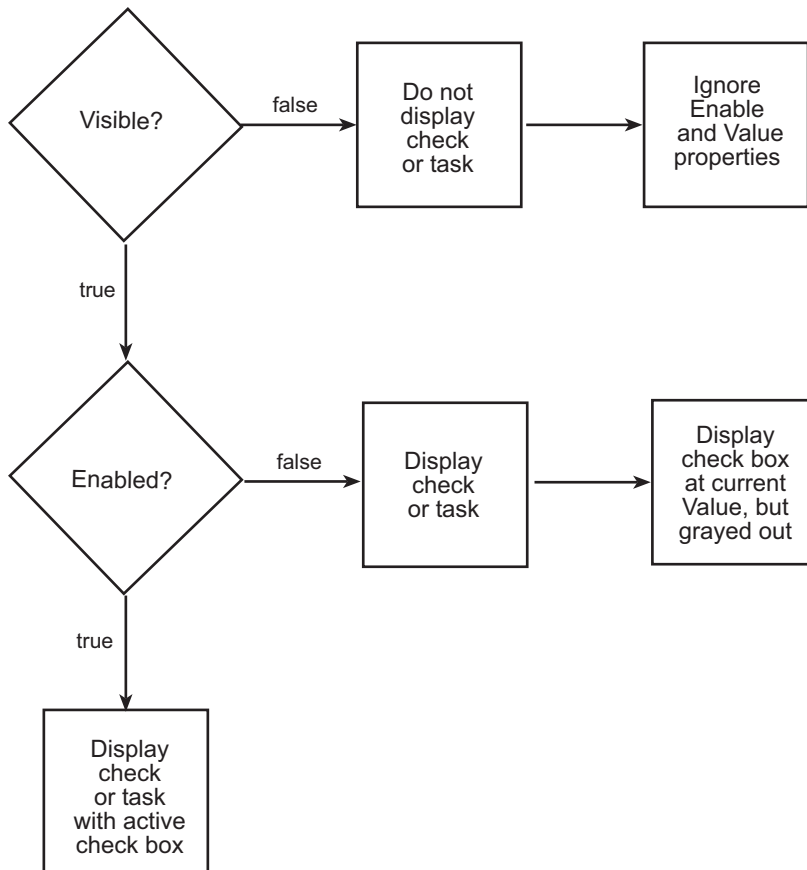
Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this id to access the check.
Handle to check callback function (required)	Function that specifies the contents of a check.
Check name (recommended)	Creates a name for the check that the Model Advisor displays.
Check properties (optional)	Creates a user interface with the check. When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for <code>Visible</code> and <code>LicenseName</code> .
Input Parameters (optional)	Adds input parameters that request input from the user. The Model Advisor uses the input to perform the check.
Action (optional)	Adds automatic fixing action.
Explore Result button (optional)	Adds the Explore Result button that the user clicks to open the Model Advisor Result Explorer.

Display and Enable Checks

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

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

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



Define Where Custom Checks Appear

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

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

Check Definition Function

This example shows a function that defines the custom checks associated with the callback functions described in “Create Callback Functions and Results” on page 7-48. The check definition function returns a cell array of custom checks to be added to the Model Advisor.

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

```
% Defines custom Model Advisor checks
function defineModelAdvisorChecks

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

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

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

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

Define Check Input Parameters

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

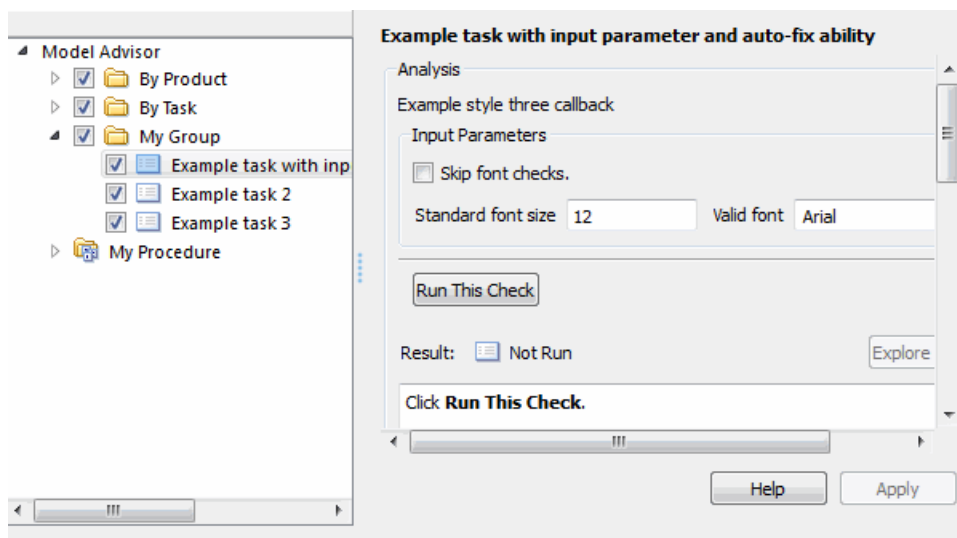
Specify the layout of input parameters with the following methods.

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

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

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.setInputParametersLayoutGrid([3 2]);
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
inputParam1.Description = 'sample tooltip';
inputParam1.setRowSpan([1 1]);
inputParam1.setColSpan([1 1]);
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
inputParam3.setRowSpan([2 2]);
inputParam3.setColSpan([2 2]);
rec.setInputParameters({inputParam1,inputParam2,inputParam3});
```


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



Define Model Advisor Result Explorer Views

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

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

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

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

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

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

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

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

Define Check Actions

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

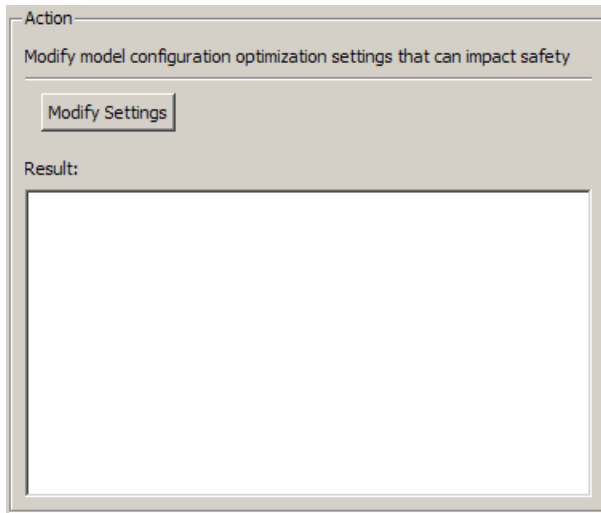
You define actions using the `ModelAdvisor.Action` class inside a custom check function. You must define:

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

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

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

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



See Also

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

Related Examples

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

More About

- “Batch-Fix Warnings or Failures” (Simulink)
- “Create Callback Functions and Results” on page 7-48
- Defining Custom Groups on page 8-13
- “Register Checks” on page 7-35

Create Callback Functions and Results

About Callback Functions

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

To	See
Create an <i>informational callback function</i> for a custom check that finds and displays the model configuration and checksum information.	"Informational Check Callback Function" on page 7-49
Create a <i>simple callback function</i> that indicates if the model passed a check, or to recommend fixing the issue.	"Simple Check Callback Function" on page 7-50
Create a <i>detailed check callback function</i> to return and organize results as strings in a layered, hierarchical fashion.	"Detailed Check Callback Function" on page 7-51
Create a callback function that automatically displays hyperlinks for every object returned by the check.	"Check Callback Function with Hyperlinked Results" on page 7-52
Create an <i>action callback function</i> that specifies the actions that the Model Advisor performs on a model or subsystem when you click the action button.	"Action Callback Function" on page 7-55
Create a callback function for a custom check with two subchecks.	"Check With Subchecks and Actions" on page 7-56
Create a callback function for a custom basic check with pass/fail status.	"Basic Check with Pass/Fail Status" on page 7-58

Informational Check Callback Function

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

An informational check includes the following items in the results:

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

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

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

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

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

% Format results in a list using Model Advisor Result Template API
ft = ModelAdvisor.FormatTemplate('ListTemplate');
% Add See Also section for references to standards
docLinkSfunction{1} = {'IEC 61508-3, Table A.8 (5)' ...
    ' 'Software configuration management' '};
setRefLink(ft,docLinkSfunction);

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

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% If err, use these values
```

```

mdlver = 'Error - could not retrieve Version';
mdlauthor = 'Error - could not retrieve Author';
mdldate = 'Error - could not retrieve Date';
mdlsum = 'Error - could not retrieve CheckSum';

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

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

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

```

Simple Check Callback Function

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

The check callback function takes the following arguments.

Argument	I/O Type	Description
system	Input	Path to the model or subsystem analyzed by the Model Advisor.
result	Output	MATLAB character vector that supports Model Advisor Formatting API on page 7-65 calls or embedded HTML tags for text formatting.

Detailed Check Callback Function

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

The detailed callback function takes the following arguments.

Argument	I/O Type	Description
<code>system</code>	Input	Path to the model or system analyzed by the Model Advisor.
<code>ResultDescription</code>	Output	Cell array of MATLAB character vectors that supports Model Advisor Formatting API on page 7-65 calls or embedded HTML tags for text formatting. The Model Advisor concatenates the <code>ResultDescription</code> character vector with the corresponding array of <code>ResultDetails</code> character vectors.
<code>ResultDetails</code>	Output	Cell array of cell arrays, each of which contains one or more character vectors.

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

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

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

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

% Check Simulation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check Simulation '...
'optimization settings:']);
```

```

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

% Check code generation optimization setting
ResultDescription{end+1} = ModelAdvisor.Paragraph(['Check code generation '...
    'optimization settings:']);
ResultDetails{end+1} = {};
if strcmp(get_param(model, 'LocalBlockOutputs'), 'off');
    ResultDetails{end}{end+1} = ModelAdvisor.Text(['It is recommended to'...
        'turn on Enable local block outputs option.'], {'italic'});
    ResultDetails{end}{end+1} = ModelAdvisor.LineBreak;
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if strcmp(get_param(model, 'BufferReuse'), 'off');
    ResultDetails{end}{end+1} = ModelAdvisor.Text(['It is recommended to'...
        'turn on Reuse block outputs option.'], {'italic'});
    mdladvObj.setCheckResultStatus(false); % set to fail
    mdladvObj.setActionEnable(true);
end
if isempty(ResultDetails{end})
    ResultDetails{end}{end+1} = ModelAdvisor.Text('Passed', {'pass'});
end

```

Check Callback Function with Hyperlinked Results

This example shows how to create a callback function with hyperlinked results. This callback function automatically displays hyperlinks for every object returned by the check so that you can easily locate problem areas in your model or subsystem. The keyword for this type of callback function is `StyleThree`. The check definition requires this keyword.

This callback function takes the following arguments.

Argument	I/O Type	Description
system	Input	Path to the model or system analyzed by the Model Advisor.
ResultDescription	Output	Cell array of MATLAB character vectors that supports the Model Advisor Formatting API calls or embedded HTML tags for text formatting.

Argument	I/O Type	Description
ResultDetails	Output	Cell array of cell arrays, each of which contains one or more Simulink objects such as blocks, ports, lines, and Stateflow charts. The objects must be in the form of a handle or Simulink path.

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

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

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

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

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

```

% find all blocks inside current system
allBlks = find_system(system);

% block diagram doesn't have font property
% get blocks inside current system that have font property
allBlks = setdiff(allBlks, {system});

% find regular font name blocks
regularBlks = find_system(allBlks, 'FontName', regularFontName);

% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font for blocks for a uniform appearance in the model. '...
        'The following blocks use a font other than ' regularFontName ': ']);
    ResultDetails{end+1} = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult, 'object');
    myLVParam.Attributes = {'FontName'}; % name is default property
    mdladvObj.setListViewParameters({myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font names '...
        'are identical.']);
    ResultDetails{end+1} = {};
end

% find regular font size blocks
regularBlks = find_system(allBlks, 'FontSize', regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
if ~isempty(searchResult)
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['It is recommended to '...
        'use same font size for blocks for a uniform appearance in the model. '...
        'The following blocks use a font size other than ' ...
        num2str(regularFontSize) ': ']);
    ResultDetails{end+1} = searchResult;
    mdladvObj.setCheckResultStatus(false);
    myLVParam = ModelAdvisor.ListViewParameter;
    myLVParam.Name = 'Invalid font size blocks'; % pull down filter name
    myLVParam.Data = get_param(searchResult, 'object');
    myLVParam.Attributes = {'FontSize'}; % name is default property
    mdladvObj.setListViewParameters...
        ({mdladvObj.getListViewParameters{:}, myLVParam});
    needEnableAction = true;
else
    ResultDescription{end+1} = ModelAdvisor.Paragraph(['All block font sizes '...
        'are identical.']);
    ResultDetails{end+1} = {};
end

```

```
mdladvObj.setActionEnable(needEnableAction);
mdladvObj.setCheckErrorSeverity(1);
```

In the Model Advisor, if you run **Example task with input parameter and auto-fix ability** for the `slvndemo_mdladv` model, you can view the hyperlinked results. Clicking the first hyperlink, `slvndemo_mdladv/Input`, displays the Simulink model with the Input block highlighted.

Action Callback Function

This example shows how to create an action callback function. An action callback function specifies the actions that the Model Advisor performs on a model or subsystem when the user clicks the action button. You must create one callback function for the action that you want to take.

The action callback function takes the following arguments.

Argument	I/O Type	Description
<code>taskobj</code>	Input	The <code>ModelAdvisor.Task</code> object for the check that includes an action definition.
<code>result</code>	Output	MATLAB character vector that supports Model Advisor Formatting API on page 7-65 calls or embedded HTML tags for text formatting.

This example shows an action callback function that contains result details.

```
% Get Simulink.ModelAdvisor object
mdladvObj = taskobj.MAObj;
% get result of 'MyCheck'
myResult = mdladvObj.getCheckResult('MyCheck');
% if the check is in style three
[ResultDescription, ResultDetails] = myResult;
```

This example shows an action callback function that fixes the optimization settings that the Model Advisor reviews as defined in “Check With Subchecks and Actions” on page 7-56.

```
% Sample Check 3 Action Callback Function: Check with Subresults and Actions
% Fix optimization settings
function result = modifyOptimizationSetting(taskobj)
% Initialize variables
result = ModelAdvisor.Paragraph();
mdladvObj = taskobj.MAObj;
system = bdroot(mdladvObj.System);
```

```
% 'Block reduction' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'BlockReduction'),'off')
    set_param(system,'BlockReduction','off');
    result.addItem(ModelAdvisor.Text( ...
        'Cleared the ''Block reduction'' check box.', {'Pass'}));
    result.addItem(ModelAdvisor.LineBreak);
end
% 'Conditional input branch execution' is selected
% Clear the check box and display text describing the change
if ~strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    set_param(system,'ConditionallyExecuteInputs','off');
    result.addItem(ModelAdvisor.Text( ...
        'Cleared the ''Conditional input branch execution'' check box.', ...
        {'Pass'}));
end
```

Action Callback Function with Result Details

This example shows an action callback function that contains result details.

```
% Get Simulink.ModelAdvisor object
mdladvObj = taskobj.MAObj;
% get result of 'MyCheck'
myResult = mdladvObj.getCheckResult('MyCheck');
% if the check is in style three
[ResultDescription, ResultDetails] = myResult;
```

Check With Subchecks and Actions

This example shows how to create a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting, and the second reviews the **Conditional input branch execution** optimization setting.

A check with subchecks includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.

- Recommended actions to take when the subcheck does not pass.
- A line between the subcheck results.

```

% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
function ResultDescription = OptimizationSettingCallback(system)
% Initialize variables
system = getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};
system = bdroot(system);

% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,['Check model configuration for optimization settings that'...
    'can impact safety.']);

% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction setting');
setInformation(ft1,'Check whether the ''Block reduction'' check box is cleared. ');
% Add See Also section with references to applicable standards
docLinks{1} = {'Reference D0-178B Section 6.3.4e - Source code ' ...
    'is traceable to low-level requirements'}];
% Review 'Block reduction' optimization
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is cleared. ');
    ResultStatus = true;
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
    setSubResultStatusText(ft1,'The ''Block reduction'' check box is selected. ');
    setRecAction(ft1,['Clear the ''Optimization > Block reduction'' ...
        ' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check whether the ''Conditional input branch execution''...
    ' check box is cleared.'])
% Add See Also section and references to applicable standards
docLinks{1} = {'Reference D0-178B Section 6.4.4.2 - Test coverage ' ...

```

```
    'of software structure is achieved']});
setRefLink(ft2,docLinks);

% Last subcheck, suppress line
setSubBar(ft2,false);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution'' ...
        'check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The ''Conditional input branch execution''...
        'check box is selected.']);
    setRecAction(ft2,['Clear the ''Optimization > Conditional input branch ' ...
        'execution'' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);
```

Basic Check with Pass/Fail Status

This example shows a callback function for a custom basic check that finds and reports unconnected lines, input ports, and output ports.

A basic check includes the following items in the results:

- A description of what the check is reviewing.
- References to standards, if applicable.
- The status of the check.
- A description of the status.
- Results for the check.
- The recommended actions to take when the check does not pass.

A basic check does not include the following items in the results:

- Subcheck results.

- A line below the results.

```

% Sample Check 2 Callback Function: Basic Check with Pass/Fail Status
% Find and report unconnected lines, input ports, and output ports
function ResultDescription = unconnectedObjectsCallbackUsingFT(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
% Initialize variables
mdladvObj.setCheckResultStatus(false);
ResultDescription = {};
ResultStatus = false; % Default check status is 'Warning'
system = getfullname(system);
isSubsystem = ~strcmp(bdroot(system), system);

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

% Description of check in results
if isSubsystem
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                   'output ports in the subsystem.'];
else
    checkDescStr = ['Identify unconnected lines, input ports, and ' ...
                   'output ports in the model.'];
end
setCheckText(ft,checkDescStr);

% Add See Also section with references to applicable standards
checkStdRef = 'IEC 61508-3, Table A.3 (3) 'Language subset' ';
docLinkSfunction{1} = {checkStdRef};
setRefLink(ft,docLinkSfunction);

% Basic checks do not have subresults, suppress line
setSubBar(ft,false);

% Check for unconnected lines, inputs, and outputs
sysHandle = get_param(system, 'Handle');
uLines = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'line', ...
    'Connected', 'off');
uPorts = find_system(sysHandle, ...
    'Findall', 'on', ...
    'LookUnderMasks', 'on', ...
    'Type', 'port', ...
    'Line', -1);

% Use parents of port objects for the correct highlight behavior
if ~isempty(uPorts)
    for i=1:length(uPorts)
        uPorts(i) = get_param(get_param(uPorts(i), 'Parent'), 'Handle');
    end
end
end

```

```
% Create cell array of unconnected object handles
modelObj = {};
searchResult = union(uLines, uPorts);
for i = 1:length(searchResult)
    modelObj{i} = searchResult(i);
end

% No unconnected objects in model
% Set result status to 'Pass' and display text describing the status
if isempty(modelObj)
    setSubResultStatus(ft,'Pass');
    if isSubsystem
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this subsystem.']);
    else
        setSubResultStatusText(ft,['There are no unconnected lines, ' ...
            'input ports, and output ports in this model.']);
    end
    ResultStatus = true;
% Unconnected objects in model
% Set result status to 'Warning' and display text describing the status
else
    setSubResultStatus(ft,'Warn');
    if ~isSubsystem
        setSubResultStatusText(ft,['The following lines, input ports, ' ...
            'or output ports are not properly connected in the system: ' system]);
    else
        setSubResultStatusText(ft,['The following lines, input ports, or ' ...
            'output ports are not properly connected in the subsystem: ' system]);
    end
    % Specify recommended action to fix the warning
    setRecAction(ft,'Connect the specified blocks. ');
    % Create a list of handles to problem objects
    setListObj(ft,modelObj);
    ResultStatus = false;
end
% Pass the list template object to the Model Advisor
ResultDescription{end+1} = ft;
% Set overall check status
mdladvObj.setCheckResultStatus(ResultStatus);
```

See Also

[ModelAdvisor.Check](#) | [ModelAdvisor.FormatTemplate](#) | [ModelAdvisor.Task](#)

More About

- Defining Custom Groups on page 8-13
- “Define Custom Checks” on page 7-40
- “Format Check Results” on page 7-65

- “Register Checks” on page 7-35

Exclude Blocks From Custom Checks

This example shows how to exclude blocks from custom checks. To save time during model development and verification, you might decide to exclude individual blocks from custom checks in a Model Advisor analysis. To exclude custom checks from Simulink blocks and Stateflow charts, use the `ModelAdvisor.Check.supportExclusion` and `Simulink.ModelAdvisor.filterResultWithExclusion` functions in the `sl_customization.m` file.

Update the `sl_customization.m` File

- 1 To open the example model, at the command prompt, type `slvnvdemo_mdadv`.
- 2 In the model window, double-click **View demo `sl_customization.m`**.
- 3 To exclude the custom check **Check Simulink block font** from blocks during Model Advisor analysis, make three modifications to the `sl_customization.m` file.
 - a Enable the **Check Simulink block font** check to support check exclusions by using the `ModelAdvisor.Check.supportExclusion` property. You can now exclude the check from model blocks. After `rec.setInputParametersLayoutGrid([3 2]);`, add `rec.supportExclusion = true;`. The check 1 section of the function `defineModelAdvisorChecks` now looks like:

```
% --- sample check 1
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
rec.setCallbackFcn(@SampleStyleThreeCallback, 'None', 'StyleThree');
rec.setInputParametersLayoutGrid([3 2]);
rec.supportExclusion = true;
```
 - b Use the `Simulink.ModelAdvisor.filterResultWithExclusion` function to filter model objects causing a check warning or failure with checks that have exclusions enabled. To do this, there are two locations in the `sl_customization.m` file to modify, both in the `[ResultDescription, ResultDetails] = SampleStyleThreeCallback(system)` function:
 - After both instances of

```
searchResult = setdiff(allBlks, regularBlks);
```

add

```
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
```

- In the first location, the function now looks like:

```
% find regular font name blocks
regularBlks = find_system(allBlks, 'FontName', regularFontName);
```


```
% look for different font blocks in the system
searchResult = setdiff(allBlks, regularBlks);
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
if ~isempty(searchResult)
```

- In the second location, the function now looks like:

```
% find regular font size blocks
regularBlks = find_system(allBlks, 'FontSize', regularFontSize);
% look for different font size blocks in the system
searchResult = setdiff(allBlks, regularBlks);
searchResult = mdladvObj.filterResultWithExclusion(searchResult);
if ~isempty(searchResult)
```

- 4 Save the `sl_customization.m` file. If you are asked if it is ok to overwrite the file, click **OK**.

Create and Save Exclusions

- 1 In the model window, double-click **Launch Model Advisor**.
- 2 If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.
- 3 In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check fails.
- 4 In the Model Advisor window, click the **Enable highlighting** button . The blocks causing the **Check Simulink block font** check failure are highlighted in yellow.
- 5 In the model window, right-click the X block and select **Model Advisor > Exclude block only > Check Simulink block font**.
- 6 In the Model Advisor Exclusion Editor, click **OK** to create the exclusion file.
- 7 In the model window, right-click the Input block and select **Model Advisor > Exclude block only > Check Simulink block font**.

Review Exclusions

- 1 In the Model Advisor Exclusion Editor, click **OK** to update the exclusion file.
- 2 In the left pane of the Model Advisor window, select the **By Product > Demo > Check Simulink block font** check. In the right pane, select **Run This Check**. The check now passes. In the right-pane of the Model Advisor window, you can see the **Check Exclusion Rules** that the Model Advisor during the analysis.
- 3 Close `slvndemo_mdadv`.

See Also

`Simulink.ModelAdvisor | ModelAdvisor.Check.supportExclusion`

Related Examples

- “Select and Run Model Advisor Checks” (Simulink)
- Example of Excluding Gain and Outport Blocks From Checks on page 3-35

More About

- Excluding Blocks From Model Advisor Checks on page 3-26
- “Run Model Checks” (Simulink)
- “Address Model Check Results with Highlighting” (Simulink)

Format Check Results

Format Results

You can make the analysis results of your custom checks appear similar to each other with minimal scripting using the `ModelAdvisor.FormatTemplate` class.

If this format template does not meet your needs, or if you want to format action results, use the Model Advisor Formatting API to produce formatted outputs in the Model Advisor. The following constructors of the `ModelAdvisor` class allow you to format the output.

Constructor	Description
<code>ModelAdvisor.Text</code>	Create Model Advisor text output.
<code>ModelAdvisor.List</code>	Create list.
<code>ModelAdvisor.Table</code>	Create table.
<code>ModelAdvisor.Paragraph</code>	Create and format paragraph.
<code>ModelAdvisor.LineBreak</code>	Insert line break.
<code>ModelAdvisor.Image</code>	Include image in Model Advisor output.

Format Text

Text is the simplest form of output. You can format text in many different ways. The default text formatting is:

- Empty
- Default color (black)
- Unformatted (not bold, italicized, underlined, linked, subscripted, or superscripted)

To change text formatting, use the `ModelAdvisor.Text` constructor. When you want one type of formatting for all text, use this syntax:

```
ModelAdvisor.Text(content, {attributes})
```

When you want multiple types of formatting, you must build the text.

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
```

```
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' for a uniform appearance in the model.');
```

```
result = [t1, t2, t3, t4, t5];
```

Add ASCII and Extended ASCII characters using the MATLAB `char` command. For more information, see the `ModelAdvisor.Text` class page.

Format Lists

You can create two types of lists: numbered and bulleted. The default list formatting is bulleted. Use the `ModelAdvisor.List` constructor to create and format lists. You can create lists with indented subsections, formatted as either numbered or bulleted.

```
subList = ModelAdvisor.List();
subList.setType('numbered');
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass', 'bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass', 'bold'}));
```

```
topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1', {'keyword', 'bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2', {'keyword', 'bold'}), subList]);
```

Format Tables

The default table formatting is:

- Default color (black)
- Left justified
- Bold title, row, and column headings

Change table formatting using the `ModelAdvisor.Table` constructor.

This example creates a subtable within a table.

```
table1 = ModelAdvisor.Table(1,1);
table2 = ModelAdvisor.Table(2,3);
table2.setHeading('Table 2');
table2.setHeadingAlign('center');
table2.setColHeading(1, 'Header 1');
table2.setColHeading(2, 'Header 2');
table2.setColHeading(3, 'Header 3');
table1.setHeading('Table 1');
table1.setEntry(1,1,table2);
```

Table 1				
Table 2				
Header 1	Header 2	Header 3		

This example creates a table with five rows and five columns containing randomly generated numbers. Use the MATLAB code in a callback function to create the table. The Model Advisor displays `table1` in the results.

```
% ModelAdvisor.Table example

matrixData = rand(5,5) * 10^5;

% initialize a table with 5 rows and 5 columns (heading rows not counting)
table1 = ModelAdvisor.Table(5,5);

% set column headings
for n=1:5
    table1.setColHeading(n, ['Column ', num2str(n)]);
end

% set alignment of second column heading
table1.setColHeadingAlign(2, 'center');

% set column width of second column
table1.setColWidth(2, 3);

% set row headings
for n=1:5
    table1.setRowHeading(n, ['Row ', num2str(n)]);
end

% set Table content
for rowIndex=1:5
    for colIndex=1:5
        table1.setEntry(rowIndex, colIndex, ...
            num2str(matrixData(rowIndex, colIndex)));

        % set alignment of entries in second row
        if colIndex == 2
            table1.setEntryAlign(rowIndex, colIndex, 'center');
        end
    end
end

% overwrite content of cell 3,3 with a ModelAdvisor.Text
```

```
text = ModelAdvisor.Text('Example Text');
table1.setEntry(3,3, text)
```

	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	81472.3686	9754.0405	15761.3082	14188.6339	65574.0699
Row 2	90579.1937	27849.8219	97059.2782	42176.1283	3571.1679
Row 3	12698.6816	54688.1519	Example Text	91573.5525	84912.9306
Row 4	91337.5856	95750.6835	48537.5649	79220.733	93399.3248
Row 5	63235.9246	96488.8535	80028.0469	95949.2426	67873.5155

Format Paragraphs

You must handle paragraphs explicitly because most markup languages do not support line breaks. The default paragraph formatting is:

- Empty
- Default color (black)
- Unformatted, (not bold, italicized, underlined, linked, subscripted, or superscripted)
- Aligned left

If you want to change paragraph formatting, use the `ModelAdvisor.Paragraph` class.

Formatted Output

The following is the example from “Simple Check Callback Function” on page 7-50, reformatted using the Model Advisor Formatting API.

```
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
if strcmp(get_param(bdroot(system), 'ScreenColor'),'white')
    result = ModelAdvisor.Text('Passed',{ 'pass'});
    mdladvObj.setCheckResultStatus(true);
else
    msg1 = ModelAdvisor.Text(...
        ['It is recommended to select a Simulink window screen color'...
        ' of white for a readable and printable model. Click ']);
    msg2 = ModelAdvisor.Text('here');
    msg2.setHyperlink('matlab: set_param(bdroot, 'ScreenColor', 'white')');
    msg3 = ModelAdvisor.Text(' to change screen color to white. ');
    result = [msg1, msg2, msg3];
```



```
    mdladvObj.setCheckResultStatus(false);  
end
```

Format Linebreaks

You can add a line break between two lines of text with the `ModelAdvisor.LineBreak` constructor.

```
result = ModelAdvisor.Paragraph;  
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

Format Images

To include an image in Model Advisor output, use the `ModelAdvisor.Image` constructor. To create an `Image` object, use this syntax.

```
image_obj = ModelAdvisor.Image;
```

See Also

[ModelAdvisor.Check](#) | [ModelAdvisor.FormatTemplate](#) | [ModelAdvisor.Task](#)

Related Examples

- “Simple Check Callback Function” on page 7-50

More About

- Defining Custom Groups on page 8-13
- “Define Custom Checks” on page 7-40

Create Custom Configurations by Organizing Checks and Folders

Create Custom Configurations

You can use the Model Advisor APIs and Model Advisor Configuration Editor available with Simulink Check to do the tasks listed in the following table.

To	See
Create custom configurations by organizing Model Advisor checks and folders.	"Organize Checks and Folders Using the Model Advisor Configuration Editor" on page 8-5
Specify the order in which you make changes to your model.	"Create Procedural-Based Configurations" on page 9-5
Deploy custom configuration to your users.	"How to Deploy Custom Configurations" on page 10-3

Create Configurations by Organizing Checks and Folders

To customize the Model Advisor with MathWorks and custom checks, perform the following tasks:

- 1** Review the information in “Requirements for Customizing the Model Advisor” on page 6-2.
- 2** Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
- 3** Organize the checks into new and existing folders to create custom configurations. See “Organize and Deploy Model Advisor Checks”.
 - a** Identify which checks you want to include in your custom Model Advisor configuration. You can use MathWorks checks and/or custom checks.
 - b** Create the custom configurations using either of the following:
 - Model Advisor Configuration Editor - “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5.
 - A customization file - “Organize Customization File Checks and Folders” on page 8-11.
 - c** Verify the custom configuration. See “Verify and Use Custom Configurations” on page 8-17.
- 4** Optionally, deploy the custom configurations to your users. See “Organize and Deploy Model Advisor Checks”.
- 5** Verify that models comply with modeling guidelines. See “Run Model Checks” (Simulink).

Create Procedural-Based Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

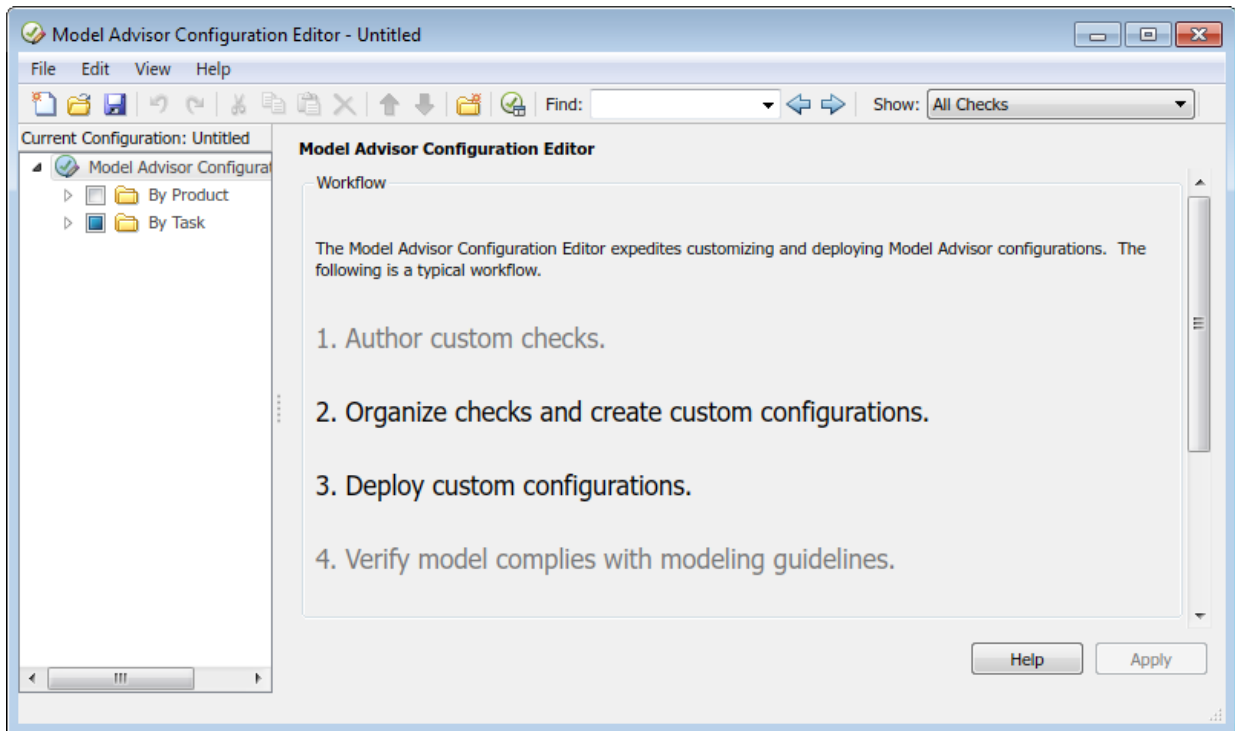
To create a procedural-based configuration, perform the following tasks:

- 1** Review the information in “Requirements for Customizing the Model Advisor” on page 6-2.
- 2** Decide on order of changes to your model.
- 3** Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.
- 4** Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
- 5** Organize the checks into procedures for a procedural-based configuration. See “Create Procedural-Based Configurations” on page 9-5.
 - a** Create procedures using the procedure API. For detailed information, see “Create Procedures Using the Procedures API” on page 9-2.
 - b** Create the custom configuration by using a customization file. See “Organize Customization File Checks and Folders” on page 8-11.
 - c** Verify the custom configuration as described in “Verify and Use Custom Configurations” on page 8-17.
- 6** Optionally, deploy the custom configurations to your users. For detailed information, see “Organize and Deploy Model Advisor Checks”.
- 7** Verify that models comply with modeling guidelines. For detailed information, see “Run Model Checks” (Simulink).

Organize Checks and Folders Using the Model Advisor Configuration Editor

Overview of the Model Advisor Configuration Editor

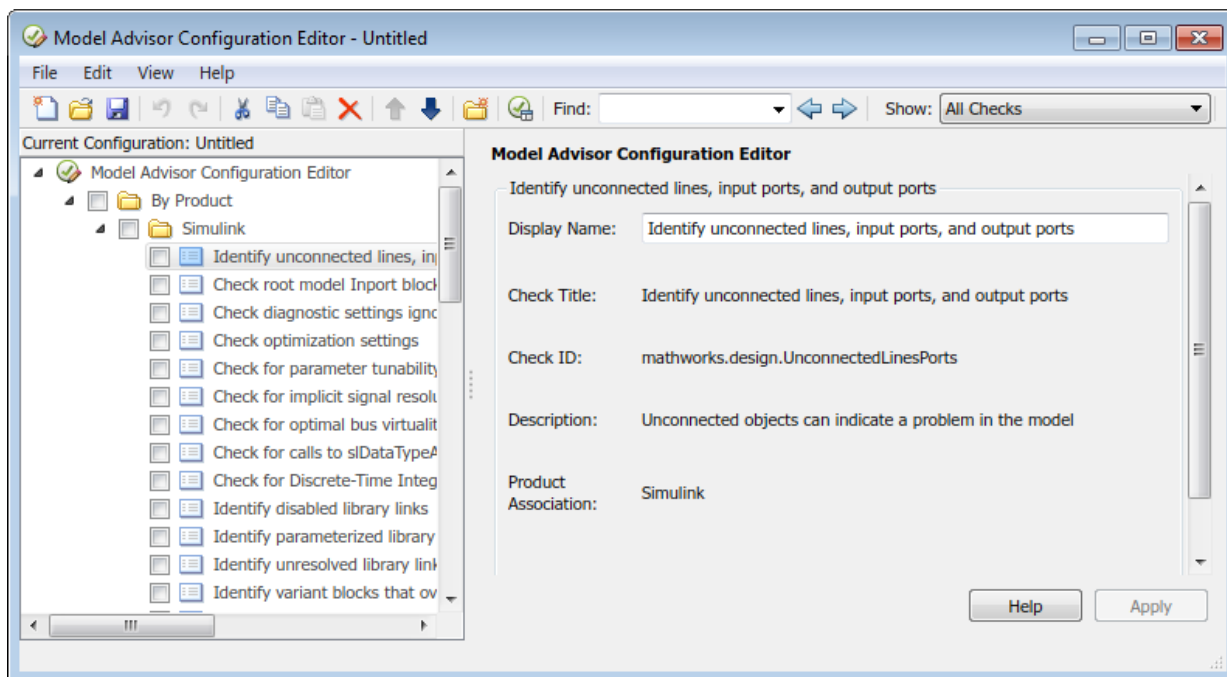
When you start the Model Advisor Configuration Editor, two windows open; the Model Advisor Configuration Editor and the Model Advisor Check Browser. The Configuration Editor window consists of two panes: the Model Advisor Configuration Editor hierarchy and the Workflow. The Model Advisor Configuration Editor hierarchy lists the checks and folders in the current configuration. The Workflow on the right shows the common workflow you use to create a custom configuration.



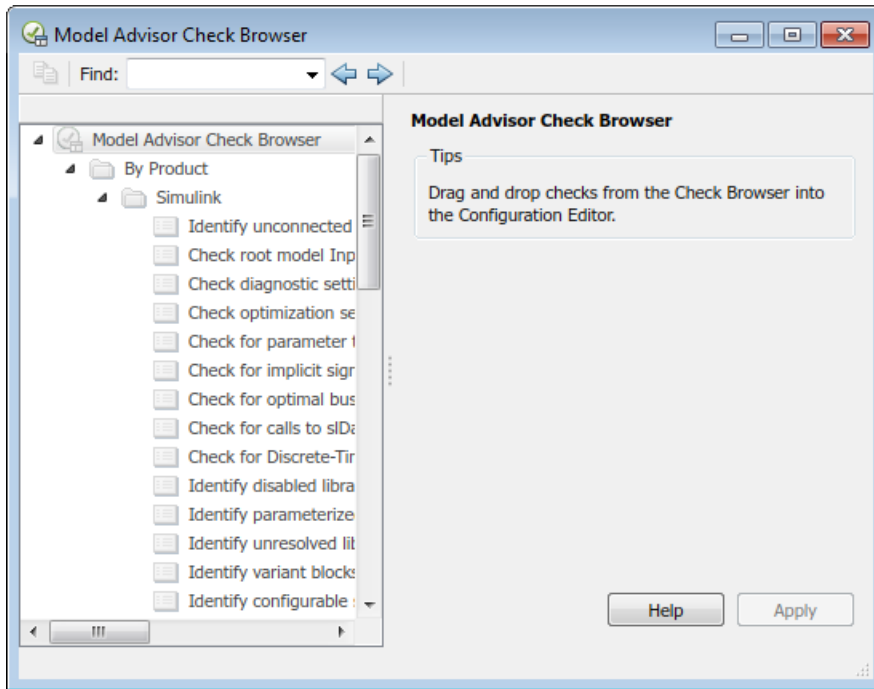
Model Advisor Configuration Editor

If you want the Model Advisor Configuration Editor hierarchy to list only the checks configured for edit-time checking, in the **Show** field, select **Edit-Time Supported Checks**. Or, in the model window, select **Analysis > Model Advisor > Configure Advisor Edit-Time Checks**.

When you select a folder or check in the Model Advisor Configuration Editor hierarchy, the Workflow pane changes to display information about the check or folder. You can change the display name of the check or folder in this pane.



The Model Advisor Check Browser window includes a read-only list of available checks. If you delete a check in the Model Advisor Configuration Editor, you can retrieve a copy of it from the Model Advisor Check Browser.



Model Advisor Check Browser

Using the Model Advisor Configuration Editor, you can perform the following actions.

To...	Select...
Create new configurations	File > New
Find checks and folders in the Model Advisor Check Browser	View > Check Browser
Add checks and folders to the configuration	Edit > Copy Edit > Paste Edit > New folder The check or folder and drag and drop
Remove checks and folders from the configuration	Edit > Delete Edit > Cut

To...	Select...
Reorder checks and folders	Edit > Move up Edit > Move down The check or folder and drag and drop
Rename checks and folders Note MathWorks folder display names are restricted. When you rename a folder, you cannot use the restricted display names.	The check or folder and edit Display Name in right pane.
Allow or gray out the check box control for checks and folders Tip This capability is equivalent to enabling checks, described in “Display and Enable Checks” on page 7-41.	Edit > Enable Edit > Disable
Save the configuration as a MAT file for use and distribution	File > Save File > Save As
Set the configuration so it opens by default in the Model Advisor	File > Set Current Configuration as Default
Restore the MathWorks default configuration	File > Restore Default Configuration
Load and edit saved configurations	File > Open

Start the Model Advisor Configuration Editor

Before starting the Model Advisor Configuration Editor, verify that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.

Note

- The Model Advisor Configuration Editor uses the `s\lprj` folder in the code generation folder (Simulink). If the `s\lprj` folder does not exist in the code generation folder, the Model Advisor Configuration Editor creates it.
-

- 1 To include custom checks in the new Model Advisor configuration, update the Simulink environment to include your `sl_customization.m` file.
- 2 Start the Model Advisor Configuration Editor.

To start the Model Advisor Configuration Editor...	Do this:
Programmatically	At the MATLAB command line, enter <code>Simulink.ModelAdvisor.openConfigUI</code> .
From the Model Advisor	<ol style="list-style-type: none"> a Start the Model Advisor. b Select Settings > Open Configuration Editor.

The Model Advisor Configuration Editor and Model Advisor Check Browser windows open.

- 3 Optionally, to edit an existing configuration in the Model Advisor Configuration Editor window:
 - a Select **File > Open**.
 - b In the Open dialog box, navigate to the configuration file that you want to edit.
 - c Click **Open**.

Organize Checks and Folders Using the Model Advisor Configuration Editor

The following tutorial steps you through creating a custom configuration.

- 1 Open the Model Advisor Configuration Editor at the MATLAB command line by entering `Simulink.ModelAdvisor.openConfigUI`.
- 2 In the Model Advisor Configuration Editor, in the left pane, delete the **By Product** and **By Task** folders, to start with a blank configuration.
- 3 Select the root node which is labeled Model Advisor Configuration Editor.
- 4 In the toolbar, click the **New Folder** button to create a folder.
- 5 In the left pane, select the new folder.
- 6 In the right pane, edit **Display Name** to rename the folder. For the purposes of this tutorial, rename the folder to **Review Optimizations**.

- 7 In the Model Advisor Check Browser window, in the **Find** field, enter optimization to find **Simulink > Check optimization settings**.
- 8 Drag and drop **Check optimization settings** into **Review Optimizations**.
- 9 In the Model Advisor Check Browser window, find **Simulink Check > Modeling Standards > DO-178C/DO-331Checks > Check safety-related optimization settings**.
- 10 Drag and drop **Check safety-related optimization settings** into **Review Optimizations**.
- 11 In the Model Advisor Configuration Editor window, expand **Review Optimizations**.
- 12 Rename **Check optimization settings** to **Check Simulink optimization settings**.
- 13 Select **File > Save As** to save the configuration.
- 14 Name the configuration `optimization_configuration.mat`.
- 15 Close the Model Advisor Configuration Editor window.

Tip To move a check to the first position in a folder:

- 1 Drag the check to the second position.
 - 2 Right-click the check and select **Move up**.
-

See Also

`Simulink.ModelAdvisor | ModelAdvisor.Check`

Related Examples

- “Update the Environment to Include Your `sl_customization` File” on page 8-17

Organize Customization File Checks and Folders

Customization File Overview

The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

Function	Description	Required or Optional
<code>sl_customization()</code>	Registers custom checks and tasks, folders with the Simulink customization manager at startup. See “Register Checks” on page 7-35.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Define Custom Checks” on page 7-40.	Required for custom checks and to add custom checks to the By Product folder.
One or more task definitions	Defines custom tasks. See “Define Custom Tasks” on page 8-13.	Required to add custom checks to the Model Advisor, except when adding the checks to the By Product folder. Write one task for each check that you add to the Model Advisor.
One or more groups	Defines custom groups. See “Define Custom Tasks” on page 8-13.	Required to add custom tasks to new folders in the Model Advisor, except when adding a new subfolder to the By Product folder. Write one group definition for each new folder.

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

Register Tasks and Folders

Create `sl_customization` Function

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

Tip

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

The `sl_customization` function accepts one argument, a customization manager object, as in this example:

```
function sl_customization(cm)
```

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

Register Tasks and Folders

The customization manager provides the following methods for registering custom tasks and folders:

- `addModelAdvisorTaskFcn (@factorygroupDefinitionFcn)`

Registers the tasks that you define in `factorygroupDefinitionFcn` to the **By Task** folder of the Model Advisor.

The `factorygroupDefinitionFcn` argument is a handle to the function that defines the checks to add to the Model Advisor as instances of the `ModelAdvisor.FactoryGroup` class.

- `addModelAdvisorTaskAdvisorFcn (@taskDefinitionFcn)`

Registers the tasks and folders that you define in *taskDefinitionFcn* to the folder in the Model Advisor that you specify using the `ModelAdvisor.Root.publish` method or the `ModelAdvisor.Group` class.

The *taskDefinitionFcn* argument is a handle to the function that defines custom tasks and folders. Simulink adds the checks and folders to the Model Advisor as instances of the `ModelAdvisor.Task` or `ModelAdvisor.Group` classes.

The following example shows how to register custom tasks and folders:

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

Define Custom Tasks

Add Check to Custom or Multiple Folders Using Tasks

You can use custom tasks for adding checks to the Model Advisor, either in multiple folders or in a single, custom folder. You define custom tasks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Task` class. Define one instance of this class for each custom task that you want to add to the Model Advisor. Then register the custom task. The following sections describe how to define custom tasks.

To add a check to multiple folders or a single, custom folder:

- 1 Create a check using the `ModelAdvisor.Check` class.
- 2 Register a task wrapper for the check.
- 3 If you want to add the check to folders that are not already present, register and create the folders using the `ModelAdvisor.Group` class.
- 4 Add a check to the task using the `ModelAdvisor.Task.setCheck` method.
- 5 Add the task to each folder using the `ModelAdvisor.Task.addTask` method and the task ID.

Create Custom Tasks Using MathWorks Checks

You can add MathWorks checks to your custom folders by defining the checks as custom tasks. When you add the checks as custom tasks, you identify checks by the check ID.

To find MathWorks check IDs:

- 1 In the Model Advisor, select **View > Source** tab.
- 2 Navigate to the folder that contains the MathWorks check.
- 3 In the right pane, click **Source**. The Model Advisor displays the **Title**, **TitleID**, and **Source** information for each check in the folder.
- 4 Select and copy the **TitleID** of the check that you want to add as a task.

Display and Enable Tasks

The **Visible**, **Enable**, and **Value** properties interact the same way for tasks as they do for checks.

Define Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor using the following guidelines:

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Task Definition Function

The following example shows a task definition function. This function defines three tasks.

Define Custom Folders

About Custom Folders

Use folders to group checks in the Model Advisor by functionality or usage. You define custom folders in:

- A factory group definition function that specifies the properties of each instance of the `ModelAdvisor.FactoryGroup` class.
- A task definition function that specifies the properties of each instance of the `ModelAdvisor.Group` class.

Define one instance of the group classes for each folder that you want to add to the Model Advisor.

Add Custom Folders

To add a custom folder:

- 1 Create the folder using the `ModelAdvisor.Group` or `ModelAdvisor.FactoryGroup` classes.
- 2 Register the folder.

Define Where Custom Folders Appear

You can specify the location of custom folders within the Model Advisor using the following guidelines:

- To define a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To define a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Note To define a new folder in the **By Product** folder, use the `ModelAdvisor.Root.publish` method within a custom check. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

Group Definition

The following examples shows a group definition. The definition places the tasks inside a folder called **My Group** under the **Model Advisor** root. The task definition function includes this group definition.

The following example shows a factory group definition function. The definition places the checks into a folder called **Demo Factory Group** inside of the **By Task** folder.

Customization Example

The Simulink Check software provides an example that shows how to customize the Model Advisor by adding:

- Custom checks
- Check input parameters
- Check actions
- Check list views to call the Model Advisor Result Explorer
- Custom tasks to include the custom checks in the Model Advisor
- Custom folders for grouping the checks
- Custom procedures

The example also provides the source code of the `sl_customization.m` file that executes the customizations.

To run the example:

- 1 At the MATLAB command line, type `slvnvdemo_mdadv`.
- 2 Follow the instructions in the model.

See Also

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

Related Examples

- “Update the Environment to Include Your `sl_customization` File” on page 8-17

More About

- “Define Custom Checks” on page 7-40
- “Display and Enable Checks” on page 7-41
- “Register Checks” on page 7-35

Verify and Use Custom Configurations

Update the Environment to Include Your `sl_customization` File

When you start Simulink, it reads customization (`sl_customization.m`) files. If you change the contents of your customization file, update your environment by performing these tasks:

- 1 If you previously started the Model Advisor:
 - a Close the model from which you started the Model Advisor
 - b Clear the data associated with the previous Model Advisor session by removing the `slprj` folder from your code generation folder (Simulink).
- 2 At the MATLAB command line, enter:

```
sl_refresh_customizations
```

If you have created custom checks, at the MATLAB command line, then also enter:

```
Advisor.Manager.refresh_customizations
```
- 3 Open your model.
- 4 Start the Model Advisor.

Verify Custom Configurations

To verify a custom configuration:

- 1 If you created custom checks, or created the custom configuration using the `sl_customization` method, update the Simulink environment.
- 2 Open a model.
- 3 From the model window, start the Model Advisor.
- 4 Select **Settings > Load Configuration**. If you see a warning that the Model Advisor report corresponds to a different configuration, click **Load** to continue.
- 5 In the Open dialog box, navigate to and select your custom configuration.
- 6 When the Model Advisor reopens, verify that the configuration contains the new folders and checks. For example, the **Review Optimizations** folder and the **Check Simulink optimization settings** and **Check safety-related optimization settings** checks.

- 7 Optionally, run the checks.

See Also

More About

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

Customize Model Advisor Check for Nondefault Block Attributes

You can customize the list of nondefault block parameters that are flagged by the Model Advisor MAAB check **Check for Nondefault Block Attributes** (Check ID: `mathworks.maab.db_0140`).

- 1 In the Model Advisor, select **Settings > Open Configuration Editor**.
- 2 In the **Find** field, enter `db_0140` and press **Enter**.
- 3 The Model Advisor Configuration Editor window displays the check **Check for Nondefault Block Attributes**. On the right pane, under **Input Parameters > Standard**, select **Custom**. The **Custom** setting enables editing of the parameter list.
- 4 In the table, find the block type for which you want to change the nondefault parameter list. Under **Parameter**, select a cell to edit. The parameters are separated with spaces.
- 5 Delete or add a parameter name that corresponds to the **BlockType**. For example, to remove the rounding method parameter from the check for each gain block, find **Gain** under **BlockType**. Under **Parameter**, delete the parameter name `RndMeth`.
Check ID: `mathworks.maab.db_0140` no longer checks for the display of nondefault rounding methods from gain blocks' annotations.


See Also

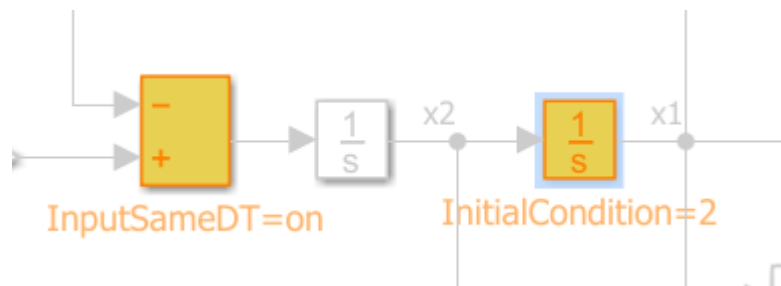
More About

- “Check for nondefault block attributes”
- “Customize Model Advisor Check for Nondefault Block Attributes” on page 8-19
- `db_0140`: Display of basic block parameters

Automatically Fix Display of Nondefault Block Parameters

To conform with Model Advisor MAAB check **Check for Nondefault Block Attributes** (Check ID: `mathworks.maab.db_0140`), you can use the **Add nondefault values into block annotation** button to automatically add descriptive text to the model editor window.

- 1 At the command prompt, type `vdp` and press **Enter**. The Van der Pol equation model opens in the Simulink editor window.
- 2 The model has two blocks which do not display nondefault values as annotations. Run the Model Advisor from **Analysis > Model Advisor > Model Advisor**.
- 3 On the left pane, select **By Product > Simulink Check > Modeling Standards > MathWorks Automotive Advisory Board Checks**. On the right pane, run the check by selecting **Run Selected Checks**.
- 4 The Model Advisor runs the check and displays a warning for the integrator block that has a nonzero initial condition not currently displayed. On the Model Advisor toolbar, select **Enable highlighting** () to highlight the blocks causing the warning.
- 5 In the right pane of the Model Advisor window, select **Add nondefault values into block annotation** to automatically add the nondefault attribute and value to the integrator block's annotation. Model Advisor displays `InitialCondition = 2`.



- 6 Run the check again to clear the warning.

See Also

More About

- “Check for nondefault block attributes”
- “Automatically Fix Display of Nondefault Block Parameters” on page 8-20
- db_0140: Display of basic block parameters
- “Select and Run Model Advisor Checks” (Simulink)

Create Procedural-Based Model Advisor Configurations

Create Procedures

What Is a Procedure?

A procedure is a series of checks. The checks in a procedure depend on passing the previous checks. If Check A is the first check in a procedure and Check B follows, the Model Advisor does not run Check B until Check A passes. Checks A and B can be either custom or provided by MathWorks.

You create procedures with the `ModelAdvisor.Procedure` class API. You first add the checks to tasks, which are wrappers for the checks. The tasks are added to procedures.

When creating procedural checks, be aware of potential conflicts with the checks. Verify that it is possible to pass both checks.

Create Procedures Using the Procedures API

You use the `ModelAdvisor.Procedure` class to create procedural checks.

- 1 Add each check to a task using the `ModelAdvisor.Task.setCheck` method. The task is a wrapper for the check. You cannot add checks directly to procedures.
- 2 Add each task to a procedure using the `ModelAdvisor.Procedure.addTask` method.

Define Procedures

You define procedures in a procedure definition function that specifies the properties of each instance of the `ModelAdvisor.Procedure` class. Define one instance of the procedure class for each procedure that you want to add to the Model Advisor. Then register the procedure using the `ModelAdvisor.Root.register` method.

Add Subprocedures and Tasks to Procedures

You can add subprocedures or tasks to a procedure. The tasks are wrappers for checks.

- Use the `ModelAdvisor.Procedure.addProcedure` method to add a subprocedure to a procedure.
- Use the `ModelAdvisor.Procedure.addTask` method to add a task to a procedure.

Define Where Procedures Appear

You can specify where the Model Advisor places a procedure using the `ModelAdvisor.Group.addProcedure` method.

Procedure Definition

The following code example adds procedures to a group:

```
%Create three procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure3');

%Create a group
MAG = ModelAdvisor.Group('com.mathworks.sample.myGroup');

%Add the three procedures to the group
addProcedure(MAG, MAP1);
addProcedure(MAG, MAP2);
addProcedure(MAG, MAP3);

%register the group and procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAG);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

The following code example adds subprocedures to a procedure:

```
%Create a procedure
MAP = ModelAdvisor.Procedure('com.mathworks.example.Procedure');

%Create 3 sub procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub1');
MAP2=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub2');
MAP3=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub3');

%Add sub procedures to procedure
addProcedure(MAP, MAP1);
addProcedure(MAP, MAP2);
addProcedure(MAP, MAP3);

%register the procedures
```

```
mdladvRoot = ModelAdvisor.Root;  
mdladvRoot.register(MAP);  
mdladvRoot.register(MAP1);  
mdladvRoot.register(MAP2);  
mdladvRoot.register(MAP3);
```

The following code example adds tasks to a procedure:

```
%Create three tasks  
MAT1=ModelAdvisor.Task('com.mathworks.tasksample.myTask1');  
MAT2=ModelAdvisor.Task('com.mathworks.tasksample.myTask2');  
MAT3=ModelAdvisor.Task('com.mathworks.tasksample.myTask3');  
  
%Create a procedure  
MAP = ModelAdvisor.Procedure('com.mathworks.tasksample.myProcedure');  
  
%Add the three tasks to the procedure  
addTask(MAP, MAT1);  
addTask(MAP, MAT2);  
addTask(MAP, MAT3);  
  
%register the procedure and tasks  
mdladvRoot = ModelAdvisor.Root;  
mdladvRoot.register(MAP);  
mdladvRoot.register(MAT1);  
mdladvRoot.register(MAT2);  
mdladvRoot.register(MAT3);
```

See Also

[ModelAdvisor.Procedure](#) | [ModelAdvisor.Procedure.addProcedure](#) |
[ModelAdvisor.Procedure.addTask](#) | [ModelAdvisor.Root.register](#) |
[ModelAdvisor.Task.setCheck](#)

Related Examples

- “Create Procedural-Based Configurations” on page 9-5

More About

- “Define Custom Tasks” on page 8-13

Create Procedural-Based Configurations

Overview of Procedural-Based Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

To create a procedural-based configuration, perform the following tasks:

- 1** Review the information in “Requirements for Customizing the Model Advisor” on page 6-2.
- 2** Decide on order of changes to your model.
- 3** Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.
- 4** Optionally, author custom checks in a customization file. See “Create Model Advisor Checks”.
- 5** Organize the checks into procedures for a procedural-based configuration. See “Create Procedural-Based Configurations” on page 9-5.
 - a** Create procedures using the procedure API. For detailed information, see “Create Procedures Using the Procedures API” on page 9-2.
 - b** Create the custom configuration by using a customization file. See “Organize Customization File Checks and Folders” on page 8-11.
 - c** Verify the custom configuration as described in “Verify and Use Custom Configurations” on page 8-17.
- 6** Optionally, deploy the custom configurations to your users. For detailed information, see “Organize and Deploy Model Advisor Checks”.
- 7** Verify that models comply with modeling guidelines. For detailed information, see “Run Model Checks” (Simulink).

Create a Procedural-Based Configuration

In this example, you examine a procedural-based configuration.

- 1 At the MATLAB command line, type `slvnvdemo_mdladv`.
- 2 In the model window, select **View demo sl_customization.m**. The `sl_customization.m` file opens in the MATLAB Editor window.

The file contains three checks created in the function `defineModelAdvisorChecks`:

- `ModelAdvisor.Check('com.mathworks.sample.Check1')` - Checks Simulink block fonts.
- `ModelAdvisor.Check('com.mathworks.sample.Check2')` - Checks Simulink window screen color.
- `ModelAdvisor.Check('com.mathworks.sample.Check3')` - Checks model optimization settings.

Each check has a set of fix actions.

- 3 In the `sl_customization.m` file, examine the function `defineTaskAdvisor`.

- The `ModelAdvisor.Procedure` class API creates procedures `My Procedure` and `My sub_Procedure`:

```
% Define procedures
MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');
MAP.DisplayName='My Procedure';

MAP_sub = ModelAdvisor.Procedure('com.mathworks.sample.sub_ProcedureSample');
MAP_sub.DisplayName='My sub_Procedure';
```

- The `ModelAdvisor.Task` class API creates tasks `MAT4`, `MAT5`, and `MAT6`. The `ModelAdvisor.Task.setCheck` method adds the checks to the tasks:

```
% Define tasks
MAT4 = ModelAdvisor.Task('com.mathworks.sample.TaskSample4');
MAT4.DisplayName='Check Simulink block font';
MAT4.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT4);

MAT5 = ModelAdvisor.Task('com.mathworks.sample.TaskSample5');
MAT5.DisplayName='Check Simulink window screen color';
MAT5.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT5);

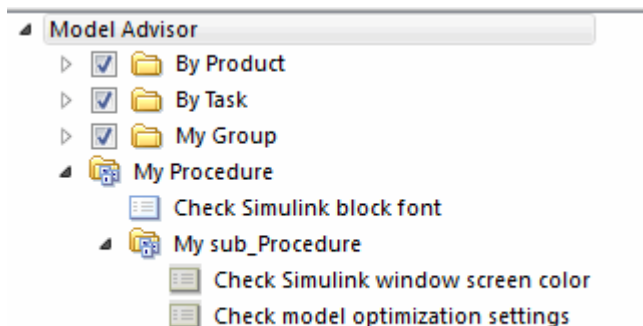
MAT6 = ModelAdvisor.Task('com.mathworks.sample.TaskSample6');
MAT6.DisplayName='Check model optimization settings';
```

```
MAT6.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT6);
```

- The `ModelAdvisor.Procedure.addTask` method adds task MAT4 to My Procedure and tasks MAT5 and MAT6 to My `sub_Procedure`. The `ModelAdvisor.Procedure.addProcedure` method adds My `sub_Procedure` to My Procedure:

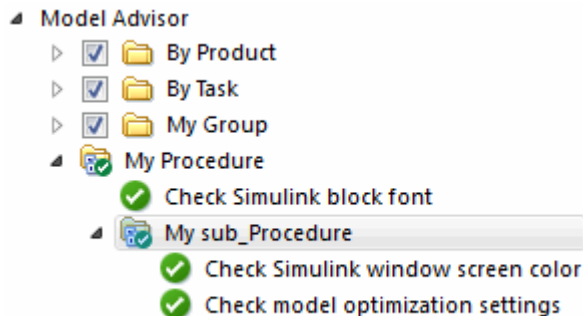
```
% Add tasks to procedures:
% Add Task4 to MAP
MAP.addTask(MAT4);
% Now Add Task5 and Task6 to MAP_sub
MAP_sub.addTask(MAT5);
MAP_sub.addTask(MAT6);
% Include the Sub-Procedure in the Procedure
MAP.addProcedure(MAP_sub);
```

- 4 From the model window, select **Analysis > Model Advisor > Model Advisor** to open the Model Advisor.
- 5 A **System Selector – Model Advisor** dialog box opens. Click **OK**. The **Model Advisor** window opens.
- 6 In the left pane, expand **My Procedure > My sub_Procedure**. The Check Simulink block font check is in the My Procedure folder. My `sub_Procedure` contains Check Simulink window screen color and Check model optimization settings.



- 7 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run All**. The Model Advisor Check Simulink block font check fails. The Model Advisor does not check the remaining two checks in the My `sub_Procedure` folder. Running the checks in the My `sub_Procedure` folder depends on passing the Check Simulink block font check.
- 8 In the **Action** section of the Model Advisor dialog box, click **Fix block fonts**.

- 9 In the left pane of the Model Advisor, select My Procedure. In the right pane of the Model Advisor, click **Run All**. The Check Simulink block font check passes. The Model Advisor runs the Check Simulink window screen color check. This check fails and the Model Advisor stops checking.
- 10 In the **Action** section of the Model Advisor dialog box, click **Fix window screen color**.
- 11 In the left pane of the Model Advisor, select My sub Procedure. In the right pane of the Model Advisor, click **Run All**. The Check Simulink window screen color check passes. The Model Advisor runs the Check model optimization settings check. This check warns.
- 12 In the **Action** section of the Model Advisor dialog box, click **Fix model optimization settings**.
- 13 In the left pane of the Model Advisor, select Check model optimization settings. In the right pane of the Model Advisor, click **Run This Task**. The Check model optimization settings check passes.



See Also

`ModelAdvisor.Procedure` | `ModelAdvisor.Procedure.addProcedure` |
`ModelAdvisor.Procedure.addTask` | `ModelAdvisor.Root.register` |
`ModelAdvisor.Task.setCheck`

More About

- “Create Procedures” on page 9-2

- “Define Custom Checks” on page 7-40

Deploy Custom Configurations

Overview of Deploying Custom Configurations

About Deploying Custom Configurations

When you create a custom configuration, often you deploy the custom configuration to your development group. Deploying the custom configuration allows your development group to review models using the same checks.

After you create a custom configuration, you can use it in the Model Advisor, or deploy the configuration to your users. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

Deploying Custom Configurations Workflow

When you deploy custom configurations, you:

- 1** Optionally author custom checks, as described in “Create Model Advisor Checks”.
- 2** Organize checks and folders to create custom configurations, as described in “Create Custom Configurations” on page 8-2.
- 3** Deploy the custom configuration to your users, as described in “How to Deploy Custom Configurations” on page 10-3.

How to Deploy Custom Configurations

To deploy a custom configuration:

- 1 Determine which files to distribute. You might need to distribute more than one file.

If You...	Using the...	Distribute...
Created custom checks	Customization file	<ul style="list-style-type: none"> • <code>sl_customization.m</code> • Files containing check and action callback functions (if separate)
Organized checks and folders	Model Advisor Configuration Editor	Configuration MAT file
	Customization file	<code>sl_customization.m</code>

- 2 Distribute the files and tell the user to include these files on the MATLAB path.
- 3 Instruct the user to load the custom configuration.

See Also

Related Examples

- “Automatically Load and Set the Default Configuration” on page 10-5
- “Manually Load and Set the Default Configuration” on page 10-4

Manually Load and Set the Default Configuration

When you use the Model Advisor, you can load any configuration. Once you load a configuration, you can set it so that the Model Advisor use that configuration every time you open the Model Advisor.

- 1 Open the Model Advisor.
- 2 Select **Settings > Load Configuration**.
- 3 In the Open dialog box, navigate to and select the configuration file that you want to edit.
- 4 Click **Open**.

Simulink reloads the Model Advisor using the new configuration.

- 5 Optionally, when the Model Advisor opens, set the current configuration as the default by selecting **File > Set Current Configuration as Default**.

See Also

Related Examples

- “Automatically Load and Set the Default Configuration” on page 10-5
- “Update the Environment to Include Your sl_customization File” on page 8-17

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5
- “Register Checks” on page 7-35

Automatically Load and Set the Default Configuration

When you use the Model Advisor, you can automatically set the default configuration by modifying an `sl_customization.m` file.

- 1 Place a configuration MAT file on your MATLAB path.
- 2 Modify your `sl_customization.m` file by adding the function:

```
function [checkCellArray taskCellArray] = ModelAdvisorProcessFunction ...  
    (stage, system, checkCellArray, taskCellArray)  
    switch stage  
        case 'configure'  
            ModelAdvisor.setConfiguration('qeAPIConfigFilePath.mat');  
        end
```

In the function, replace `qeAPIConfigFilePath.mat` with the name of the configuration MAT file in step 1.

- 3 The `sl_customization.m` file is loaded every time you start the Model Advisor, using `qeAPIConfigFilePath.mat` as the default configuration.

Tip You can restore the MathWorks default configuration by selecting **File > Restore Default Configuration**.

See Also

Related Examples

- “Manually Load and Set the Default Configuration” on page 10-4
- “Update the Environment to Include Your `sl_customization` File” on page 8-17

More About

- “Organize Checks and Folders Using the Model Advisor Configuration Editor” on page 8-5
- “Register Checks” on page 7-35

